DYNAMIC ORDERED INHERITANCE AND FLEXIBLE METHOD DISPATCH

Paradigm, Concepts, and Use Cases

Dissertation zur Erlangung des akademischen Grades
Doktor der technischen Wissenschaften
in der Studienrichtung Informatik

Angefertigt am Institut für Praktische Informatik

Betreuung:
o. Univ.-Prof. Dr. Hanspeter Mössenböck

Von:
Dipl.-Ing. Gerhard Schaber

Gutachter:
o. Univ.-Prof. Dr. Hanspeter Mössenböck
o. Univ.-Prof. Dr. Wolfgang Pree

Linz, Juli 2003

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbständig und ohne fremde Hilfe verfasst habe. Ich habe keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche kenntlich gemacht.

Linz, 2003

Abstract

The aim of component-oriented programming is to construct large software applications out of sets of small components with low complexity, rather than building monolithic applications which usually are difficult and expensive to maintain and extend. This enables rapid application development, makes maintenance easier, and therefore also reduces development costs.

Using third-party components further reduces the efforts, but it also has its shortcomings. Developers are often confronted with the lack of flexibility of third-party components to be adapted to their needs. Companies offering components are confronted with the problem to anticipate how developers (customers) and users might want to adapt the components. They have to build in hooks that allow to customize and extend the components.

The other way is to offer a solution that allows developers and users to dynamically modify components without being limited to a few hooks. However, current approaches for component modification are still not satisfying or sufficient. The support is often only available at compile- or link-time. Current approaches that provide generic (unanticipated) support for dynamic component modification are not sufficient.

This thesis presents a small framework which supports dynamic component modification through a flexible method dispatch mechanism. Method dispatch is the mechanism for delegating a method call from the object where the method has been invoked to the object, respectively class where the method is implemented. The framework also provides proper base classes and template configurations for components, which can help to significantly simplify development of applications and their components. The framework is called PCoC which is the abbreviation for Prioritized Coupling and Control of objects and components. Many of its concepts and facilities can also be deployed independently of each other if necessary.

We use a reflection-based (meta-programming) approach where operations are treated as first-class objects. So-called Dispatchers, similar to the Microsoft .NET delegates (type-safe method-pointers), are responsible for dynamic method dispatch. The approach enables to add, remove, or replace operations at runtime, and to attach listeners to method objects, which are notified before or after each call. It simulates delegation—a dispatch mechanism in which an object (the receiver) delegates a message to another object (the method holder and delegate) in response to a message. The delegate carries out the request on behalf of the original object, and may send subsequent messages to the original receiver. This includes the invocation of methods.

A priority ranking of components determines in which order requests for operations are forwarded, which gives the framework its name PCoC. This ranking can change through user interaction (focus change in the GUI), or explicitly.

There are implementations of PCoC in C++ and Java. These are parts of two integrated development environments (IDEs) available on various platforms, including Sun Solaris, Linux, and the Microsoft Windows NT line.

Kurzfassung

Das Ziel von Komponenten-orientierter Programmierung ist es, grosse Anwendungen aus einer Menge von kleinen Komponenten mit niedriger Komplexität zusammenzustellen, anstatt monolithische Anwendungen zu bauen, die schwierig und teuer zu warten und erweitern sind. Dieser Ansatz unterstützt eine schnelle Entwicklung von Anwendungsprogrammen und deren Wartung, und senkt damit auch Entwicklungskosten.

Die Verwendung von Komponenten von Drittanbietern reduziert den Aufwand nochmals, aber hat auch Schattenseiten. Entwickler sind oft mit einem Mangel an Flexibilität von solchen Komponenten konfrontiert, welche sich oft nicht an Ihre speziellen Bedürfnisse anpassen lassen. Firmen, die Komponenten anbieten, sind ihrerseits oft mit dem Problem konfrontiert, herausfinden und voraussehen zu müssen, wie (künftige) Benutzer der Komponenten diese erweitern und anpassen wollen.

Ein anderer Ansatz ist eine allgemeine Lösung, die es Entwicklern und Benutzern erlaubt, Komponenten dynamisch zu verändern, ohne auf wenige vorgesehene Callback-Funktionen und Konfigurationsmöglichkeiten beschränkt zu sein. Jedoch sind aktuelle Ansätze für dynamische Komponentenmodifikation immer noch nicht sehr befriedigend und ausgereift. Modifikationen sind meistens nur zur Compile- oder Linkzeit möglich. Aktuelle Ansätze für eine allgemeine Unterstützung für dynamische Komponentenmodifikation sind noch nicht zufrieden stellend.

Diese Arbeit präsentiert ein kleines Framework, das dynamische Komponentenmodifikation durch einen flexiblen Verteilungsmechanismus für Methodenaufrufe realisiert. Dieser Mechanismus delegiert dynamische Methodenaufrufe von den Objekten, wo Operationen aufgerufen werden, zu den Objekten, wo die entsprechenden Methoden implementiert sind. Das Framework bietet auch geeignete Basisklassen und Konfigurationsvorlagen für Komponenten, die die Entwicklung von Anwendungen und deren Komponenten um vieles vereinfachen können. Die Bezeichnung des Frameworks ist PCoC, als Abkürzung für Prioritized Coupling and Control of objects and components.

Wir verwenden einen Reflection-basierten Ansatz, bei dem Operationen als First-Class-Objekte behandelt werden. Sogenannte Dispatcher, ähnlich der Microsoft .NET Delegates (typsichere Methodenzeiger), sind für die Verteilung von dynamischen Methodenaufrufen zuständig. Der Ansatz ermöglicht das Hinzufügen, Entfernen, und Ersetzen von Operationen zur Laufzeit, und das Anmelden von Listener-Objekten, die vor und nach jedem Aufruf oder Statusänderung der jeweiligen Operation benachrichtigt werden. Der Ansatz simuliert Delegation—ein Verteilungsmechanismus für Methodenaufrufe, bei dem ein Objekt (der Empfänger eines Aufrufes) Aufrufe zu einem oder mehreren anderen Objekten (wo die Methode implementiert ist—der “Delegate” eines Aufrufes) weiterleitet. Ein Delegate führt die entsprechende Operation aus und kann selbst weitere Operationen am ursprünglichen Empfängerobjekt aufrufen, welche dann möglicherweise wiederum weitergeleitet werden.

Eine Reihung von Komponenten nach Priorität bestimmt in welcher Reihenfolge die Aufrufe von Operationen zu welchen Komponenten weitergeleitet werden, was dem Framework seinen Namen PCoC gibt. Diese Reihung kann einerseits durch Benutzinteraktion geändert werden (Fokus-Änderung in der Benutzerschnittstelle, behandelt durch das Framework), oder explizit im Client-Code.

Es gibt Implementierungen von PCoC in C++ and Java. Sie sind Teile von zwei integrierten Entwicklungsumgebungen (IDEs) auf Sun Solaris, Linux, und der Microsoft Windows NT-Linie.

Acknowledgments

First and foremost I want to express my gratitude to my advisor, Hanspeter Mössenböck. None of this would have been possible without his guidance and support. His constructive criticism of my thinking and writing helped to keep me on track, to develop new ideas, and to avoid pitfalls. His guidance on literature and research I should focus on was invaluable.

I would like to thank my parents, Waltraud and Georg Schaber, who made my education possible and supported by during my study.

I would like to express my appreciation to my sister, Karin Schaber, who gave me the inspiration and motivation that kept me on track. I have always admired her ambition and determination, and I still do. She has always been a person to whom I have looked up to, and a very good friend. Her advice regarding the writing of this thesis was also most helpful.

I also would like to express my appreciation to Michael Scharf who is a great software architect. We had many really good discussions. Our work as a team and all the nightly discussions we had were sources for many new ideas and steady motivation. The whole framework presented in this thesis would not have been possible without him.

My thanks go also to Werner Kurschl who supported me with good literature and advice.

Our technical writers, Jyo Samjee and Eric Strobl, helped to improve the English. I am truly thankful for their help. I appreciate the personality of both, and our many and extensive talks. Many thanks go to Eric who proof-read this thesis, and to Jyo who always had a good advice and is one of my best friends.

Last, but not least, I would like to thank my former employer, WindRiver Systems GmbH, more precisely Tomas Evensen and Rudolf Frauenschuh for giving me the permission to use concepts, documents, and source code I created during my work for WindRiver for this thesis. I have learned a lot in this company.

I started working for Takefive Software in Salzburg in 1997. The company was acquired first by Integrated Systems, and then by WindRiver Systems. I want to thank all my colleagues there for our great cooperation in the last years, and many good ideas. I appreciate all of them as great developers and colleagues: Helmut Haigermoser (who does great work as developer, workspace and system administrator, and Harry Potter and Lord of the Rings story teller), Walter Brunauer, Christian Linhart (the living C++ standard—he knows everything about C++), Max Weninger, Edi Vorauer, Robert Lichtenberger, Johann Draschwandtner, Toni Leherbauer, Martin Gutschelhofer, Laszlo Juhasz, Rudolf Zimmel (a great project manager).

Contents

1 Introduction

1.1 Motivation and Goals

In order to simplify the development of large applications, they can be constructed out of small *components* with low complexity. The assembly of applications from existing components should increase reuse, thus allowing developers to concentrate on value-added tasks and to produce software with high quality within a shorter time.

In the literature, the term *software component* is diversely defined. The notion of the term as found in [SZYPE98] on Page 34, is in line with this thesis. The book provides a detailed discussion about what a component is and what is not. Other definitions of the term and comments on the definitions can be found on Pages 164-168.

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

With this definition we find a comment that a software component must be a binary unit. We agree with this notion, but we use the term also for sets of class(es) together with a proper configuration (how a component or application is customized) from which a software component is built.

Component oriented programming promises the flexibility to adapt ready-made components to the user's needs. Component modification includes the replacement of components and operations by others that are better suited for specific tasks. It also includes the customization of the look & feel and other properties of components and whole applications.

When developing components, we often encounter one or more of the following requirements:

- the ability to add, remove, and replace data sources and targets at runtime
- the ability to add, remove, and modify functionality at runtime
- multiple reuse of components as instances in different applications or within the same application
- priority management, for example for interactive components and their operations
- single, broadcast, and filtered dispatch of operation requests
- deferred execution of operations
- application control via menus, toolbars, scripting, and remote requests—preferably without the need to implement separate dispatch functions for each operation and their different deployments

The need for the ability to dynamically add and remove data sources and targets is given in almost every interactive application. Think of a file-browser as source component, in which we interactively select a particular file. We might need to dynamically associate a specific target component with the selected file, depending on its file type. The target component may be a part of a newly installed application, and should be loaded when we double-click the file in the file-browser.

It is sometimes necessary to extend the associated target component by additional functionality. For example, we might want to extend a word-processor component by functionality to calculate particular statistics or to do additional highlighting of special phrases of a text document, which is not supported by the original component. In the case of third-party components, we usually do not have the source code which we could modify, so we need a mechanism that enables us to modify components dynamically. The ability to extend and modify a component is called *dynamic component modification*, or *dynamic component adaptation*.

In order to get a single look and feel for applications, and to save development time and costs, we should be able to reuse components in different applications or within the same application.

In certain cases it is required to switch between several implementations of an operation, class or whole component, and to introduce a ranking for the different implementations. For example, when applying visual effects on an image, or filters in a text-search dialog, then it is relevant in which order the effects or filters are applied. If the effects and filters are realized as classes with a method `apply`, we need assemblies, or more precisely ordered collections, where their instances can be put into. The ranking within such an assembly could determine in which order the effects, respectively filters, are applied. The user could interactively change the ranking, as well as the developer could change it in source code.

Special method dispatch strategies are needed for certain use cases. To invoke single methods is the most common case, but also *broadcasts* (multicasts) are needed, for example, for shutting down all services and saving all documents when an application is going to be terminated. In some cases, multicast messages should be sent only to certain components. We call this dispatch mode *filtered broadcast*.

It is necessary that time-consuming tasks are executed asynchronously, usually in a separate thread. This concept, called *deferred execution* or *late invocation*, helps to prevent that the corresponding application, respectively its current thread, is blocked, as it would be the case if the task was executed synchronously.

Most applications have a user interface, a scripting interface, and maybe an interface to control them remotely, for example via Web-browsers. Usually, for each interface and required operation, proper dispatch functions or hooks (callback methods) must be implemented. In addition to that, the same operations may be invoked from various places in the source code. To simplify development, it is reasonable to introduce a uniform method dispatch mechanism for handling all of these interfaces, and which is easy to adapt to new interfaces. The component developer can then concentrate on implementing the core functionality (the purpose) rather than having also to handle all kinds of interfaces over which the component functionality is exposed.

In addition to the already mentioned requirements, some applications have to be portable to various platforms. An architecture is required that is able to cope with small as well as huge applications, providing easy concepts and usage. In order to be able to find defects in the software, a facility to inspect objects and classes at runtime can be helpful. Such a facility to display information about objects is called *introspection* and is based on *reflection*. Reflection is the ability to obtain information about the fields, constructors and methods of any class or component. See *Developing JAVA Beans* [ENGLA97] on Pages 40-42 for a short introduction into Java reflection.

In short, we need a flexible, scalable, and easy to use approach for dynamic component modification with support for introspection.

In [KNIES99] on Pages 1f, Kniesel G. classifies known approaches of dynamic component modification according to:

- their need for preexisting hooks in the application as either suitable for *anticipated* or *unanticipated* changes
- the time when a modification is made as either *static*, *load-time* or *dynamic* (runtime)
- their ability to adapt whole component types or individual component instances as either *global* or *selective*. Selective approaches can be further classified as either *replacing* or *preserving*, depending on whether they replace an existing component instance by its modified version or let both be used simultaneously.

- the applied techniques as either *code-modification-based*, *wrapper-based* or *meta-level-based*.

The need of unanticipated dynamic component modification has been repeatedly pointed out in literature (e.g. [MAETZ97]).

When developing a component, we may not always know which hooks might be needed in the future by users of the component. We cannot always anticipate which enhancements users might want to make, and we usually do not want to make the source code available.

Global approaches affect classes, whereas selective approaches affect objects. A modification of a class has an impact on all its instances, whereas a modification of an object has only a local impact.

By introducing wrappers for components, the interfaces and functionality can be adapted to the client's needs without changing the original components. A wrapper acts as proxy (a surrogate or placeholder) of its associated component, and may load it on demand. If the component cannot be loaded, the wrapper can throw an error.

A more detailed discussion of this classification is given in Section 2.2.5.

Our goal is an approach that offers support for dynamic component modification and which meets the requirements mentioned earlier in this section.

1.2 A Flexible Method Dispatch Mechanism

With respect to the classification above, we introduce a flexible method dispatch mechanism that offers support for dynamic, selective, wrapper-based component modification. This approach is object-preserving when applied at instance-level (a generic component-instance wrapper handles dynamic operations; such an operation is represented as object with a reference to the wrapper). When integrated at class-level (wrappers for classes rather than instances; operations do not have a reference to a wrapper), it is global and meta-level-based. The flexible method dispatch is based on reflection, more precisely on dynamic method calls. No proprietary compiler and preprocessing of source code is necessary.

The approach enables to add, remove, or replace operations (method objects) at runtime, and to attach listeners to operations, which are notified before or after each call. We call these operations *Activities*. An Activity can either be directly added to a so-called *Activity Set*, or *acquired* from another. In our approach, each component is associated with an Activity Set. *Acquisition* denotes a *delegation* mechanism. That is, when an Activity Set (the child) acquires from another (the parent), and an operation is called on the child, and the child does not provide the operation, the call is delegated to the parent. A reference to the receiver (where the operation has been invoked) is passed with each operation, so that the parent can invoke subsequent operations on the original receiver (the child).

The difference to what we understand as delegation is that our approach uses sets of dynamic operations (Activity Sets) rather than objects. However, the dispatch algorithm is the same.

As opposed to class inheritance which is used to share behavior between classes, delegation (and acquisition) is used to share behavior between objects. The delegation concept is described in detail in Section 2.2.4.

Activities are treated as first-class objects. So-called Dispatchers, similar to the Microsoft .NET delegates (type-safe method-pointers), are responsible for dynamic method dispatch, or more precisely for delegating operation requests. The main difference to delegates is that Dispatchers can be used for delegation, whereas delegates can only be used for forwarding. With forwarding, the object where a method originally has been invoked (the receiver) is not known to the method holder (where the method is actually executed), and therefore the method holder cannot use it for subsequent method calls.

Dispatchers are automatically generated when an Activity is added to a component, respectively to its associated Activity Set, or acquired from another. Their type safety is ensured at runtime, whereas the type safety of .NET delegates is ensured at compile time.

Activities can be extended by aspects at runtime. Aspects are well-modularized reusable cross-cutting concerns in software: Code fragments can sometimes be separated into their main functionality and several aspects. Such an aspect applies to a set of code fragments, and thus is a means of reusability. Examples are plausibility checks for method arguments and return values. They also include tracing capabilities, additional data-types for algorithms, etc.

> “Aspect oriented programming does for concerns that are naturally cross-cutting what object-oriented programming does for concerns that are naturally hierarchical.” - [KICZAL00], Page 1.

1.3 The PCoC Framework

We present a small framework that supports the flexible method dispatch mechanism, and meets the requirements discussed earlier in this chapter. The framework also provides proper base classes and template configurations for components, which can help to significantly simplify development of applications and their components. The method dispatch mechanism including its support for different component interfaces is encapsulated in the base classes, as well as the processing of component startup, initialization, and termination. This approach allows developers to concentrate on value-added tasks such as implementing the core functionality (the purpose) of each component.

The framework is called PCoC as abbreviation for Prioritized Coupling and Control of objects and components. The name denotes the support for prioritized dynamic inheritance (acquisition), and the already mentioned generic, flexible method dispatch mechanism used for component interoperation (coupling) and application control. A priority ranking of acquired components determines in which order requests for operations are delegated. This ranking can change through user interaction (focus change in the GUI), or explicitly.

All operations of all components are automatically exposed in a generic way for their use in the GUI (menu bars, tool bars, etc.), for scripting, for component interoperation, and for remote control via XMLRPC (XML-based remote procedure calls), etc.

Many of the concepts and facilities of PCoC can be deployed independently of each other if necessary. There are implementations of the framework in C++ and Java. These are parts of two integrated development environments (IDEs) that have been released on various platforms, including Sun Solaris, Linux, and the Microsoft Windows NT line.

The Java version is used as a basis for this thesis.

Some concepts and source code in this thesis are intellectual property of WindRiver Systems, Inc., and must not be used exactly as presented here without explicit permission of the author or WindRiver Systems. However, they may help to find similar solutions. WindRiver Systems has contractually granted all rights to use and present the concepts that are described in this thesis to the author.

1.4 Deployment of PCoC

PCoC is made for all kinds of applications that require extensive user interaction or scripting. The goal is to provide a comfortable framework that reduces the implementation overhead for component interoperation and controlling by separating the component assembly from source code, providing built-in priority management of components, and a mechanism for dynamically dispatching method calls based on this priority management.

Priority management means that components are ranked by time of last usage. The most recently used component gets the highest priority. When sending a message to a group of components, the message is sent to the first component in the ranking that is interested in this message and which subsequently performs an operation associated with the message.

PCoC is not a competing component standard to CORBA, COM, .NET, JavaBeans, etc. Rather, it works together with, or on top of, these standards.

It is also not a reasonable basis for applications without a graphical user interface, for applications providing only one interactive component, or generally for limited-functionality applications. Although the framework would be a too big an overhead for such small applications, it may nevertheless be used to quickly achieve a running and extendable application.

When developing smaller applications, in most cases standard operations (methods, functions) and component interfaces would be sufficient. When developing larger applications (over 20 components), developers often miss mechanisms for organizing all the components of an application and their operations.

PCoC addresses this issue by providing mechanisms for dynamic method dispatch over different components, organizing components in a logical containment relationship, prioritizing them and their operations, automatically exposing operations via an application plugin interface, and defining execution relevant states per operation, e.g. “Enabled”, “Disabled”.

Composition and coupling (connecting for interoperation) of components is configurable, maximizing reuse. To keep the customization effort low, the visual representation of different operations of the same kind, for example, operation `copy` provided by many components, can be defined in a uniform way. A definition in the configuration can be shared by any operations. It can include text strings for menu and toolbar entries, the help string in a status line, an icon, and also associated keyboard shortcuts. The whole configuration of an application can be modified and reloaded at runtime.

Concepts presented in this thesis need not be used as a monolithic package. Some of the concepts and patterns are versatile, meeting requirements also in fields other than IDE development.

1.5 Keywords

Dynamic Coupling, Component Interoperation, Dynamic Component Modification, Dynamic Component Adaption, Menu and Toolbar Setup, Scripting, Application Control, (Priority, Focus) Management, Activity, Delegate, (Method, Request, Message) Dispatching, Dynamic Inheritance, Acquisition, Delegation, Very Late Binding, Meta-Programming, Reflection, Code Reuse, Object-Oriented, Component-Oriented, Application Framework, IDE, Tools, Services.

1.6 Conventions

The following typefaces are used in this thesis:

For source code, mono-space (Courier New)

- Example: `getX`

To emphasize a word or text, or to introduce terminology, italics

- Example: An application may consist of components called *Tools* and *Service Providers*.

Types begin with a capital letter, operation names with a lower case letter.

- Examples: `Point`, `getX`

Classes of the PCoC framework have the prefix “PCC”. This prefix is sometimes omitted for simplicity in diagrams, figures, and text sections where the context is self-explanatory.

- Example: `PCCDispatcher`

1.7 Defining Roles

We distinguish the following roles for developers in this thesis:

1. Framework developer: Responsible for the concepts and the architecture behind an application.
2. Component developer: Develops components without needing to know much about the architecture and how messages are sent between components to perform operations. There is also not much need to know about customizing the representation of operations in the application’s menu or toolbars. The component developer concentrates on developing the functional part of a component.
3. Customizer / Configurator. Assembles components to an application in a reasonable manner and configures, for example, menus and toolbars.

2 Method Dispatch and Delegation

2.1 Overview

This chapter gives a short overview of the Smalltalk, C++/Java, and Oberon method dispatch mechanisms, and some thoughts that lead to the development of new concepts such as a flexible method dispatch mechanism.

We propose a solution to make inheritance relationships and scopes more flexible. The approach is useful for dynamically adding operations to objects or for dynamically changing inheritance relationships such as replacing parent classes, respective parent objects.

The concepts introduced in this section are the basis for the PCoC framework described later in this thesis.

2.2 State of the Art

Two well-known method dispatch facilities are the Smalltalk method dictionaries and the C++/Java virtual tables. Method dispatch is a mechanism to forward method calls to the classes where the methods actually are implemented. It comprises two basic steps: the method look-up in a method dictionary or v-table, and the execution in the corresponding class. The process of finding the class where a method is implemented, and the subsequent invocation on this class is similar to delegation. See later in this section.

The Smalltalk method dictionaries are dictionaries (key/value maps) containing references to methods of a class. Each class has one such method dictionary. When invoking a method, respective when sending a message to invoke it, the corresponding method is searched by name in the class hierarchy of the corresponding object. This method dispatch is done by a component called message handler, respectively dispatcher. This mechanism is very flexible, since all method invocation is done using messages, and methods are dispatched by name; methods can be altered, added or removed at runtime. This flexibility has its price: this mechanism is slower than v-tables (see later in this section).

As optimization, Smalltalk holds a cache of recently sent messages; according to [KRASN84], an appropriate method cache can have hit ratios as high as 95%, reduce method lookup time by a factor of 9, and increase overall method system speed by 37%.

The Smalltalk method dispatch mechanism is described in detail in Section 2.2.1.

More efficient than the Smalltalk method dispatch using method dictionaries are v-tables (virtual method tables)—arrays where the method pointers of a class and its base classes are stored, and the indexes of the method pointers are calculated at compile time. Method lookup in C++ works without any kind of engine or dispatcher. A method call is just a call to the method pointer at the corresponding index in the v-table. The compiler generates code to execute the method. No further look-up has to be done. This mechanism is fast, but not very flexible.

> “Adding or removing a C++ method requires recompilation of potentially many v-tables to preserve their parallel structure. And although we know that polymorphic client code does not need to be rewritten, it must still be recompiled to account for new offsets to the pointers in the tables. This recompilation overhead discourages exploratory programming. On the other hand, C++ cooperates readily with foreign languages, while Smalltalk’s runtime engine gets in the way of calling into or out of the Smalltalk image.” - [LIU96], Pages 183ff (Section 16.2).

A detailed description of v-tables is given in Section 2.2.2.

The method dispatch facilities have one thing in common—delegation. Delegation is a dispatch mechanism originally introduced by Henry Lieberman in [LIEBER86]. As opposed to forwarding, with delegation a method can always refer to the object where the method has originally been invoked, regardless of the number of indirections due to object composition or class inheritance. Delegation is relevant for method calls in object-oriented languages, but also for component interoperation. This mechanism is explained by means of virtual method calls in Section 2.2.4.

Methods are not the only way to handle messages. The Oberon system (Oberon is a descendant of the programming language Modula-2, which itself followed Pascal) supports, besides ordinary methods, so-called message objects. Message objects are data packages. They are subject to be passed to so-called method interpreters (special methods) for processing. A method interpreter may implement functionality for certain message types; it may also ignore message objects of some types, or forward message objects to other message interpreters.

Object-oriented programming with message objects is similar to the way how Smalltalk dispatches method calls; the main difference is that in Smalltalk the dispatcher is integrated in the system, and in Oberon the message interpreter must be implemented per object by the developer. Smalltalk holds method references in method dictionaries; this can be implemented for each message interpreter in Oberon. Results are stored in the message object passed to a method. An overview of Oberon message objects is given in Section 2.2.3. H. Mössenböck describes the Oberon message objects in more detail in [MÖSS95], on Pages 127ff (German edition: [MÖSS98]).

2.2.1 Smalltalk Method Dispatch

Each Smalltalk class has a dictionary (key/value-map) that contains references to the methods of the class. When a method is invoked, it is first searched in the dictionary of the object's class. If it is not found there, it is searched in the base classes until it is found.

To illustrate how the Smalltalk method dispatch works in detail, let us take a look at the following example.

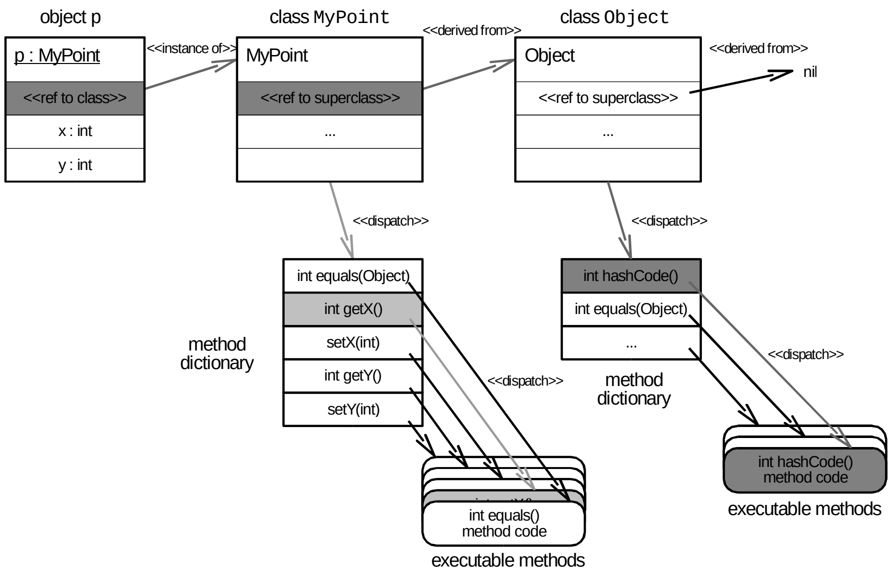


Figure 2.2-1 Method dispatch as in Smalltalk

In the given example we see object `p`, its class `MyPoint`, and the base class (superclass) of `MyPoint`, `Object`. `MyPoint` defines the methods `equals`, `getX`, `setX`, `getY`, and `setY`, i.e. its method dictionary contains references to these methods. `Object` defines, for example, the methods `hashCode` and `equals`.

In Smalltalk, method invocation comprises the following steps:

1. Get the class object
   (`MyPoint`) of the object (`p`) where the method (`getX`) was invoked
2. Get the method dictionary
   The entries in this key/value map are pointers, respectively references, to methods.
3. Look up the method in the method dictionary of the current class
4. If found, invoke the method code (`getX`) and go to step 7
5. If not found then continue with the base class (`Object`) at step 2
6. If there is no base class, then throw an error and go to step 7
7. Done

In our example, the class object is `MyPoint`. When calling `setX`, the method is searched and found directly in the class of `p`, `MyPoint` (light grey path in Figure 2.2-1). In the case of method `hashCode`, no method reference can be found in class `MyPoint`, so the search is continued in the base class `Object`. Finally, `hashCode` is found and executed there (dark grey path in Figure 2.2-1).

2.2.2 V-Tables

V-tables (virtual method tables) are arrays holding method pointers of a class and its base classes. Each method corresponds to an index in a v-table. The indexes are calculated at compile time. When a method is invoked on an object, the method pointer at the corresponding index in the v-table of the object's class is retrieved and the method finally executed.

Let us take a look at the following example to illustrate how v-table method dispatch works.

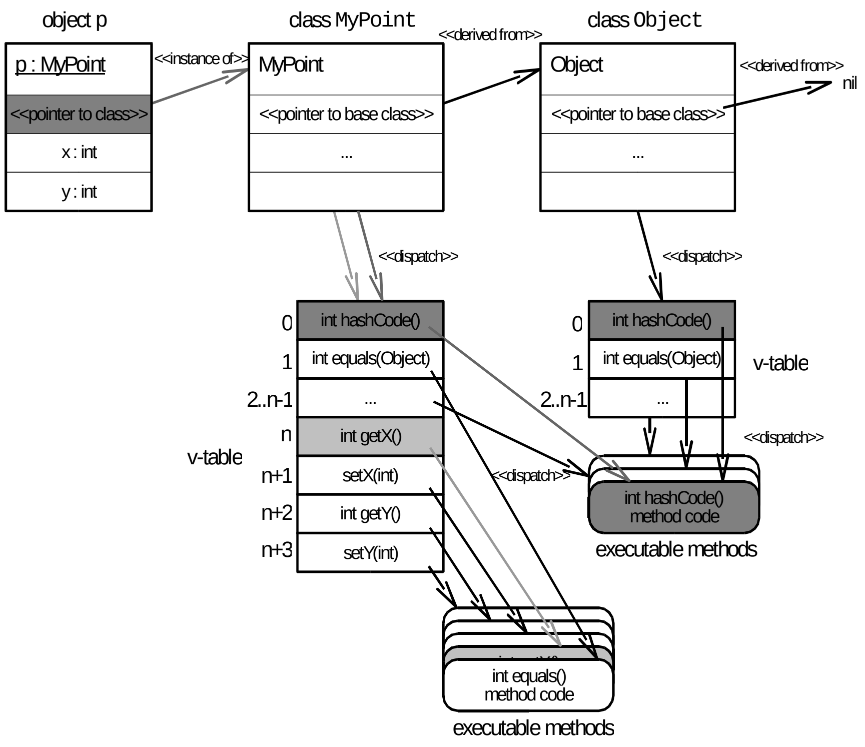


Figure 2.2-2 V-tables

Figure 2.2-2 shows an object `p` and its class `MyPoint` and the base class `Object` including their v-tables. Note that the indexes of method pointers for the v-tables are calculated by the compiler.

A v-table always starts with the v-table-entries of the base class(es), but contains also the pointers to the methods of the actual class. The indexes of the method pointers of base classes are always the same as in the base classes. For example, in Figure 2.2-2 the pointer to method `hashCode` may get the index `0` (calculated by the compiler) in the v-table of `Object`, so `hashCode` has also index `0` in the v-tables of all classes derived from `Object`. So, in Figure 2.2-2 the v-table of `MyPoint` starts with the method pointer to `hashCode`. After the `Object` method pointers we find the pointers to the methods of `MyPoint`, e.g. `getX`, etc.

An equivalent implementation of `MyPoint` in Java can be found in Program 2.2-5.

The method invocation using v-tables comprises the following steps:

1. Get the class object
   (`MyPoint`) of the object (`p`) where the method (`getX`) has been invoked
2. Get the v-table
   The value entries in this array are pointers to methods.
3. Get the method pointer at the index represented by method `getX`. The index was already calculated at compile-time, so you are actually not calling method `getX`, but method `3` (assuming a class structure as in Figure 2.2-2)
4. Invoke the corresponding method, with object `p` as implicit parameter (“this”).
5. Done

Basically, method look-up reduces to retrieving the method pointer at the index for the invoked method in the v-table of the current class. This mechanism is fast, but not very flexible; since v-tables are created at compile-time, there is no way to add or remove methods at runtime. Even if v-table information could be changed at runtime, it would carry some difficulties; if we added or removed a method in a base class, it would be necessary to reorganize the v-table indexes of all derived classes. This would be very time-consuming. With multiple base classes, this is even more difficult—the index of a method must be the same in each base class, derived class, and the current class providing the method.

Besides in object-oriented languages as Java or C++, the v-table concept is also used, for example, in the Microsoft component standard COM and its derivatives (DCOM, etc.). Clemens Szyperski has a review of these component standards in [SZYPE98] on Pages 194ff.

2.2.3 Message Objects

Oberon provides, besides ordinary method dispatch via v-tables, the concept of message objects and interpreters. Message objects are data packages that can be sent to message interpreters (message handlers) in order to be processed. As method of an object, a message interpreter may implement functionality for some message types; it may also ignore messages or forward them to other interpreters.

This concept needs one or many message interpreters responsible for handling incoming message objects. The message handler analyzes the dynamic type of a message and performs a specific operation accordingly.

Let us take a look at some sample code. First, we define some message types:

Program 2.2-1 Message Types

```
TYPE
  Message = RECORD END; (* empty message, base type for all messages *)
  SetXMessage = RECORD (Message) x: INTEGER END; (* a concrete message *)
  GetXMessage = RECORD (Message) x: INTEGER END; (* a concrete message *)
```

As base class of our messages, we define the type `Message`. Concrete types are `SetXMessage` and `GetXMessage`. Both declare a member variable `x`.

To send a message to an object, we write:

Program 2.2-2 Sending messages

```
...
VAR
setXMsg: SetXMessage;
getXMsg: GetXMessage;
result:  INTEGER;

setMsg.x := 10;
obj.Handle(setMsg);
obj.Handle(getMsg);
result := obj.x;
...
```

We create instances of types `SetXMessage` and `GetXMessage`. Furthermore, we initialize `x`, and pass the message object `setXMsg` instance to the message interpreter `Handle` of object `obj` (it can be any object of any type). Then we pass the message object `getXMsg` to `Handle`. Finally, we assign the result of the processed `getXMsg` to `result`.

Program 2.2-3 Interpreting (handling) messages

```
PROCEDURE (VAR m: Message) Handle;
BEGIN
   WITH
      m: SetXMessage DO
         ...  (* do something *)
      |m: GetXMessage DO
         ...  (* do something else *)
      |m: ... (* handle another message *)
   ELSE
      (* ignore other messages *)
   END
END Handle;
```

Program 2.2-3 is a simple dispatch code in Oberon. Each message type has a meaning; it usually stands for an operation. However, the actual behavior (the code that is to be executed) is determined by the dispatch code. The content of a message object is data needed to process the message; in our case the `SetXMessage` carries a value to set the x-position of a point-object, for example, to move a graphical object to that position.

In some cases it may make sense to forward messages to other objects, for example to a child object of a graphical composite-object. Assuming that object `obj` of Program 2.2-2 is such a composite object containing two rectangles, we can add the following statement to the `SetXMessage` section of the WITH-DO-clause of Program 2.2-3:

Program 2.2-4 Sending messages

```
...
PROCEDURE (VAR m: Message) Handle;
BEGIN
   WITH
      m: SetXMessage DO
         childRect.Handle(m);
         childRect2.Handle(m);
      |m: GetXMessage DO
         ...  (* do something else *)
...
```

where `childRect` and `childRect2` also must implement the method `Handle` in order to handle message `SetXMessage`.

This is a powerful way to forward messages, including the possibility to forward them through objects which do not implement all methods associated with the given messages.

Message objects have some advantages over methods ([MÖSS98]):

- Messages are data packages; they can be stored and forwarded later on.
- A message object can be passed to a method that forwards it to different objects. This allows broadcasts, which are not or difficult to realize with methods.
- It is sometimes easier if the sender does not have to care about whether a receiver understands a message (implements a corresponding method) or not.
- It is also possible to access the message interpreter (handler) through a procedure variable, which enables us to replace it by another at runtime.

Message objects also have disadvantages:

- The interface of a class does not reflect which messages can be handled by instances of the class. It is difficult to realize at compile time which objects are related through message forwarding at runtime.
- Messages are interpreted and forwarded at runtime, which is much slower than direct method calls. It depends on how fast dynamic type information is evaluated, or in the case of handling messages using their names, how fast strings are parsed.
- Sending message objects requires more code to be written. Arguments must be packed into the message object. A regular interface such as for methods is not available.
- Invalid messages are not recognized at compile time. Although this provides the flexibility to forward messages through objects that cannot handle them themselves, it can be troublesome to find errors.

Generally one should use methods rather than message objects. However, in some cases it is useful to use message objects (see advantages).

2.2.4 Delegation

Delegation is a dispatch mechanism to share behavior between objects. An object (the receiver) delegates a message to another object (the method holder and delegate) in response to a message. The delegate carries out the request on behalf of the original object, and may send subsequent messages to the original receiver. This includes the invocation of methods. Using delegation, a method can always refer to the object on which it has originally been invoked, regardless of the number of indirections due to object composition or class inheritance.

Basically, there are two types of methods: virtual and non-virtual methods. Virtual methods are dynamically bound methods. In a virtual method invocation, the runtime type of the instance for which the invocation takes place determines the actual method implementation to invoke. In a non-virtual method (statically bound method) invocation, the compile-time type of the instance is the determining factor. The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as overriding the method.

Messages are commonly used to send information from one object or component to another, for example to execute a method on an object. Basically, there are two ways to send messages: forwarding and delegation. The following picture illustrates the difference between these two.

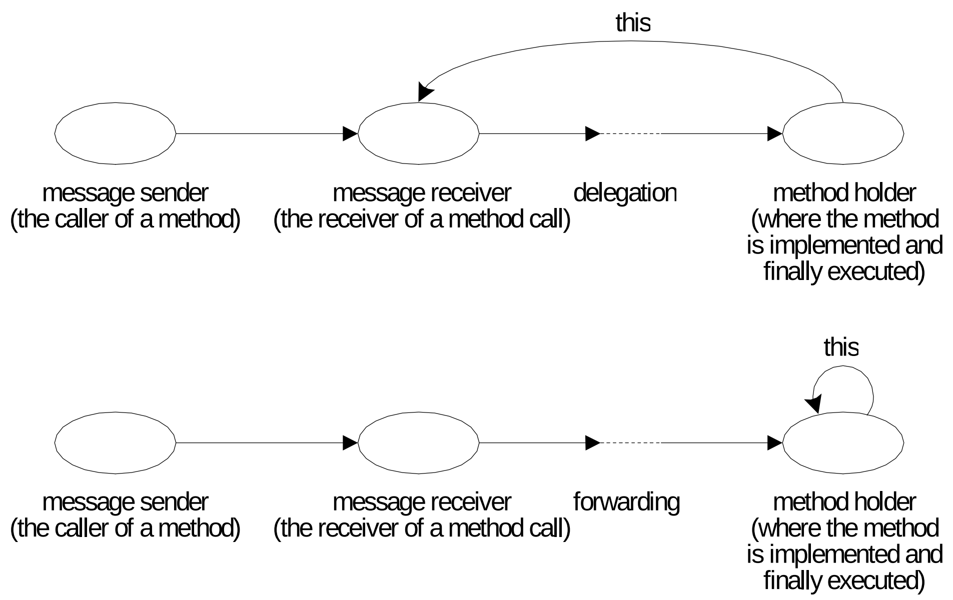


Figure 2.2-3Delegation versus Forwarding

The receiver of a message, or more precisely the object where a method is invoked, may delegate the method call to a base class, if its own class cannot handle it. The class that implements the method, i.e. the class where the method is finally executed, is called method holder. In contrast to simple forwarding, with delegation, messages to “this” or “self” are sent to the original message receiver. Note that many sources confuse delegation with forwarding. For instance, .NET delegates simply forward method calls, although the name may imply another meaning. Some sources describe delegation as message dispatch concept for prototypes (cloning instead of deriving from classes) only. We think, this concept is similar to the method lookup algorithms in class relationships (Smalltalk method dispatch, v-tables).

Let us take a look at the invocation of ordinary virtual methods.

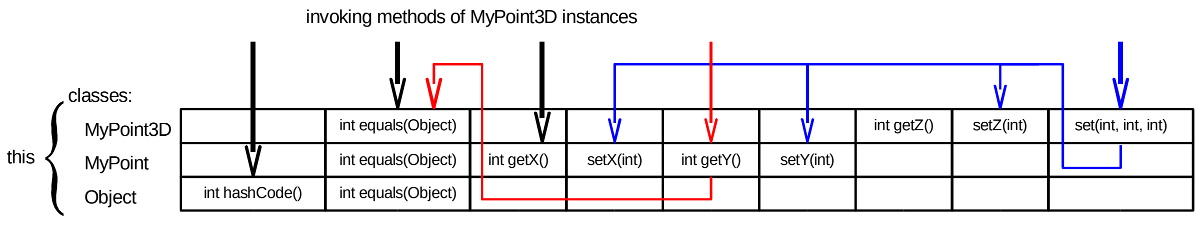


Figure 2.2-4 Virtual method calls

In our example (Figure 2.2-4), we find three classes: `MyPoint3D` (top row), its base class `MyPoint` (middle), and the base class `Object` (bottom). Each instance of class `MyPoint3D` combines all three classes to a unit. In the scope of these classes, the implicit parameter “this” (C++, Java) or “self” (Smalltalk) refers to the current instance (the object where a method has been invoked).

Program 2.2-5 Class MyPoint

```
public class MyPoint extends Object {
   private int x;
   private int y;

   public MyPoint() { x = 0; y = 0; }
   public int equals(Object obj) {
      if (obj != null && obj instanceof MyPoint) {
         MyPoint p2 = (MyPoint)obj;
         if (getX() != p2.getX() || getY() != p2.getY()) { return -1; }
      }
      return super.equals(obj);
   }
   public int getX() { return x; }
   public void setX(int x) { this.x = x; }
   public int getY() { return y; }
   public void setY(int y) { this.y = y; }
}
```

We use Program 2.2-5 as implementation of `MyPoint`; `MyPoint3D` adds the methods `getZ` and `setZ`, and overrides the virtual method `equals`.

Program 2.2-6 Using MyPoint

```
MyPoint p = new MyPoint();
int val = p.getX();
int hc = p.hashCode();
```

In the example above, we call various methods of a `MyPoint` instance.

Program 2.2-7 Class MyPoint3D

```
public class MyPoint3D extends MyPoint {
   private int z;
   public MyPoint() { z = 0; }
   public int equals(Object obj) {
      if (obj != null && obj instanceof MyPoint3D) {
         MyPoint3D p2 = (MyPoint3D)obj;
         if (getZ() != p2.getZ()) { return -1; }
      }
      return super.equals(obj);
   }
   public int getZ() { return z; }
   public void setZ(int z) { this.z = z; }
}
```

In this example, the method `equals` of `MyPoint3D` calls the `equals` method of `MyPoint`, which in turn calls `equals` of `Object`; all calls are executed at the same object (“this”).

The class may be used as follows.

Program 2.2-8 A method call

```
MyPoint3D p = new MyPoint3D();
MyPoint3D p2 = new MyPoint3D();
int result = p.equals(p2);
int resultX = p.getX();
```

Since `p` has the static type (the type at compile-time) `MyPoint3D`. The method `equals` is executed in `MyPoint3D`, no matter, if it is a virtual or non-virtual method, since `MyPoint3D` itself provides it. If `equals` did not exist in `MyPoint3D`, calls to it would be forwarded to the base class (`MyPoint`), and so forth until a base class is found that provides the corresponding implementation. For example, a call to `getX` must be forwarded this way. `getX` is searched in the scope of `MyPoint3D`; since it is not implemented there, the call is forwarded to `MyPoint`, where it is finally executed.

Program 2.2-9 A method call (2)

```
MyPoint p = new MyPoint3D();
MyPoint p2 = new MyPoint3D();
int result = p.equals(p2);
int resultX = p.getX();
```

In Program 2.2-9, we assign `MyPoint3D` instances to variables of type `MyPoint` (in Program 2.2-8, we used variables of type `MyPoint3D`). In this example, `equals` is executed in `MyPoint3D`, if it is a virtual method, and in `MyPoint`, if not. `getX` is executed in `MyPoint` as before.

Generally, for a virtual method the method dispatch always starts the search in the dynamic type of the object, where the method has been invoked; if the method is not found there, it is searched in the base class, and so on, until a class is found which implements the method. In the case of non-virtual methods, the search starts in the scope of the static type of the object (the type at compile-time, `MyPoint`). The same is valid, for example, if `equals` is called in `getY` (implemented in `MyPoint`): if `equals` is a virtual method, it is executed in `MyPoint3D` (see dark grey path in Figure 2.2-4); if not, it is executed in `MyPoint`.

Generally, all method calls to an object and within an object (method calls without explicit specification of the receiver object), are delegated to the same unit “this” or “self”, no matter if the methods are implemented in the object’s class or one of of its base classes. With delegation, as the term is used in prototypical programming languages, `Object`, `MyPoint`, and `MyPoint3D` can be different instances (rather than classes), and we would get the same behavior (see Figure 2.2-4).

Note that delegation is not class inheritance. It is also not (only) forwarding; when a method is invoked on an object, the call may be delegated from the object’s class to a base class which implements the method (in the case that it is not directly implemented in the class of which the object is an instance). In short, delegation is forwarding plus the concept of a common “self”.

2.2.5 Type-Safe Delegation and Dynamic Component Modification

Before we go on with an introduction of our new approach, let us take a look at the following research project. Kniesel G. describes in [KNIES99] a concept for dynamic component modification and delegation through object-based inheritance, and discusses its advantages and issues. The concept was developed in the frame of the research project *Darwin*.

The goal of the project is described on the corresponding Internet pages as follows:

> “The Darwin project aims to improve the foundation of object-oriented systems by bridging the gap between the two families of object-oriented languages known today: class-based and prototype-based ones.”
>
> “The Darwin model describes typed, class-based inheritance extended by static and dynamic object-based inheritance.”

The Darwin model is realized in the programming language *Lava*, an extension of Java. Its type-safe delegation concept and integration into a programming language provides the developer with a high degree of usability through the possibility of static type-checking, while also providing the flexibility of dynamic component modification.

Dynamic component modification includes the customization of components such that they are better suited for specific tasks. Kniesel proposes a classification of known approaches of dynamic component modification according to

- their need for preexisting “hooks” in the application as either suitable for *anticipated* or *unanticipated* changes.
- the time when a modification is made as either *static*, *load-time* or *dynamic* (runtime). We assume that the reader is familiar with the meaning of these terms.
- their ability to adapt whole component types or individual component instances as either *global* or *selective*. Selective approaches can be further classified as either *replacing* or *preserving*, depending on whether they replace an existing component instance by its modified version or let both be used simultaneously.
- the applied techniques as either *code-modification-based*, *wrapper-based* or *meta-level-based*.

The need of unanticipated dynamic component modification has been repeatedly pointed out in literature (e.g. [MAETZ97]). When developing a component, you may not always know which hooks might be needed in the future by users of your component. Users of your component might want to make enhancements to your component (without having the source code), as well as you might want to make modifications to third-party components.

With global approaches, changes are applied to a component type rather than to its instances, so the change affects every single instance of such a component. This may be comfortable, because changes must be applied at one place only, but raises an evolution problem: some clients might still require access to instances of the component using its “original” interface and semantics. You cannot simply break their code. On the other hand, selective approaches also have their shortcomings. For instance, the component to be replaced might not be prepared to hand over its private data to its new version, if the modification was not anticipated. The application might also require simultaneous use of the component’s original and new version.

Code-modification uses two inputs, a class to be modified and a specification of the modifications. For instance, aspect-oriented programming uses this approach. Aspects are “weaved” into functional code, i.e. code is generated from the original source code and aspects; see Section 6.2.

Meta-level-based architectures allow manipulation of a system at runtime. Examples for meta-level-architectures are Smalltalk and the IBM System Object Model (SOM); see Sections 6.8 and 6.11. Kniesel argues in [KNIES99] on Page 3 that meta-level-architectures have two main limitations: they defeat static type system and are inefficient.

When active components neither can be directly modified, nor unloaded from the system, we are faced with the problem to change their behavior. One way to cope with this problem are wrappers. Clients do not directly send messages to the original component, but to wrapper(s). The wrappers may implement the corresponding functionality, or simply forward messages to the original component.

With respect to this classification, the Darwin model presents an approach to unanticipated, dynamic, selective, wrapper-based, object-preserving component modification. Like in Darwin, our flexible method dispatch can be used for dynamic, selective, wrapper-based, object-preserving component modification, when it is applied at instance-level (generic component instance wrappers handle dynamic methods and forward calls to static ones). When integrated at class-level like shown in Program 2.7-1, we use it rather globally and meta-level-based. The concept is the same. The meta-level-approach has one major shortcoming: the programming language or more precisely its reflection facilities have to be adapted. Reflection is the ability to obtain information about the fields, constructors and methods of any class or component at runtime. See [ENGLA97] on Pages 40-42 for a short introduction into Java reflection.

Now back to delegation. In contrast to code modification, dynamic component modification based on delegation does not require the source code of the components to be adapted. It can be deployed at runtime.

Many “simulations” of delegation have been proposed, either as language-specific idioms [COPLI91] or general patterns [GAMMA95]. Simulation techniques and their drawbacks are discussed and summarized in [HARRIS97] and [KNIES98].

The main disadvantages of the simulations are:

- the need to anticipate the use of a piece of software as part of a larger composite and to provide “hooks” that allow the correct treatment of “this” in the context of the composite.
- the need to obey rigid coding convention to implement the hooks.
- the need to edit or at least recompile the wrapper class (the delegating class) when the interface of the original class (delegated-to class) changes.

According to [KNIES99], each of the simulation techniques has additional shortcomings in terms of limited applicability, limited functionality, limited reusability and excessive costs:

- Storing a reference to “this” or “self” in parent objects (or base classes) has limited applicability. Sharing of one parent by multiple delegating children cannot be expressed at all and recursive delegation can only be simulated with significant runtime and software maintenance costs.
- Passing a reference to “this” as an argument of forwarded messages requires to extend the interface of methods in parent objects, which might not be possible, if the parent object is part of a ready-made, black-box component (a third-party component). Furthermore, the typing of the explicit “this” argument interacts in subtle ways with the construction of subclasses of parent classes. In the end, the simulation either does not reach full functionality of delegation or it does so at the price of excessive costs for managing class hierarchy changes, rendering reuse ad absurdum.

In Darwin / Lava, objects can delegate to others referenced by their delegation attributes. If a class `C` declares a delegation attribute of type `P`, we say that `C` is a *declared child class* of type `P` (and of its subtypes), and `P` is a *declared parent class* of `C`. A class can use the methods of its declared parent classes as if they were declared in the same class, or in a base class.

Delegation attributes are declared in an object’s class by adding the keyword `delegatee` to an instance variable declaration. There are two types of delegation attributes: *mandatory* and *optional*. An attribute is called mandatory if it must have a non-null value; optional otherwise.

As example we use a class `Shape`. A `Shape` represents a visual object and holds the necessary data structure for the object. Concrete examples are rectangles, polygons, images. `Painter`s are responsible for actually drawing `Shape` objects on a canvas in a graphical user-interface.

In this code fragment we declare `Painter` to be a declared parent class of class `Shape`. As painter we instantiate a `BorderPainter` which paints a `Shape` object and its enclosing border.

Program 2.2-10 Declared parent

```
public class Shape {
   // the delegatee "painter" must not be null
   mandatory delegatee Painter painter;
   public Shape() {
      painter = new BorderPainter();  // draws the object and its border
   }
   // switch paint strategy
   void setPainter(Painter painter) { this.painter = painter; }
   Point getPos() {...}
}
```

The following picture illustrates how the classes are related to each other. We use the extended UML-notation "delegates to".

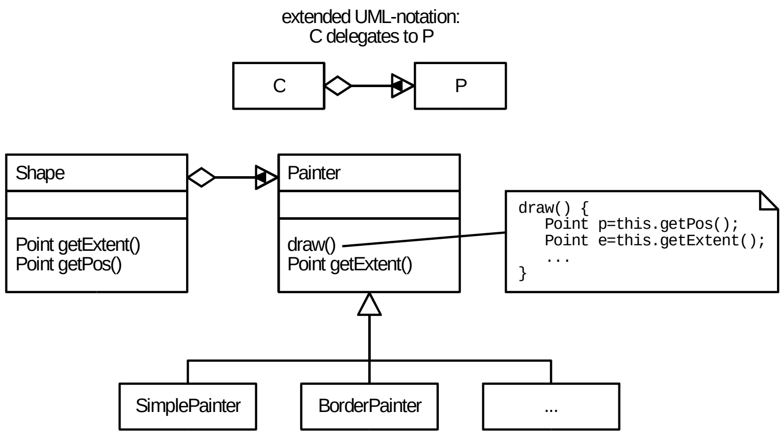


Figure 2.2-5 Declared parent and child

Since `Shape` is a declared child class of `Painter` and its derived classes, the methods of `BorderPainter` can be used as if they were declared in `Shape` or one of its base classes.

In the following case, we call the method `draw` on `c`, an instance of class `Shape`. The method call will be delegated to `p`, an instance of class `BorderPainter`; `draw` in turn calls `getPos` which is implemented in class `Shape`, so the method call is delegated back to `c`. In short, `c` and `p` are united by a common "this".

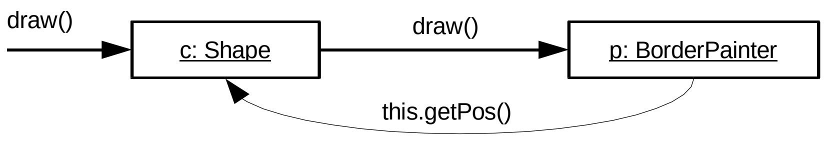


Figure 2.2-6 Delegation to declared parent

Note that if `BorderPainter` introduces methods that are neither declared in `Painter` (the declared parent of `Shape`), nor in a base class `Shape`, e.g. an own `getPos`-method, then calls to such methods are not delegated back to `Shape` instances (one would expect that a child always overrides methods of the parent). This is a special behavior of the Darwin model and is described in [KNIES99] on Page 11.

The Darwin model offers a certain level of flexibility while providing good performance and type-safety. On the other hand, a shortcoming is the necessity to statically declare parents. Possible future component enhancements must be anticipated; hooks like in Program 2.2-10 (`mandatory delegatee Painter`) are necessary.

Read more about the Darwin / Lava delegation mechanism and a summary of dynamic component modification features and issues in [KNIES99] and the corresponding Internet pages.

2.3 Flexible Method Dispatch

The method look-up and invocation (also called method dispatch) can be optimized so that methods of base classes need not be searched in the class hierarchy as necessary in Smalltalk method dictionaries, while keeping method dispatch dynamic like in Smalltalk and unlike v-tables.

Based on this idea we propose a flexible message dispatch mechanism and show examples for the usage. The mechanism has been designed to be deployed on top of the reflection facilities of common programming languages. We understand reflection as the ability to obtain information about the fields, constructors and methods of any class or component.

Before we take a closer look at the dynamic capabilities of the flexible method dispatch, let us see how the mechanism basically works and how it is realized in contrast to the mechanisms described in the previous sections.

Figure 2.3-1 depicts the same object `p` and its classes as Figure 2.2-1, with one difference: method references of a base class (e.g. `Object`) are also in the method dictionaries of derived classes (`MyPoint`). Note that in Smalltalk each class has a method dictionary with direct references to its methods. Methods of base classes are not stored in method dictionaries of derived classes. In C++, v-table (virtual method tables) entries are direct pointers to the methods of the class or its (direct or indirect) base classes. With the optimization shown in Figure 2.3-1, a value in a method dictionary is a structure containing a direct reference to a method (`getX`) of the same class and a list of indirect references to dictionary entries (`hashCode`) of other classes (`Object`). The key is a method name. We call such a structure a method reference, respectively a Dispatcher.

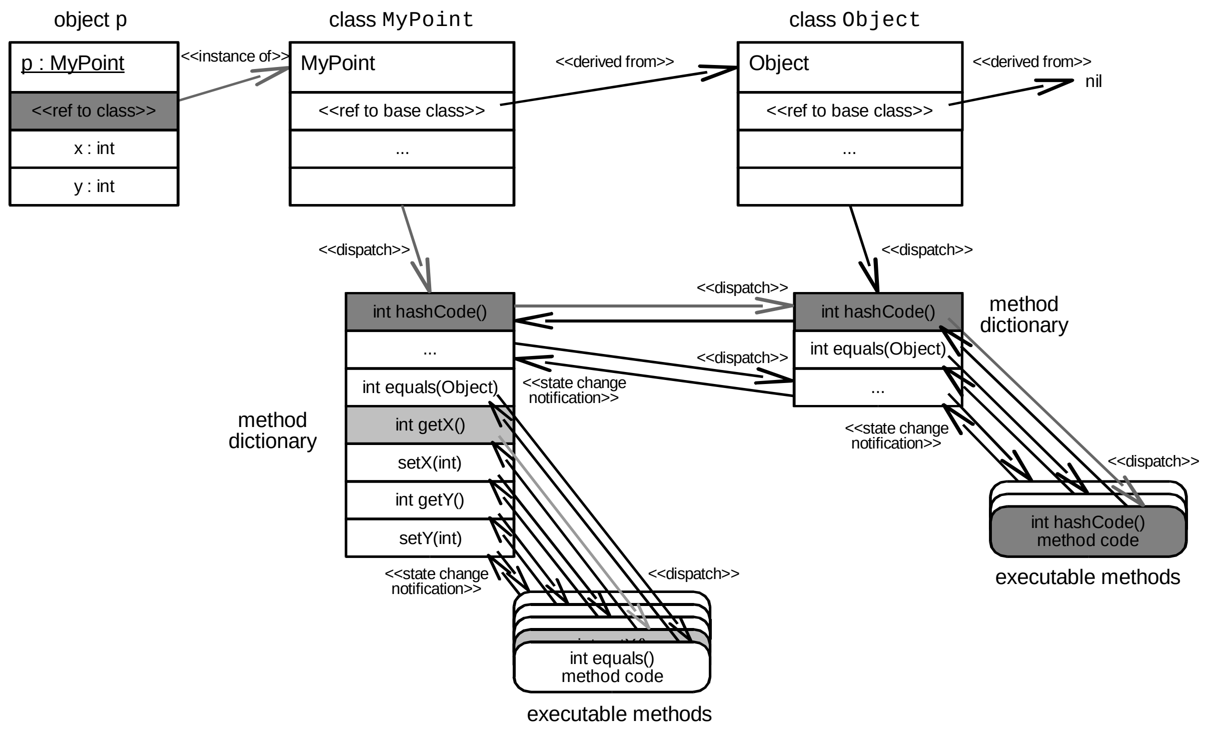


Figure 2.3-1 Method dispatch using indirect method references

This approach allows us to replace base classes and methods by others at runtime. In contrast to v-tables no indexes must be recalculated in this case. Method references are notified whenever the state (“Enabled”, “Disabled”, etc.) of an associated method changes. For such a notification, the method references (for example, `hashCode` in `MyPoint`) need not be looked up, since they are listeners of either the method object itself (method `hashCode` in `Object`), or an other associated method reference (method reference `hashCode` in `Object`). Furthermore, the approach allows us to attach listeners also to method references (not only to methods), which are notified with each method invocation or state change. Note that method states are a feature introduced with this method dispatch mechanism. See also Section 4.3.7.

We can also use this mechanism for multicasts (broadcasts) by delegating method calls to a set of Dispatchers (equivalent to .NET delegates). Like .NET delegates, our Dispatchers can reference method objects of different classes and objects.

When a method is invoked, the method is searched by simply following the Dispatchers (`hashCode` in class `MyPoint`) to a real method (method `hashCode` of class `Object`).

The flexible method invocation comprises the following steps. Compare to the Smalltalk method dispatch algorithm in Section 2.2.1 (Pages 8f) and the v-table-based dispatch in Section 2.2.2 (Pages 9f):

1. Get the class object
   (`MyPoint`) of the object (`p`) where the method (`getX`) was invoked
2. Get the method dictionary
   a dictionary (key/value map) where the entries are references to methods and the keys are the corresponding method names.
3. Look up the method reference (Dispatcher)
   for `getX` in the Dispatcher dictionary of the current class
4. If found, then resolve the method reference
   a) If the method reference is a real method, then invoke the method code (`getX`) and go to step 6
   b) If the method reference is an indirect reference to a method, then dereference it (take the value which it references) and continue with step 4a
5. If not found then throw an error (the method does not exist) and go to step 6
6. Done

2.4 Method Dispatch Using Reflection

In the following illustrations, code fragments and explanations are based on the reflection facilities of Java. One reason why we choose a reflection-based approach rather than a direct integration of the flexible method dispatch mechanism into the Java programming language is that we want to avoid confusion with code-fragments of the Java programming language (what is part of the Java development kit and what is new?). Another reason is that this mechanism was mainly designed for dynamic object-based use. More about that later.

To avoid confusion with the well-known class concept and its implementations, we introduce the term *Activity Set* for sets of operations (see also Section 1.2). The concept is similar to the class concept. Like a class, an Activity Set is a named scope and contains a dictionary (key/value map) with references to operations. In our case, operations are called Activities, and reference objects to Activities are called Dispatchers. Activity Sets can be nested (so they have a path) and can acquire from (delegate to) others. The main difference to classes is that Activity Sets are based on objects rather than on static type-information. Their Activities can be defined in different objects. An Activity Set treats its Activities, respectively their associated objects, as unit, so it supports delegation to independent objects. But more about that later.

In the following, we do not use the terms “base class” or “derived class” together with the flexible method dispatch, but rather the terms “parent and “child”. Note that an Activity Set may directly be related to a class.

To reflect that we choose a dynamic object-based approach rather than class inheritance, we slightly adapt Figure 2.3-1.

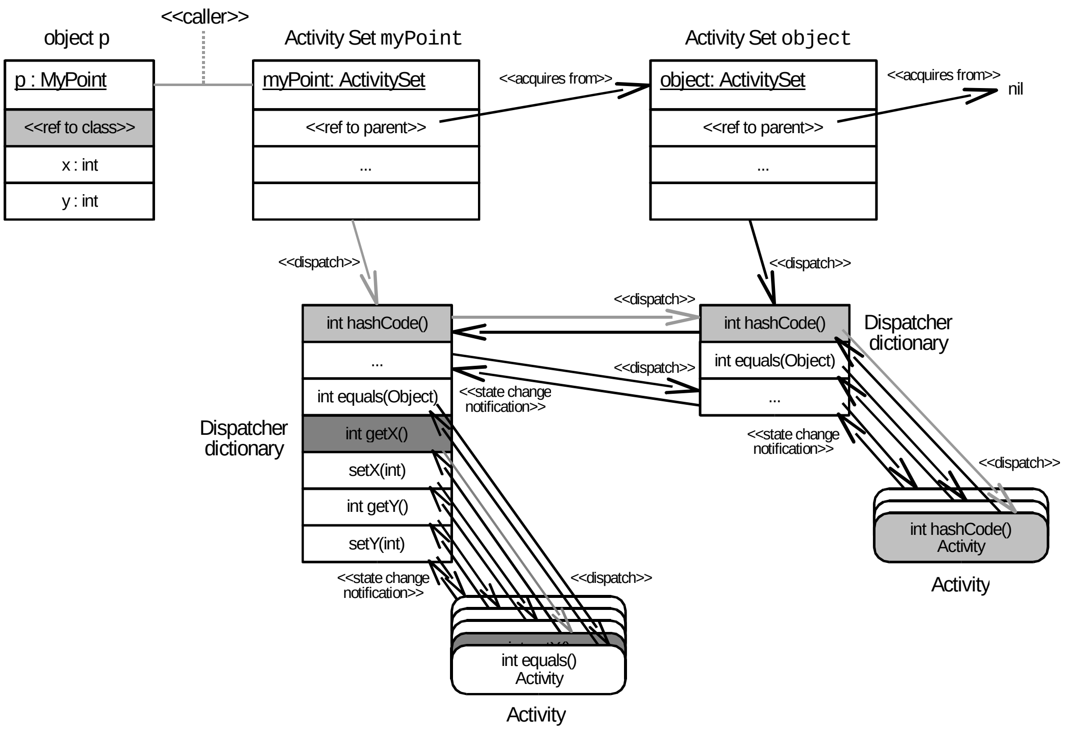


Figure 2.4-1 Method dispatch using Dispatchers and Activity Sets

There is not much difference to Figure 2.3-1—we use the classes `MyPoint` and `Object`, or more precisely their Activity Set equivalents, for this example. We assume that the class `MyPoint` is not statically derived from the class `Object` anymore. The corresponding Activity Set `myPoint` is a child of `object` and “acquires” from it, that is it dynamically inherits from `object`. So we get a dynamic relationship between these classes instead of the static one of Figure 2.3-1.

The caller of an Activity (a “dynamic” method) makes an association between the involved object and an Activity Set. More precisely, the Activity Set is initialized with the methods of the object’s class; the Activity Set can then be used to dynamically invoke methods.

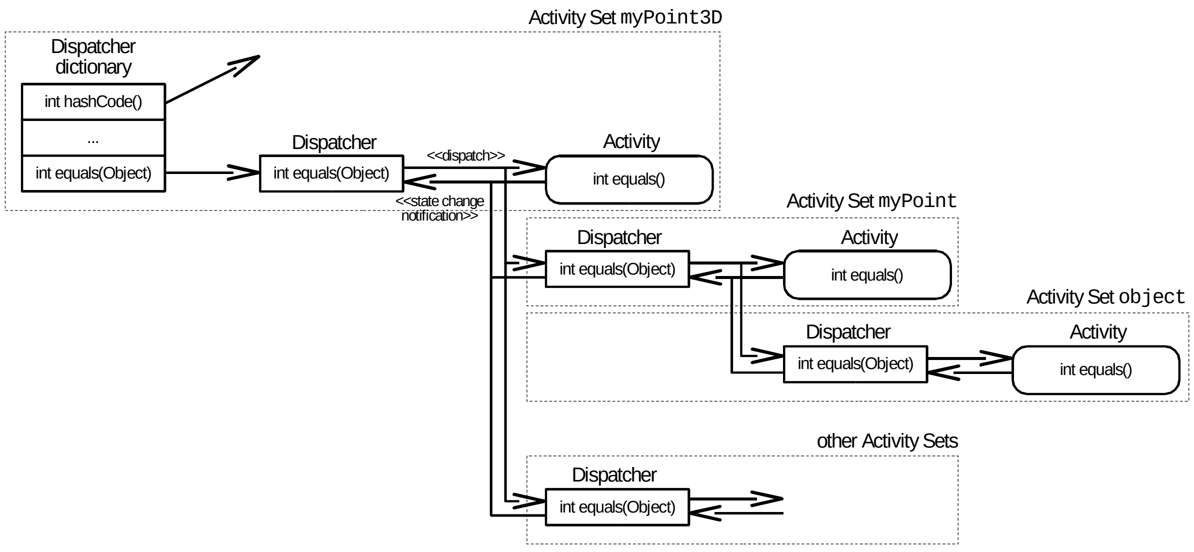


Figure 2.4-2 Dispatcher chain

Figure 2.4-2 illustrates how Dispatchers and Activities are related to each other. An entry in a Dispatcher dictionary—here that of Activity Set `myPoint3D`—is a reference to a Dispatcher in the same Activity Set. Note that the Dispatcher dictionaries of the other Activity Sets are not shown here. The Dispatcher may have a reference to an Activity of the same Activity Set and/or references to Dispatchers of other Activity Sets (the parent Activity Sets in a delegation relationship). The parent of `myPoint3D` is `myPoint`, and that of `myPoint` is `object`.

In the following we see the flexible method dispatch algorithm as pseudo code.

Program 2.4-1 Method look-up using (indirect) Dispatchers (pseudo code)

```
Object perform(ActivitySet a, Object p, String activityName, Object[] arguments) {
   Dispatcher m = a.getDispatcher(activityName);    // get a dispatcher
   if (m != null) {
      while (!m.isActivity()) {
[1]      m = m.getDispatcher();                     // get dispatcher target
      }
[2]   return m.getActivity().perform(p, arguments); // invoke the retrieved
   }                                                // activity
   else {
      ...  // throw an error (activity not found)
   }
}
```

An Activity Set is passed to the `perform`-method. It contains Dispatchers (method references) to the object `p`. Program 2.4-3 describes how an Activity Set can be generated from an object. We assume that the class `ActivitySet` provides a method `getDispatcher` which returns a (direct or indirect) reference to the Activity associated with the given Activity name (`activityName`); the implementation of the class is shown later. Actually a Activity signature, i.e. the Activity name including parameter and result types, must be specified for `getDispatcher`, but for the simplicity of the example we ignore that for now. If the value returned by `getDispatcher` is not null, we call `getDispatcher` until we get the actual Activity [1]; otherwise we throw an error (Activity not found). Finally we execute the Activity (`getActivity().perform()`) [2].

Note that the code fragment above is pseudo code and may not compile. However, there is a conceptually equal implementation in the PCoC framework. The term *Activity* is used for method objects in the context of PCoC; their *Activity Sets* are dynamic scopes where Dispatchers of different objects can be added to; method reference objects containing the forwarding logic are called *Dispatchers*. See Section 5.6. A more detailed description of Activity Sets is given later.

Program 2.4-2 illustrates a simple implementation of an Activity. When `perform` is called, the argument types are checked and finally the template method `doPerform` is called. As containers for arguments and return values we use instances of class `Material`. The implementation of class `Material` is trivial.

See Program 4.3-5 for the corresponding implementation of an Activity in PCoC.

Program 2.4-2 A simple Activity implementation

```
class Activity {
   Method performMethod;
   String name;
   Object provider;
   Activity(Method m) {
      name = m.getName();
      performMethod = m;
      provider = null;    // is set when added to an Activity Set
   }
   public Material perform(Object o, Material args) {
      if (checkArguments(args)) {    // do a dynamic type check
         return doPerform(o, args);
      }
      return null;
   }
   protected Material doPerform(Object o, Material args) {
      Object result = null;
      try { result = performMethod.invoke(o, args.toArray());
      } catch ... // NoSuchMethodException, IllegalAccessException,...
      return new Material(result);
   }
   ...
}
```

As example we dynamically invoke the method `hashCode` of an instance of the class `MyPoint` using an Activity Set. We assume that the method is exposed as Activity by the parent Activity Set `object`, that is it is implemented in the associated class `Object`, and not by `myPoint`:

Program 2.4-3 Invoking a method using Dispatchers (pseudo code)

```
...
MyPoint p = new MyPoint();
...
// use the methods of the object's class to initialize an Activity Set
ActivitySet myPoint = PCCRegistry.getOrCreateActivitySet(p.getClass());
ActivitySet object = PCCRegistry.getOrCreateActivitySet(Object.class);
myPoint.acquire(object);
Object[] arguments = new Object[] {};  // no arguments
activityName = "hashCode";
Object result = perform(myPoint, p, activityName, arguments);
```

First, we retrieve the class of object `p` (where the specified method will be invoked) and use it for the initialization of a new `ActivitySet` instance.

Note that such an Activity Set is generally created only once and reused many times. For that reason, we create the Activity Set using the Activity Set registry (`PCCRegistry`). The registry is a singleton and assures that there is only one Activity Set per class, respectively class name, in the system. As already mentioned, we decided not to integrate the flexible method dispatch approach into a programming language in order to be able to use conventional compilers. On the other hand, if we integrated it into a programming language, classes would be capable of generating and delivering corresponding Activity Sets. However, it may not make sense that each class (or object) has an own Activity Set, and there may be Activity Sets which are not associated with only a single class or object. An Activity Set may have method references (Dispatchers) to several independent classes and/or objects.

The second Activity Set in our example corresponds to the methods of the class `Object`, and is acquired by the `myPoint` Activity Set. The Activity Sets stay related to each other, unless we call `discard` (the opposite of `acquire`).

We use the method `perform` of Program 2.4-1 to invoke `hashCode` on the object `p`. `myPoint` does not directly offer the method, therefore it has an indirect Dispatcher pointing to the Dispatcher of `hashCode` in Activity Set `object`. The Activity and the associated method are finally found and executed in `object`, respectively in the class `Object`. The reasons why we use dynamic method invocation, i.e. method invocation based on reflection, rather than static method invocation become clear when we go into detail.

But first, let us take another look at the method dispatch algorithm of Smalltalk and compare it to the flexible method dispatch. We use Java-like pseudo code in order to make the comparison easier:

Program 2.4-4 Method look-up algorithm of Smalltalk (pseudo code)

```
Object perform(Class c, Object p, String methodName, Object[] arguments) {
   Method m;
   Class pc = c;
   while (pc != null) {
      m = pc.getMethod(methodName); // get the method, if defined in the
                                    // current class
      if (m != null) {
[2]      return m.invoke(p, arguments);  // invoke the retrieved method
      }
[1]   pc=pc.getSuperclass();        // provided that there is only one base class
   }
}
```

The main difference to the flexible method dispatch mechanism is that the requested method is retrieved by searching it in the class hierarchy (see [1] in Program 2.4-4); in contrast to that the flexible method dispatch retrieves a method by resolving indirect Dispatchers (see [1] in Program 2.4-1). More precisely, in the worst case Smalltalk does a lookup in the method dictionaries of the class and each base class of the given object, whereas the flexible method dispatch does only one lookup in the Dispatcher dictionary of the object's Activity Set, respectively class, and traverses the hierarchy by resolving (dereferencing) Dispatcher references.

Note that some Smalltalk systems may have slightly different implementations of this look-up algorithm.

2.5 Dispatcher-Dictionary Implementation

A Dispatcher dictionary for Figure 2.4-1 could look like Program 2.5-1 (simplified).

In this Dispatcher dictionary class we define methods for adding and removing methods. When an Activity or Dispatcher is added, a new Dispatcher is created as wrapper of the original one and stored in the dictionary. Take a look at Program 2.4-1 to see how Dispatchers are resolved.

Note that, for example, in Java, a `Method` object (as needed for `addActivity`) of a class `c` can be retrieved by calling `c.getMethod("<methodName>", Class[] parameterTypes)`. `asInterfaceString`, a method of `PCCActivityInterface`, generates an interface specification string for the parameter type list of the given method. See Sections 4.3.4 and 5.5.1 for descriptions of the dynamic Activity type concept and the `PCCActivityInterface` class.

The implementations of `asInterfaceString` and methods of `DispatcherDictionary` are omitted here, since they are not relevant to explain the concept.

Program 2.5-1 Dispatcher dictionary (pseudo code)

```
public class DispatcherDictionary {
   private HashMap dispDict;

   public DispatcherDictionary() {
   }
   public void addActivity(Method m) {  // add method and wrap it by a dispatcher
      HashMap dsps = (HashMap)dispDict.get(m.getName());
      if (dsps == null) {
         dsps = new HashMap();                  // create a hash map for methods
         dispDict.put(m.getName(), dsps);       // with the same name
      }
      // build a signature from the parameter types and use it as key for
      // the hash map
      String signature = PCCActivityInterface.asInterfaceString(
                            m.getParameterTypes());
      dsps.put(signature, new Dispatcher(new Activity(m)));
   }
   public void addDispatcher(Dispatcher d) {    // add Dispatcher and wrap
                                                // it by another Dispatcher
      ...  // the same as above, except last line
      dsps.put(signature, new Dispatcher(d));
   }
   public Dispatcher removeDispatcher(String dispName, Class[] parameterTypes) {
      HashMap dsps = (HashMap)dispDict.get(dispName);     // get method hash map
      String signature = PCCActivityInterface.asInterfaceString(
                            parameterTypes);              // and remove the
      if (dsps != null) {                                 // Dispatcher
         Dispatcher m = (Dispatcher)dsps.remove(signature);
         if (dsps.size() == 0) {
            dispDict.remove(dispName);
         }
         return m;
      }
      return null;
   }
   public Dispatcher getDispatcher(
            String dispName, Class[] parameterTypes) {
      HashMap dsps = (HashMap)dispDict.get(dispName);    // get method ref
      String signature = PCCActivityInterface.asInterfaceString(
                            parameterTypes);             // from the hash map
      if (dsps != null) {
         return (Dispatcher)dsps.get(signature);
      }
      return null;
   }
   public Dispatcher[] getDispatchers(                   // get all Dispatchers
            String dispName) {                           // with name dispName
      HashMap dsps = (HashMap)dispDict.get(dispName);
      if (dsps != null) {
         return dsps.values().toArray();   // returns a set of Dispatchers
      }
      return null;
   }
   public Dispatcher[] getDispatchers() {  // get all Dispatchers
      ...  // returns a set of Dispatchers
   }
   public String makeSignature(Class[] parameterTypes) {
      ...  // concatenate class names
   }
   ...
}
```

2.6 Dispatcher Implementation

For the following code fragments we assume that methods are always represented by instances of the class `Activity` and that they are subject to be explicitly used by the developer (not only implicitly by the reflection system of the programming language).

The corresponding Dispatcher class could look like Program 2.6-1.

Program 2.6-1 Dispatcher implementation (pseudo code)

```
public class Dispatcher {
   private Activity a;
   private Dispatcher dsp;  // for simplicity of this example only one acquired
                            // dispatcher; normally this is an object array
                            // (for multicasts)

   public Dispatcher(Dispatcher dsp) {
      this.dsp = dsp;
      a = null;
   }
   public Dispatcher(Activity a) {
      this.a = a;
      dsp = null;
   }
   public boolean isActivity() { return dsp == null; }
   public Dispatcher getDispatcher() { return dsp; }
   public void addDispatcher(Dispatcher dsp) { this.dsp = dsp; }

   public Activity getActivity() { return a; }
   public void setActivity(Activity a) { this.a = a; }

   public String getName() {
      if (a != null) {
         return a.getName();
      } else if (dsp != null) {
         return dsp.getName();
      }
      return "";
   }
   public setName(String name) {
      if (a != null) {
         return a.setName(name);
      } else if (dsp != null) {
         return dsp.setName(name);
      }
   }
   Object perform(Object p, Object[] arguments) {
      if (isActivity()) {
         return getActivity().perform(p, arguments);// invoke retrieved Activity
      } else {
         return getDispatcher().perform(p, arguments);  // get Dispatcher target
      }
   }
   ...
}
```

The class `Dispatcher` supports direct references to Activities and indirect ones (references to other Dispatchers). Dispatchers can be linked in chains (`new Dispatcher(dsp)`), whereas the ends of such chains are always direct references to Activities.

Note that we have two member variables—a reference to an Activity and one to another Dispatcher. For multiple delegation parents you may introduce an array of Dispatchers and extend or shrink it on demand.

`Dispatcher` provides a method `perform` which corresponds to the method dispatch algorithm of Program 2.4-1, but as recursive implementation. The Dispatcher for a given Activity name and object `p` is retrieved somewhere else:

Program 2.6-2 Retrieving a Dispatcher (pseudo code)

```
Class pCl = p.getClass();   // or MyPoint.class
Class[] parameterTypes = new Class[] {Object.class};
Object[] arguments = new Object[] {};
ActivitySet myPoint = PCCRegistry.getOrCreateActivitySet(pCl);
Dispatcher dsp = myPoint.getDispatcher("getX", parameterTypes);
Integer result = (Integer)dsp.perform(p, arguments);
```

When the caller does not need (or want) to keep Dispatchers for later use, there is a simpler way for invoking Activities, respectively their associated methods:

Program 2.6-3 Dynamically invoking a method (pseudo code)

```
ActivitySet myPoint = PCCRegistry.getOrCreateActivitySet(p);
Integer result = (Integer)myPoint.perform(p, "getX", arguments);
```

This assumes that the class `ActivitySet` provides a few convenience methods. For instance, there is a method that allows us to specify an object instead of a class when creating an Activity Set. The method `perform` of class `ActivitySet` may automatically determine the signature of the requested method. It can do so by looking for a Dispatcher in the Dispatcher dictionary, whose signature matches the given method name and the types of the given arguments. If there are ambiguities, an error could be thrown. Note that such type-checks are performed at runtime, therefore errors can also be reported only at runtime.

The Activity Set is normally reused many times, so it may be stored as member variable in a class.

2.7 Using Dispatcher Dictionaries and Indirect References

Now, let us take a look at the implementation of the class `ActivitySet` and how it collaborates with its Dispatcher dictionary:

Program 2.7-1 Class `ActivitySet`

```
public final class ActivitySet {
   private DispatcherDictionary dd;
   private ActivitySet parent;

   ActivitySet(Class cl) {   // can only be created by the Activity Set registry
      md = new DispatcherDictionary();
      if (cl != null) {
         int i;
         Method[] methods = cl.getMethods();
         for (i=0; i<methods.length; i++) {  // add all methods of the given class
            addActivity(methods[i]);
         }
         parent = PCCRegistry.getOrCreateActivitySet(cl.getSuperclass());
         acquire(parent);
      }
      ...
   }
   ...
```

The Activity Set constructors take either a class as argument, an object, or a name (which may correspond to a class name). In all cases the methods of the given class are exposed as Activities and added to the Activity Set at creation time. If there is a base class, a corresponding parent Activity Set is created and acquired, that is its Dispatchers are wrapped and added to this Activity Set (iterate over all Dispatchers of the parent and add it by using `addDispatcher`).

Program 2.7-2 Class `ActivitySet` (2)

```
   ...
   ActivitySet(Object o) {    // can only be created by the Activity Set registry
      this(o.getClass());     // add all methods of the object’s class
   }
   ActivitySet(String name) { // can only be created by the Activity Set registry
      this(Class.forName(name));  // add all methods of the class
   }
   ...
   public void addActivity(Method m) {  // creates an Activity from the given
      dd.addActivity(m);                // method object
   }
   public void addDispatcher(Dispatcher dsp) {
      dd.addDispatcher(dsp);
   }
   public void removeDispatcher(String dispName, Class[] parameterTypes) {
      dd.removeDispatcher(dispName, parameterTypes);
   }
   public Dispatcher getDispatcher(
            String dispName, Class[] parameterTypes) {
      return dd.getDispatcher(dispName, parameterTypes);
   }
   public Method[] getDispatchers() {
      return dd.getDispatchers();
   }
   ...
```

We use the method `getMethods` of the Java reflection package to get all methods of the given class. Generally, Activities and Dispatchers can be added or removed at any time. For the retrieval of a single or all Dispatchers we can use the methods `getDispatcher`, respectively `getDispatchers`.

Note that we use the Dispatcher dictionary class of Program 2.5-1 and the Dispatcher class of Program 2.6-1.

Program 2.7-3 Implementation of `perform` in class `ActivitySet` (pseudo code)

```
   public Object perform(Object p, String dispName, Object[] arguments) {
      int i=0;
      // get all Dispatchers with the given name
      Dispatcher[] dsps = dd.getDispatchers(dispName);
      while (i < dsps.length) {          // iterate over all Dispatchers
         Class[] parameterTypes = dsps[i].getParameterTypes();
         if (parameterTypes.length() == arguments.length) {
            int j=0;                     // search for a Dispatchers with a
            while (j<arguments.length) { // matching interface; the parameter
                                         // types must match the types of the
                                         // given arguments (actual parameters)
               if (!parameterTypes[j].isInstance(arguments[j])) { break; }
               j++;
            }
            // found a matching Dispatcher; invoke it
            if (j >= arguments.length) { return dsps[i].perform(p, arguments); }
         }
         i++;
      }
      ... // throw error
      return null;
   }
   ...
}
```

The implementation of the `perform` method is shown above. It retrieves a Dispatcher with a matching interface from the Activity Set’s Dispatcher dictionary.

The Dispatcher dictionary `dd` returns an array of Dispatchers which match the given Dispatcher name. The array is searched for a Dispatcher whose parameter types match the types of the arguments (actual parameters) passed to this method (`perform`). If there is a matching Dispatcher, it is invoked, and thus its associated Activity; otherwise an error is thrown.

Now let us make use of the dynamic method capabilities of the flexible method dispatch:

Program 2.7-4 Adding indirect Dispatchers to a class (pseudo code)

```
ActivitySet myPoint = PCCRegistry.getOrCreateActivitySet(MyPoint.class);
ActivitySet object = PCCRegistry.getOrCreateActivitySet(Object.class);
myPoint.acquire(object);
Class[] parameterTypes = new Class[] {Integer.class, Integer.class};
myPoint.addActivity(                                       // add new method
   "setXY"
   new Activity("setXY", parameterTypes) {
      public Object doPerform(Object obj, Object[] args) { // executed when the
         setX((Integer)args[0]);                           // Activity is invoked
         setY((Integer)args[1]);
         return null;  // the actual functionality
      }
   ...
});
```

First, we create an Activity Set from the class `MyPoint`. Remember, an Activity Set adds Activities and Dispatchers for all methods of the specified class to its Dispatcher dictionary at creation time. In the given example all `MyPoint` methods are added as Activities and Dispatchers to the Activity Set `myPoint`. After that, we explicitly add an Activity (method object) `setXY`. The Activity may contain functional code in its (static) method `doPerform`, or simply forward calls to the corresponding methods of the object `obj`. In the example above, the functional code is directly implemented in the Activity's method `doPerform`.

The implementation of `addMethod` can be found in Program 2.5-1.

2.8 Dynamically Adding Aspects to Methods

We may want to wrap existing methods in order to add aspects. An aspect is specific code to validate actual parameters, to trace calls, etc. The following example should illustrate why flexible method dictionaries (Dispatcher dictionaries) are useful for this purpose. We add aspects before and after the original method `equals`.

For illustration of this example, we use Figure 2.3-1 and adapt it accordingly. See Figure 2.8-1. `eq_aspect_in` and `eq_aspect_out` are the aspects we want to add. They are implemented as Activities which should be called before and after the original Activity `equals` is invoked.

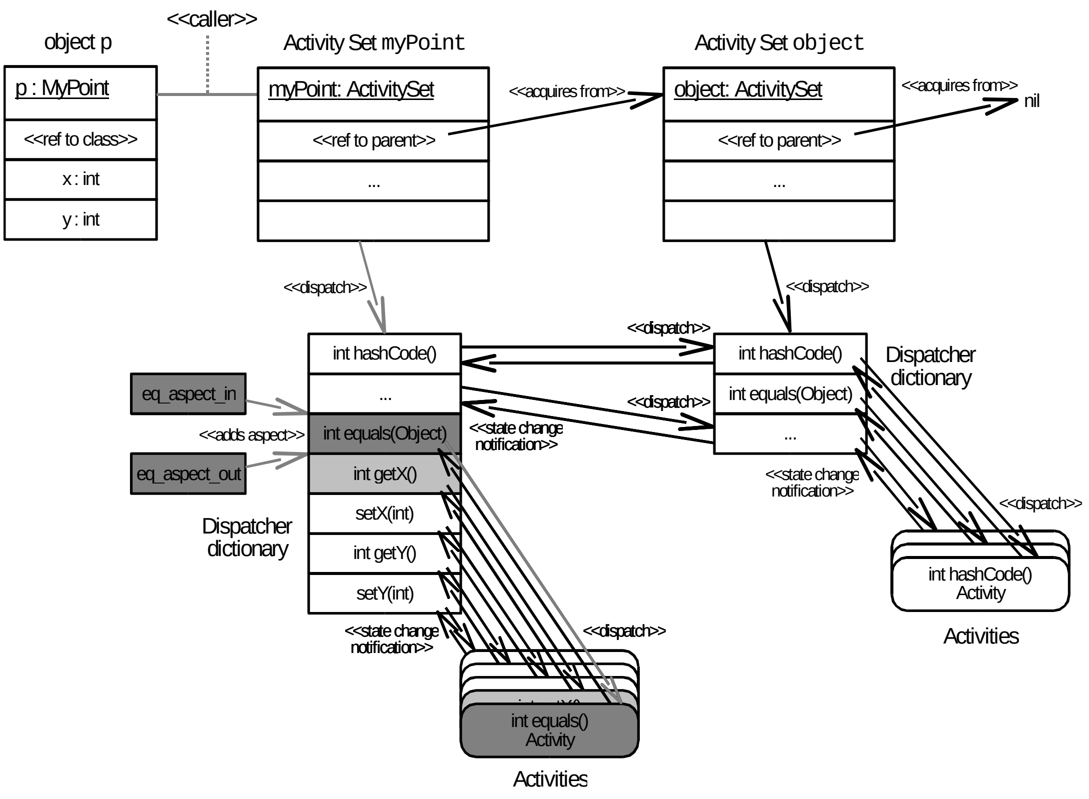


Figure 2.8-1 Dynamically adding aspects

The following code fragment shows the implementation:

Program 2.8-1 Adding aspects to Dispatchers (pseudo code)

```
ActivitySet myPoint = PCCRegistry.getOrCreateActivitySet(MyPoint.class);
ActivitySet object = PCCRegistry.getOrCreateActivitySet(Object.class);
myPoint.acquire(object);
Dispatcher dsp = myPoint.removeDispatcher("equals", parameterTypes);
dsp.setName("equals_orig");
myPoint.addDispatcher(dsp); // add the original "equals"-Dispatcher as
                            // "equals_orig"
myPoint.addActivity(        // add new Activity and Dispatcher "equals",
                            // including new aspects
   new Activity("equals", parameterTypes) {
      public Object doPerform(Object obj, Object[] args) {
         ActivitySet as = PCCRegistry.getActivitySet(obj);
         as.getDispatcher("eq_aspect_in").perform(obj, args);
         Object ret = as.getDispatcher(
                        "equals_orig").perform(obj, args);
         as.getDispatcher("eq_aspect_out").perform(obj, args);
      }
   ...
   }
);
```

We use the `equals` method of the class `MyPoint` for our example. The method is exposed as Activity, respectively as Dispatcher. To wrap our original Dispatcher, we remove it, rename it to `equals_orig` and add it again with the new name. Now the original Dispatcher name `equals` is not used any more, so we can use it for our own Dispatcher wrapping the original one. In the body of method `doPerform` we call `eq_aspect_in`, then the original equals-Dispatcher `equals_orig`, and then `eq_aspect_out`. Note that the implementations of `eq_aspect_in` and `eq_aspect_out` do not give any new insight into the discussed concept, therefore they are not shown here.

After having replaced the original `equals` Dispatcher with a wrapper, whenever client code is using a Dispatcher with this name, our new Dispatcher, including the aspects `eq_aspect_in` and `eq_aspect_out`, is used. Client code needs no change; existing calls to the Dispatcher `equals` can stay syntactically the same, but lead to the invocation of the wrapper Activity rather than the original Activity.

2.9 Notifications

Another feature of the introduced method dispatch mechanism is the ability to attach listeners to Dispatchers. Listeners are used when additional operations should be executed before or after the execution of a Dispatcher, or when an Activity or Dispatcher is added or removed or their states change. In contrast to aspects as described above, notifications cannot be used to override the behavior of the original Dispatcher. The actual Dispatcher and its associated Activity are executed in any way. To illustrate this capability, we enhance Program 2.6-1 by a notification mechanism.

Program 2.9-1 Dispatchers with notifications (pseudo code)

```
public class Dispatcher {
   Object listeners;
   ...
   Object perform(Object p, Object[] arguments) {
      Object result;
      notifyListenersBeforePerform(p, arguments);
      if (isActivity()) {  // if there is an Activity associated, invoke it
         result = getActivity().perform(p, arguments);
      }
      else if (getDispatcher() != null) { // otherwise use an attached Dispatcher
         result = getDispatcher().perform(p, arguments);
      }
      notifyListenersAfterPerform(p, arguments, result);
   }
   void addListener(DispatcherListener listener) {
      if (listeners == null) { listeners = new ArrayList(); }
      ((ArrayList)listeners).add(listener);
   }
   void notifyListenersBeforePerform(Object p, Object[] arguments) {
      ArrayList l = ((ArrayList)listeners);
      if (l == null) { return; }
      for (int i=0; i<l.size(); i++) {
         ((DispatcherListener)l.get(i)).beforePerform(p, arguments);
      }
   }
   void notifyListenersAfterPerform(Object p, Object[] arguments, Object result) {
      ArrayList l = ((ArrayList)listeners);
      if (l == null) { return; }
      for (int i=0; i<l.size(); i++) {
         ((DispatcherListener )l.get(i)).afterPerform(p, arguments, result);
      }
   }
...
```

The also modify the methods for adding and removing Activities and Dispatchers.

Program 2.9-2 Dispatchers with notifications (pseudo code)

```
   public void addDispatcher(Dispatcher dsp) {
      this.dsp = dsp;
      notifyListenersActivityAdded(dsp);
   }
   public void removeDispatcher(Dispatcher dsp) {
      this.dsp = null;
      notifyListenersActivityRemoved(dsp);
   }
   public void setActivity(Activity a) {
      notifyListenersActivityRemoved(this.a);
      this.a = a;
      notifyListenersActivityAdded(a);
   }
}
```

`notifyListenersActivityAdded` and `notifyListenersActivityRemoved` are implemented equivalently to `notifyListenersBeforePerform` and `notifyListenersAfterPerform`. We do not show the implementation in detail, since it does not add relevant information to this example.

The corresponding listener interface may look as follows:

Program 2.9-3 Dispatcher listener interface (pseudo code)

```
public interface DispatcherListener {
   void beforePerform(Object p, Object[] arguments);
   void afterPerform(Object p, Object[] arguments, Object result);
   void activityAdded(AbstractActivity a);   // AbstractActivity is the base
   void activityRemoved(AbstractActivity a); // class of Dispatcher and Activity
...
}
```

The relevant statements in Program 2.9-1 are the method calls `notifyListenersBeforePerform` and `notifyListenersAfterPerform`. Each `Dispatcher` manages a list of listeners (also called observers), and notifies with these methods the listeners whenever the Dispatcher is invoked, i.e. whenever the method `perform` is called. Note that listeners attach to specific Dispatchers and not directly to methods. This gives us the flexibility to get notifications of invoked Dispatchers only for specific children of an Activity Set providing the corresponding method. For example, we may want to be notified whenever the method `hashCode` is invoked on the Activity Set `myPoint`, but we do not want to be notified when `hashCode` is invoked on other children of the Activity Set `object` or on `object` itself. Remember, the method `hashCode` is implemented in the class `Object` and added to the corresponding Activity Set, therefore all children of `object` dynamically “inherit” the Dispatcher.

The following code fragment illustrates how to make use of the notification capability. First, we create an instance of `MyPoint`. After that, we retrieve a Dispatcher for method `hashCode` in class `MyPoint`. Then we attach a listener to the Dispatcher and invoke the Dispatcher on object `p`. The listener will be notified before and after the Dispatcher is invoked, respectively its associated method `hashCode` of object `p`.

Program 2.9-4 Using Dispatcher notifications (pseudo code)

```
public void foo() {
   MyPoint p = new MyPoint(10,20);
   ActivitySet myPoint = PCCRegistry.getOrCreateActivitySet(p);
   ActivitySet object = PCCRegistry.getOrCreateActivitySet(Object.class);
   myPoint.acquire(object);
   Dispatcher dsp = myPoint.getDispatcher("hashCode", new Class[] {});
   dsp.addListener(new MyListener());
   int hc = ((Integer)dsp.perform(p, "hashCode", new Object[] {})).intValue();
}
public class MyListener implements DispatcherListener {
   beforePerform(Object p, Object[] arguments) { ... }
   afterPerform(Object p, Object[] arguments, Object result) { ... }
   ...
}
```

This example is quite simple. There is only one place where the Dispatcher `hashCode` is invoked and and we know which arguments are passed. In this example, we actually need no notification. We can modify the code directly, for example, to trace the method call. In real applications it is more usual that there are many code fragments where a specific method is called. As users of third-party libraries or frameworks, we might want to be able to hook into methods of library or framework classes to trace method calls, trigger related operations, or to cancel method calls, but we do not have the source code and therefore cannot directly modify it, respectively, we should not do that for the sake of encapsulation. As library or framework developers we may want to provide such hooks rather than allow clients to modify our source code.

Note that with the PCoC framework, this notification mechanism is not directly provided by the Dispatcher class, but by a base class. See Section 4.3.2.

2.10 Flexible Delegation

So far, we have got insight into the most common facilities for method and message dispatch: method dictionaries, v-tables, and message interpreters. We have learned that delegation is a dispatch mechanism to share behavior between objects. We know that delegation may be used together with any of the dispatch facilities described above. In Section 2.2.5 we finally read about the Darwin model—a mechanism for type-safe delegation and dynamic component modification ([KNIES99]).

With a flexible method dispatch, we propose a more drastic approach than the Darwin model in terms of flexibility. It offers much flexibility (method objects and components can be added and removed entirely at runtime like in Smalltalk), no static declarations of parents are necessary, and “standard” compilers can be used (no proprietary compilers are necessary). Like Darwin, this mechanism can be integrated into an existing project (at least, if the flexible-method-dispatch framework has been implemented in the corresponding programming language) with little effort. Client code that needs to use the flexible method dispatch, must invoke methods via the `perform`-method of Activity Sets, instead of directly calling methods on the corresponding classes or objects. Known drawbacks are the worse performance compared to the static declaration system of Darwin, and the missing type-safety at compile time (type-safety is assured at runtime only).

The Dispatchers we first encountered in Section 2.3 can be enhanced to pass the receiver Activity Set (where a method is invoked) as argument to the dispatch method `perform`. The Activity Set is passed on to the actual method which can use it for subsequent method calls. This concept makes delegation possible across independent objects. We say it is a simulated delegation—Activity Sets are not integrated in a programming language and therefore must be passed explicitly when invoking Activities (no implicit “self” parameter; see below).

Compare the following implementation of the `perform` method with that of Program 2.6-1.

Program 2.10-1 Dispatchers with receiver context (pseudo code)

```
public class Dispatcher {
   ...
   Object perform(ActivitySet context, Object[] arguments) {
      if (isActivity()) {   // invoke retrieved method
         return getActivity().perform(context, arguments);
      }
      else {              // get method ref target
         return getDispatcher().perform(context, arguments);
      }
   }
   ...
}
```

The `perform` method passes on the (receiver) Activity Set from Dispatcher to Dispatcher and finally to a concrete Activity.

The corresponding modification in the `perform` method of the Activity Set class is illustrated below (compare to Program 2.7-3).

Program 2.10-2 Dispatchers with receiver context (pseudo code)

```
public final class ActivitySet {
   ...
   Object perform(String dispName, Object[] arguments) {
      int i=0;
      // get all Dispatchers with the given name
      Dispatcher[] dsps = dd.getDispatchers(dispName);
      while (i < dsps.length) {          // iterate over all Dispatchers
         Class[] parameterTypes = dsps[i].getParameterTypes();
         if (parameterTypes.length() == arguments.length) {
            int j=0;                     // search for a Dispatcher with a
            while (j<arguments.length) { // matching interface; the parameter types
                                         // must match the types of the given
                                         // arguments (actual parameters)
               if (!parameterTypes[j].isInstance(arguments[j])) {
                  break;
               }
               j++;
            }
            if (j >= arguments.length) { // found a matching Dispatcher; invoke it
               return dsps[i].perform(this, arguments);
            }
         }
         i++;
      }
      ... // throw error
      return null;
   }
   ...
}
```

Basically this implementation is very similar to Program 2.7-3. We do not pass the receiver object as first argument any more. Instead, with this implementation the Activity Set passes itself as argument to the corresponding Dispatcher. Passing the Activity Set instead of the original object may not make sense if all Dispatchers and Activities come from the same Activity Set, but it does if some Activities are associated with different Activity Sets. This is similar to the invocation of an instance method of a class, where we (implicitly) pass the receiver object (“this”) as argument, but the invoked methods may be defined in the class or any of the direct or indirect base classes of the object.

When a method is invoked, the method call must be delegated to the corresponding class of the object where it has been invoked. If no receiver is specified explicitly, the current object (“this”) is used. All object-oriented programming languages support this “class-based delegation” by means of the implicit “this” parameter.

For dynamic component modification it makes sense that objects of independent classes can also be combined to units, quite like an object represents a unit combining the behavior of its class and its base classes. There is only Lava which has a built-in object-based delegation mechanism in addition to class-based delegation. Other approaches simulate object-based delegation by, for example, using hooks or storing references in the parent objects of delegation relations. Simulation techniques and their drawbacks are discussed and summarized in [HARRIS97] and [KNIES98]. [KNIES99] summarizes disadvantages of simulation techniques (Pages 6f).

The mandatory static declaration of parents in Lava (see the beginning of this section) may not be flexible enough for some cases, since it requires anticipation for where a hook might be needed. This leads us back to our delegation approach.

Now that the Activity Set where a Dispatcher is invoked (the receiver) is passed to the Activity, we can invoke other Dispatchers on the same Activity Set from within that Activity. In this case the Activity Set has the same meaning for dynamic method calls across different objects as the implicit parameter “this” has for static method calls across different classes.

Program 2.10-3 Delegation across different objects (pseudo code)

```
// add all methods of MyPoint to a new Activity Set: setX, setY, getX,...
MyPoint p = new MyPoint();  // a point object
Integer z = new Integer(17);  // the z-value which extends p to a 3D point
ActivitySet a = PCCRegistry.getOrCreateActivitySet(p);

a.addActivity(                                          // add new method "setZ"
   new BoundActivity("setZ", new Class[] {Integer.class}, z) {
      public Object doPerform(ActivitySet a,
                              Object[] args)           // executed when the method
      {                                                // is called
         ((Integer)obj).set((Integer)args[0]);
         return null;
      }
   ...
});
a.addActivity(                                         // add new method "getZ"
   new BoundActivity("getZ", new Class[] {}, z) {
      public Object doPerform(ActivitySet a,
                              Object[] args)           // executed when the method
      {                                                // is called
         return obj;
      }
   ...
});
a.addActivity(                                          // add new method "setXYZ"
   new BoundActivity("setXYZ",
                   new Class[] {Integer.class, Integer.class, Integer.class},
                   z) {
      public Object doPerform(ActivitySet a,
                              Object[] args)            // executed when the method
      {                                                 // is called
         a.perform("setX", new Object[] {args[0]});
         a.perform("setY", new Object[] {args[1]});
         ((Integer)obj).set((Integer)args[2]);
         return null;
      }
   ...
});

a.perform("setZ", new Object[] {10});
Integer result = (Integer)a.perform("getZ", new Object[] {});
a.perform("setXYZ", new Object[] {10, 20, 30});
```

In this example, we extend an Activity Set by an Activity from another Activity Set in order to show the value of the flexible method dispatch. As basis for this example we use class `MyPoint` of Program 2.2-5 (Page 14), and the Activity Set client code of Program 2.7-4 (Page 29).

In this example we introduce our own Activity-class `BoundActivity`. As opposed to instances of class `Activity`, a bound Activity is associated with a particular object. This object is passed as additional argument (`z`) to the `BoundActivity` constructor and is stored as member variable `obj`. The method `getOrCreateActivitySet` creates an Activity Set and initializes it with Activities for the methods of `p`. Instances of `BoundActivity` are created and added to the Activity Set for each method of `p`, as opposed to Program 2.7-1 where instances of class `Activity` were added.

We explicitly add the methods `setZ`, `getZ`, and `setXYZ` to the Activity Set `a`. Although point `p` has only two dimensions (`x`, `y`) and the `z`-parameter is stored in a separate object, the associated Activity Set `a` can be used as 3D-point. We say, it simulates a 3D-point. Dispatcher calls are either delegated to Activities associated with the original object `p` (`setX`, `setY`) or to a bound Activity added explicitly to the Activity Set (`setZ`).

As long as our objects or components are used through Activity Sets, we or users of our components have the flexibility to modify the functionality at runtime without the need to recompile and reload the corresponding client code.

Note that we used an argument array in this example, but in a concrete implementation of this concept we may rather use an argument list class which supports named parameters. In addition to that, we can introduce convenience methods to make the use of the flexible method dispatch easier.

You find more information about delegation and its integration in the framework introduced in this thesis in Section 4.6. Also relevant are Sections 5.4 and 6.1.

2.11 Prioritized Flexible Method Dispatch

Now let us take a closer look at parent/child-relationships between Activity Sets. We know that an Activity Set can acquire (dynamically inherit from) other Activity Sets, just like a class can inherit from other classes.

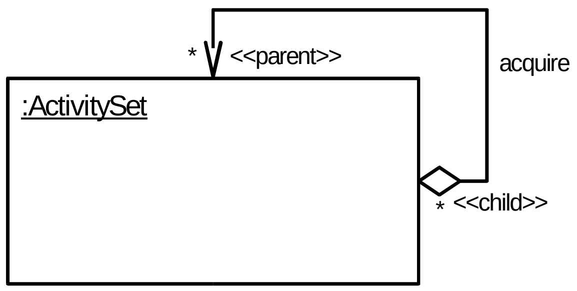


Figure 2.11-1 Parent/child relationship

This relationship can be used to switch between different implementations of the same kind of service. This includes particular operations or whole objects or components.

To illustrate this idea, we take the example of Figure 2.2-5 and adapt it slightly. Instead of classes and their instances, we use the corresponding Activity Sets. We assume that the class `Shape` associated with the Activity Set `shape` contains only the data structure for a particular shape. It has methods for setting and getting its position, extent, etc., but no method for displaying the shape on the screen.

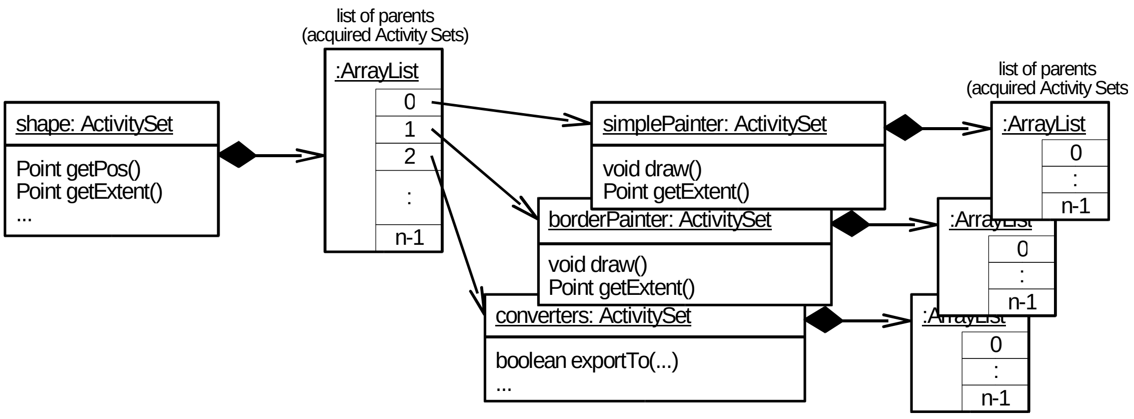


Figure 2.11-2 Prioritized acquisition

Using Activity Sets, we can dynamically enhance `shape` by additional functionality. We may provide several painters to display `Shape` objects on the screen. Each painter implements the method `draw` which does the actual job. We let the Activity Set `shape` acquire our painters, respectively their methods. Additionally we acquire a converter which provides functionality to export the shape to a graphics file.

Note that an acquisition relationship allows multiple parents, therefore a call of `acquire` adds a new parent, and does not replace a previously acquired parent.

Program 2.11-1 Acquisition of Activity Sets (pseudo code)

```
ActivitySet shape = PCCRegistry.getOrCreateActivitySet(new Shape());
ActivitySet simplePainter = PCCRegistry.getOrCreateActivitySet(
                               new SimplePainter());
ActivitySet borderPainter = PCCRegistry.getOrCreateActivitySet(
                               new BorderPainter());
ActivitySet converter = PCCRegistry.getOrCreateActivitySet(
                               new Converter());
shape.acquire(simplePainter);
shape.acquire(borderPainter);
shape.acquire(converter);
```

For each Dispatcher of the acquired Activity Sets (parents), a Dispatcher is added to `shape`. We can then invoke, for example, the Dispatchers `draw` and `exportTo` on `shape`, even if the class `Shape` does not provide the corresponding methods. When such a Dispatcher is invoked, the call is delegated, following the corresponding Dispatchers to the Activity Set which has the actual (bound) Activity. Finally the static method associated with the bound Activity is invoked on the corresponding object.

In our case, we have two painters and therefore we get a name clash with the `draw` Dispatchers. We could remove one painter, but we assume that one painter reuses Dispatchers or more precisely the associated Activity and static method of the other, and therefore both are required.

To cope with this problem, the parents, respectively their Dispatchers, are ranked. The ranking of a parent is determined by its position in the list of parents. We say, Activity Sets with a lower index have a higher priority.

In order to give `borderPainter`, respectively its Dispatchers a higher priority in `shape`, we can use any of the following statements:

Program 2.11-2 Setting the focus to borderPainter (pseudo code)

```
shape.moveTo(borderPainter, 0);
borderPainter.setFocus();
```

Both statements lead to a new ranking in the list of parents of the Activity Set `shape`. In contrast to `moveTo`, a call of `setFocus` gives `borderPainter` the highest ranking in all its (acquisition) children, and not only in `shape`.

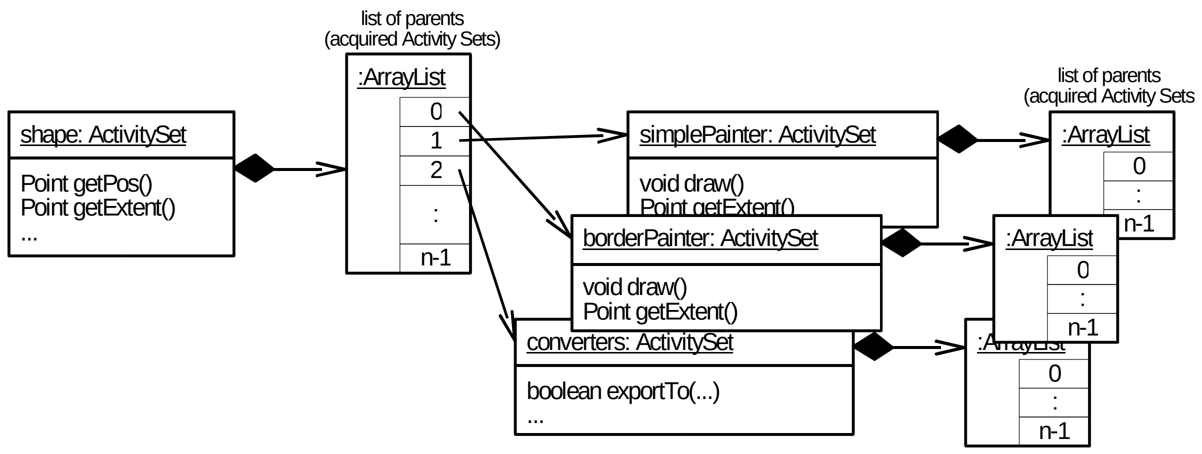


Figure 2.11-3 Prioritized acquisition (2)

When we invoke the Dispatcher `draw` on `shape`, the call is delegated to `borderPainter` rather than to `simplePainter` now. When we invoke the Dispatchers `exportTo`, the call is delegated to `converter` as before, since none of the higher priority Activity Sets provide the corresponding Activity.

For broadcasts, that is when requests are delegated to two or more Activity Sets, the order in which the actual methods are executed is determined by the ranking of the Activity Sets in the list of parents. In order to support broadcasts, we have to make some enhancements to the `Dispatcher` class.

Program 2.11-3 Dispatcher implementation (pseudo code)

```
public class Dispatcher {
   ...
   Object broadcast(ActivitySet a, Object[] arguments) {
      if (isActivity()) {
         return getActivity().perform(p, arguments); // invoke retrieved Activity
      }
      else {
         int i;
         for (i=0; i < dsps.size(); i++) {
            ((Dispatcher)dsps.get(i)).perform(p, arguments);
         }
         return null;
      }
   }
   ...
   private Activity a;
   private ArrayList dsps;
}
```

The most relevant changes are the list of (parent) Dispatchers instead of a single Dispatcher, and the new `broadcast` method which delegates to each Dispatcher in the list (cf. Program 2.6-1).

2.12 Containment Hierarchy

Besides the possibility of ordered acquisition (dynamic inheritance) and delegation, we may need to group Activity Sets, like classes can be grouped through name spaces. For that reason, Activity Sets have a path instead of only a name. See also Section 5.3.

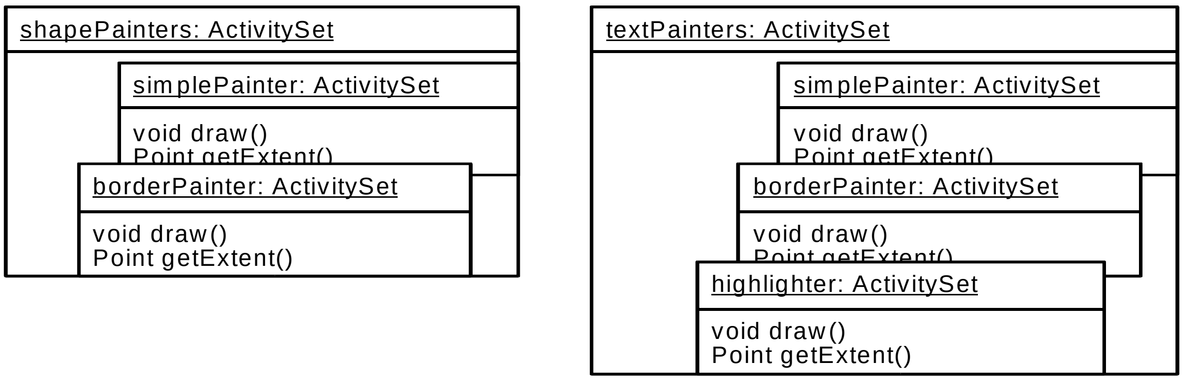


Figure 2.12-1 Activity Set hierarchy

In this example, we have two Activity Set groups: `shapePainters` and `textPainters`. The corresponding paths are "`|ShapePainters`" and "`|TextPainters`". We use character "|" as path delimiter to avoid confusion with file paths ("/", "\") or operators "." and "->". The group `shapePainters` contains the two Activity Sets `simplePainter` and `borderPainter`. The group `textPainters` contains the Activity Sets `highlighter`, `simplePainter` and `borderPainter`. The latter two are identically named but different from those of `shapePainters`.

The given Activity Set tree results in following Activity Set paths: "`|ShapePainters`", "`|ShapePainters|SimplePainter`", "`|ShapePainters|BorderPainter`", "`|TextPainters`", "`|TextPainters|Highlighter`", etc.

The path of an Activity Set can be set as follows:

Program 2.12-1 Setting the path (pseudo code)

```
borderPainter.setPath("|ShapePainters|BorderPainter");
borderPainter.setContainer(shapePainters);
```

Both statements are equivalent. `shapePainters` becomes the container of `borderPainter`. Once created, the Activity Set can be retrieved from the registry as follows:

Program 2.12-2 Setting the path (pseudo code)

```
ActivitySet borderPainter =
   PCCRegistry.getActivitySet("|ShapePainters|BorderPainter");
```

Activity Set groups have only one purpose: to distinguish between semantically similar Activity Sets with the same name, but different behavior. They are ordinary Activity Sets and therefore they can also acquire their nested Activity Sets (such with subpaths), which we call *elements*. However, by default, containers do not automatically acquire their elements. They may provide their own Activities. The path is stored as member of the `ActivitySet` class. We use the terms *container* and *elements* for containment relationships, in order to avoid confusion with the terms *parent* and *children* used in delegation relationships.

2.13 Remarks

So far, we have seen how method dictionaries (Smalltalk) and v-tables (C++/Java) work, what delegation is and how it can be realized. We also learned about the Darwin model, its delegation concept, and its realization in the programming language Lava.

We have introduced a new message dispatch mechanism that gives a certain level of flexibility, but requires a different way of method invocation (more precisely, dynamic method invocation) compared to ordinary method calls. To leave existing code unchanged, a code generation tool could be provided which merges functionality or code into existing classes or components. This has some drawbacks: the code generator must replace every single method call by a dynamic one, which is error-prone unless the code-generator includes a perfect source-code parser, or the code-generator is perfect, and thus expensive and slow; the code generator must be invoked for your original source code after each change. Another way is to integrate the dispatch mechanism into a programming language, respectively a compiler. An approach for dynamic component modification and integration into a JAVA-based language is shown by Kniesel G. in *Type-Safe Delegation for Run-Time Component Adaption* [KNIES99]. The paper resumes in a very detailed way the advantages and issues of dynamic component modification and delegation through object-based inheritance.

With respect to the classification in Section 2.2.5, the flexible method dispatch presents an approach to unanticipated, dynamic, selective, wrapper-based, object-preserving component modification, quite like the Darwin model. With this approach we need not anticipate possible future requirements for enhancements of our components, since modifications are generally possible at runtime. Therefore this approach is dynamic and unanticipated. It is also selective, because with bound Activities an Activity Set is associated with particular objects rather than with a class. On the other hand, Activity Sets can also be used with unbound Activities only, and therefore the concept is also suitable as class-based (global) approach. The approach is wrapper-based (Activity Sets can be wrappers for objects or components). The adapted wrapper can coexist with the original component, or replace it. There can also be many wrappers of the same object or component at the same time.

Let use resume some other aspects of the flexible method dispatch introduced in this thesis.

The advantages of the flexible method dispatch are:

- Dynamic modifications
  - Method objects (Activities) can be added and removed at runtime, while almost no reorganization of Dispatcher dictionaries is necessary for subclasses after adding or removing method objects to/from a parent object.
    - ✗ In contrast, v-tables (like in C++ or Java) are static and the entries have static indices; a method has the same index in the v-table of derived classes as in that of the base class; reorganizing the related v-tables of derived classes after changing entries in the base class would be very time-consuming at runtime; parts of v-tables of derived classes have to be moved accordingly, if a base class gets a new method. See Figure 2.2-2.
    - ✗ Smalltalk method dictionaries need no reorganization at all after such a change, since the base class methods are only referenced in the base-class method dictionary. However, for looking up a method, the class hierarchy must be searched for the method, which is time-consuming at runtime. A method lookup cache optimizes the message dispatch drastically (according to [KRASN84], an appropriate method cache can have hit ratios as high as 95%, reduce method lookup time by factor 9, and increase overall method system speed by 37%).
    - ✗ With the flexible method dispatch, Dispatcher dictionaries only keep indirect references to method objects (Activities). When an Activity is added to an Activity Set, all delegation children get Dispatchers to the corresponding Dispatchers of their parents.
    - ✗ See Program 2.5-1 and Program 2.7-4 for Dispatcher dictionary and references.
  - With bound Activities (see Program 2.10-3), functionality can be added to an Activity Set, although it is defined in another: bound Activities store a reference to its associated

  Activity Set, respectively its provider. This way, an Activity can always access the environment (Activity Set) where they are deployed, and the environment, where they are created (the Activity Set of the original provider).

  - Activities, respectively their Dispatchers can be wrapped dynamically, aspects can be added to Dispatchers: code can be added before and/or after the original method code by wrapping it. See Program 2.8-1.

- Efficient dynamic method look-up
  - A method object (Activity) respectively Dispatcher only needs to be looked up in the Dispatcher dictionary of an object's associated Activity Set, and not also in its parents like in the Smalltalk method look-up algorithm. We follow Dispatchers, starting with the looked-up one until we get an Activity. See Program 2.4-1.
  - Search of Activities in parents is done through directly linked Dispatchers. See Program 2.4-1 and Program 2.6-1.
  - This solution is not as fast but more dynamic than v-tables where methods are looked up with their integer index created at compile time.
  - It is more efficient than message interpreters of Oberon where message is checked for its type with WITH-DO clauses. The message type checking has runtime complexity `O(n)`, where `n` is the number of types to be checked. In contrast to that, the lookup-time of the flexible message dispatch is constant.
- Dynamic reorganization of acquisition (dynamic inheritance) relationships. An Activity Set can acquire or discard another at any time. See Program 2.11-1.
- Activity Sets can be dynamically grouped by setting their path adequately, or by setting the parent path.
- Ordered multicasts (broadcasts) and other special dispatch strategies can easily be realized (like in the message objects of Oberon), respectively are even included in the dispatch mechanism. (Bound) method objects can be added to an Activity Set even if they belong to others. All added Activities can be reached through Dispatchers. A Dispatcher may collect all its descendants and call `perform` for each.
- Activities can be packaged with arguments and invoked later on. You find this feature also in the Oberon message interpreter mechanism.
- Delegation through passed reference to the receiver. The scope (the Activity Set) where an Activity originally was requested can be passed through the involved Dispatcher chain and can be used for subsequent requests; different objects can use the same Activity Set; the concept provides a common “this” for instances of independent classes.
- Encapsulation of the distribution of components. Activity Set contents can live behind interface standards such as COM, XMLRPC, etc., but these interfaces can be hidden to the developer for convenience; components and operations from various sources can be attached and associated with Activity Sets dynamically.
- Hooks for method calls. Listeners can be attached to Dispatchers. They are notified before and after an Activity is requested, respectively a corresponding Dispatcher is invoked. See Program 2.9-1 and Program 2.9-4.

- Method state support. Since methods are represented by objects, more precisely Activities, they can be enhanced by features such as states—typical states are “Enabled”, “Disabled”, etc. For example, beside method `perform` the `Activity` class could provide a method `getState` to retrieve the current state, and a method `stateChanged` to notify listeners of changes. States can be set and switched at any time. They may depend on factors such as given application licenses (full version, demo version), etc. Activities may have different behavior depending on the state. Common object oriented programming languages do not support such a feature.

Disadvantages are:

- Increased memory footprint for Dispatcher dictionaries
  - Compared to Smalltalk method dictionaries, additional entries for Dispatchers of parents are necessary in Dispatcher dictionaries of children.
  - Compared to v-tables, Dispatcher dictionaries are less memory-efficient. Dispatchers are stored in dictionaries with strings as keys, instead of flat, memory-saving method pointer arrays. V-tables also contain method pointers of base classes like the optimized Dispatcher dictionaries, but a flat array is always smaller.
  - There needs to be an Activity Set for each object or component where the flexible method dispatch is applied. To avoid massive memory consumption it is reasonable to use it rather for components than for each small object.
- Slower than v-tables. Dispatchers have to be looked up in dictionaries using their name, respectively the hash-code of the name, as key. It depends mainly on how much time the look-up of a Dispatcher name takes. The time for resolving Dispatchers is not relevant (following the Dispatcher chain to a concrete Activity). In v-tables, the index representing a method is calculated at compile-time; method look-up is only the access to the method pointer at an index in the corresponding v-table.
- The static interface of a class does not reflect which messages can be handled by instances of the class. It is difficult to realize at compile time which objects are related through message forwarding at runtime.
- Invalid dynamic method calls are not recognized at compile time, as opposed to static method calls.

In contrast to the proposed solution (indirect references), a less time-consuming solution is to store references to the Activities of parents directly in the Dispatcher dictionaries of the children, similar to v-table-approach.

However, in the case of a v-table-approach, method indexes have to be recalculated. Parts of v-tables of derived classes must be moved when a method is added or removed in a base class. For example, if the method `clone` is added to the base class `Object`, a pointer to this method has to be added to the v-table of `MyPoint` as well as to those of all other derived classes. The tricky thing is that the new method must be added to the v-table of `MyPoint` at the end of the section that comes from `Object` and before the pointers to the methods implemented directly in `MyPoint`. It gets even more complicated, if multiple inheritance is supported (which we use for multicasts). In this case, it must be ensured that a method that is inherited from more than one base class gets the same method index in the v-tables of all related classes. See Figure 2.2-2 for an overview of v-tables.

Also, like v-tables, Dispatcher dictionaries of all child Activity Sets have to be reorganized, when an Activity is added to a parent. The difference is, that the positions of Dispatchers in Dispatcher dictionaries are not relevant, since they are looked up via names and not via pre-calculated indexes.

When using direct references or pointers to methods (such as v-table method pointers), the method look-up may be slightly faster (for inheritance depth > 1) than with indirect references, but the impact is only `n × <time for a method call>` (see Program 2.4-1), where `n` is the inheritance depth of the current class.

With our approach, client code can attach listeners to Dispatchers at each indirection level, which allows clients to be notified whenever an Activity on a specific class is invoked. This is useful for tracing calls or adding aspects from different sources.

Finally, the flexible method dispatch enables delegation across independent objects, and dynamic component modification, using Activity Sets, Dispatchers, and bound Activities. Each Activity of an Activity Set may be bound to a different object. In fact, in PCoC, each Activity is in any case bound to an object. There are no unbound Activities. That is, they are always used on object level (selective approach), rather than on class level (global approach).

Like .NET delegates, Dispatchers can be used for multicasts, and for forwarding to methods (in our case through Activities) of different objects or classes.

Nevertheless, there are still some open issues. The performance and memory-consumption can be improved through better implementations. The current implementation as framework is used in a productive environment (for a commercial application), and will be reworked in order to get cleaner code. The dispatch mechanism could be integrated in a programming language in order to make it easier to use.

The following sections give a detailed insight in how these concepts are deployed.

3 PCoC Terms

This section explains essential basics of PCoC for better understanding its usage as shown in Chapter 4, *Using PCoC*.

3.1 Overview

An application can basically consist of interactive and non-interactive parts—so-called components.

In the context of this thesis, we use the term *Tool* for interactive components of (G)UI applications. A Tool provides a user interface (expects user interaction) and operations which we call *Activities*. For example, a word processor component can be a Tool.

Tools need Materials to work on. For this reason we use the term *Material* for containers of Activity arguments and return values.

The meanings of the terms *Tool* and *Material* are quite similar to those of the WAM-metaphor. WAM stands for Werkzeug-Automat-Material (Tool-Automaton-Material). See [ZULLIG98].

Note that the term *Tool* is used for many different things in other sources. People use it for whole applications (grep tool, shell tool, image manipulation tool), for objects (a selected brush, pen, or crop utility in an image manipulation program), and also for very basic things such as the mouse cursor.

The non-interactive components of PCoC applications are called *Service Providers*. For example, a component providing file system functionality (`moveFile`, `openFile, etc.`) via Activities is a Service Provider. Both, Tools and Service Providers are so-called *Activities Providers*—components which expose a part of their functionality as Activities.

The following sections describe PCoC terms and their meanings, and differences to definitions of the WAM-metaphor.

3.2 Activities Providers

3.2.1 Overview

An *Activities Provider* is a component providing its functionality via Activities. The figure below illustrates how PCoC classes are related to each other; the annotations explain the responsibilities of the framework classes.

The classes highlighted in grey are those implemented by the component developer: Service Providers, the services they provide, and Simple Tools. A Combined Tool is a generic class, and is created and initialized by PCoC depending on the configuration. See Section 4 for details.

The Activities, Dispatchers, and dynamic containment and acquisition information of an Activities Provider are kept in its Activity Set which is created when the Activities Provider is initialized.

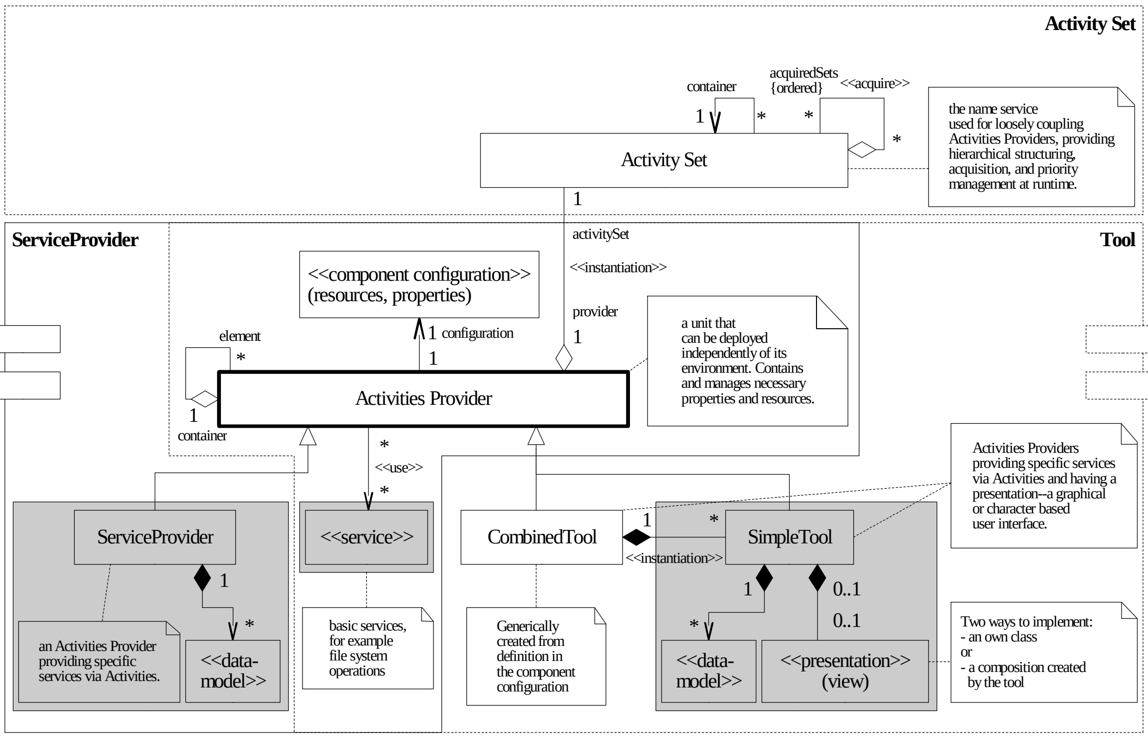

Activity Set
container
acquiredSets {ordered}
<<acquire>>
1
*
*
*
Activity Set
the name service used for loosely coupling Activities Providers, providing hierarchical structuring, acquisition, and priority management at runtime.
1
ServiceProvider
activitySet
Tool
<<instantiation>>
<<component configuration>> (resources, properties)
provider
1
configuration
element
1
a unit that can be deployed independently of its environment. Contains and manages necessary properties and resources.
Activities Provider
container
<<use>>
Activities Providers providing specific services via Activities and having a presentation--a graphical or character based user interface.
ServiceProvider
<<service>>
CombinedTool
SimpleTool
<<instantiation>>
0..1
an Activities Provider providing specific services via Activities.
<<data-model>>
basic services, for example file system operations
Generically created from definition in the component configuration
<<data-model>>
<<presentation>> (view)
Two ways to implement: - an own class or - a composition created by the tool

Figure 3.2-1 Activities Provider Architecture

See the sections below for definitions of Tool, Service Provider, Activity, Dispatcher, and Task.

3.2.2 Class(es)

The corresponding class is `PCCActivitiesProvider`.

3.3 Activity Sets

3.3.1 Overview

An *Activity Set* is an object providing a set of operations called Activities, their Dispatchers (which delegate requests for invoking Activities to the corresponding Activity Sets), and a corresponding lookup mechanism for Activities and Dispatchers. It is often associated with an Activities Provider (see above). Activity Sets are distinguished by their path (for example, “|A|B”).

This concept is useful for building acquisition relationships, and for grouping semantically related objects or components. See also Section 2.4 and Section 2.12.

Activity Sets are added automatically to a registry. The registry is realized as a singleton class `PCCRegistry`. It can be used to look up Activity Sets (using their path), and their associated Activities Providers.

3.3.2 Class(es)

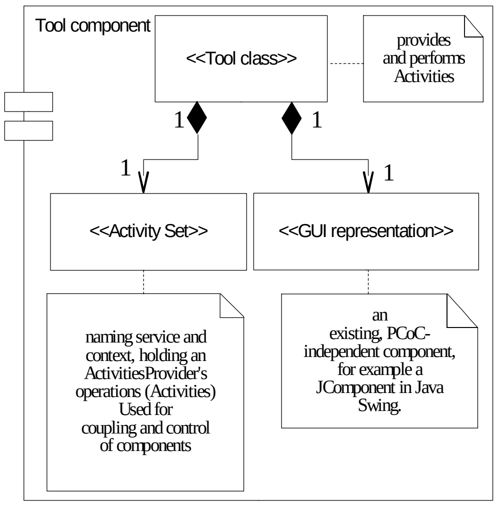
The corresponding class is `PCCActivitySet`.

3.4 Tools

3.4.1 Overview

A *Tool* is an Activities Provider which expects user interaction. Tools provide a specific object as visual representation, for example a JComponent in Java. The main functionality (operations on data, algorithms, etc.) is directly implemented in the Tools class.

In the WAM-metaphor, the term *Tool* can, for example, stand for a notepad or organizer application. In the context of this thesis, Tools are rather those parts of applications that actually provide the functionality. As in the WAM-metaphor, [ZULLIG98], Page 280, Simple Tools can be combined to more complex Tools, which are then called Combined Tools.



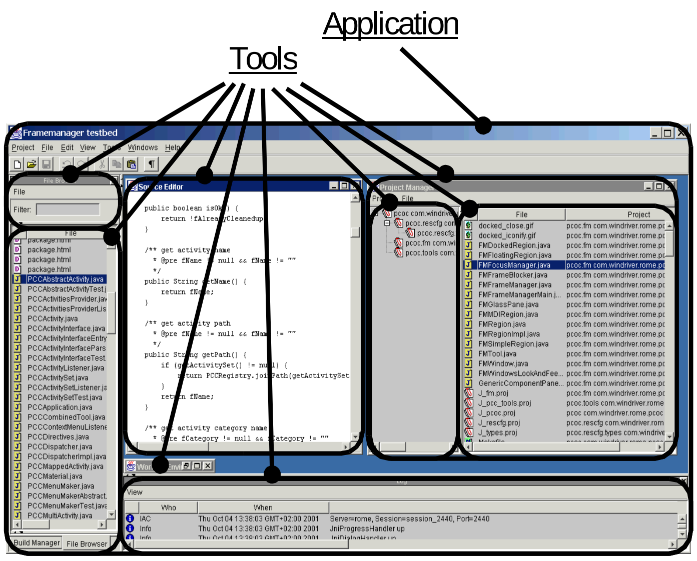
Tool component
<<Tool class>>
provides and performs Activities
1
1
1
1
<<Activity Set>>
<<GUI representation>>
naming service and context, holding an ActivitiesProvider's operations (Activities) Used for coupling and control of components
an existing, PCoC-independent component, for example a JComponent in Java Swing.

Figure 3.4-1 A Tool

Figure 3.4-1 depicts the relationships between some classes that form a Tool.

3.4.2 Examples

Figure 3.4-2 shows what Tools look like in the context of PCoC and this thesis.

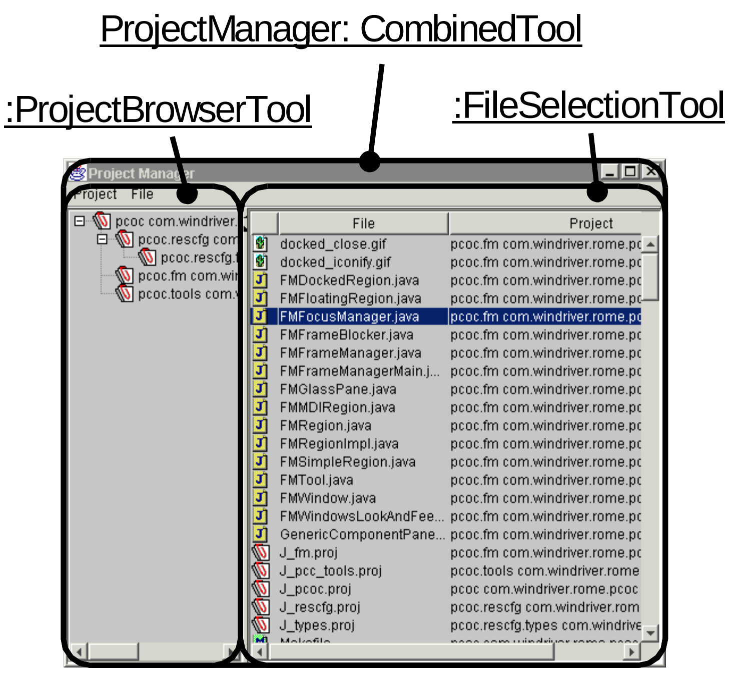


```
public boolean isOk() {
    return !fAlreadyCleanedup
}

/** get activity name
  * @pre fName != null && fName != ""
  */
public String getName() {
    return fName;
}

/** get activity path
  * @pre fName != null && fName != ""
  */
public String getPath() {
    if (getActivitySet() != null) {
        return PCCRegistry.joinPath(getActivitySet
    }
    return fName;
}

/** get activity category name
  * @pre fCategory != null && fCategory != ""
```

	Who	When	
●	IAC	Thu Oct 04 13:38:03 GMT+02:00 2001	Server=rome, Session=session_2440, Port=2440
●	Info	Thu Oct 04 13:38:03 GMT+02:00 2001	JniProgressHandler up
●	Info	Thu Oct 04 13:38:03 GMT+02:00 2001	JniDialogHandler up

Figure 3.4-2 A PCoC application

Tools can be composed to more complex Tools. The following figure shows a typical Combined Tool (`ProjectManager`) containing some Simple Tools.



pcoc com.windriver.
pcoc.rescfg com
pcoc.rescfg.t
pcoc.fm com.wi
pcoc.tools com.

File	Project
docked_close.gif	pcoc.fm com.windriver.rome.pc
docked_iconify.gif	pcoc.fm com.windriver.rome.pc
FMDockedRegion.java	pcoc.fm com.windriver.rome.pc
FMFloatingRegion.java	pcoc.fm com.windriver.rome.pc
FMFocusManager.java	pcoc.fm com.windriver.rome.pc
FMFrameBlocker.java	pcoc.fm com.windriver.rome.pc
FMFrameManager.java	pcoc.fm com.windriver.rome.pc
FMFrameManagerMain.j...	pcoc.fm com.windriver.rome.pc
FMGlassPane.java	pcoc.fm com.windriver.rome.pc
FMMDIRegion.java	pcoc.fm com.windriver.rome.pc
FMRegion.java	pcoc.fm com.windriver.rome.pc
FMRegionImpl.java	pcoc.fm com.windriver.rome.pc
FMSimpleRegion.java	pcoc.fm com.windriver.rome.pc
FMTool.java	pcoc.fm com.windriver.rome.pc
FMWindow.java	pcoc.fm com.windriver.rome.pc
FMWindowsLookAndFee...	pcoc.fm com.windriver.rome.pc
GenericComponentPane...	pcoc.fm com.windriver.rome.pc
J_fm.proj	pcoc.fm com.windriver.rome.pc
J_pcc_tools.proj	pcoc.tools com.windriver.rome
J_pcoc.proj	pcoc com.windriver.rome.pcoc
J_rescfg.proj	pcoc.rescfg com.windriver.rom
J_types.proj	pcoc.rescfg.types com.windrive

Figure 3.4-3 A Combined Tool

`ProjectBrowserTool` and `FileBrowserTool` are Simple Tools—they are classes (directly or indirectly) derived from `PCCSimpleTool`. Combined Tool is a generic class not subject for deriving subclasses. Instances, such as `ProjectManager`, are automatically generated from configuration files.

Simple Tools can also contain other Simple Tools, but they are not generated automatically in contrast to Combined Tools. Composition must be done in source code. This may be solved differently in other frameworks.

3.4.3 Class(es)

The corresponding classes are `PCCSimpleTool` and `PCCCombinedTool`.

3.4.4 Remarks

Chapter 4, *Using PCoC*, explains Tools and their collaboration in more detail.

3.5 Service Providers

3.5.1 Overview

Services, such as file system functionality, can be realized as *Service Providers*. A Service Provider is a non-interactive Activities Provider, as opposed to Tools. It exposes services via Activities.

It is basically implemented like a Tool, except that it does not provide a specific GUI representation object. Figure 3.5-1 depicts the relationship between some classes that form a Service Provider.

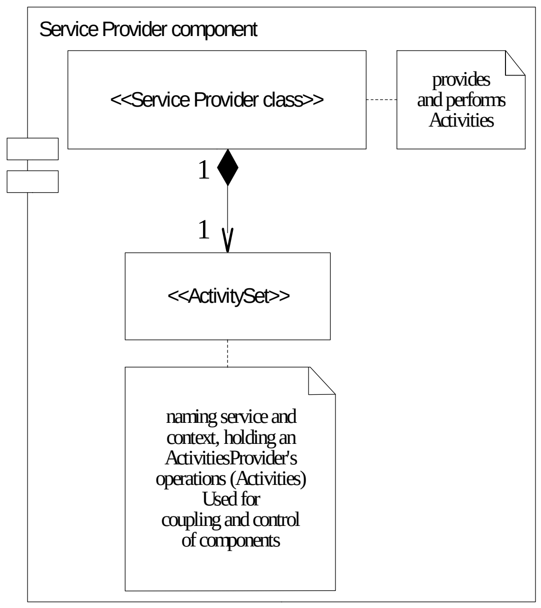

Service Provider component
<<Service Provider class>>
provides and performs Activities
1
1
<<ActivitySet>>
naming service and context, holding an ActivitiesProvider's operations (Activities) Used for coupling and control of components

Figure 3.5-1 A Service Provider

3.5.2 Class(es)

The corresponding class is `PCCServiceProvider`.

3.5.3 Remarks

Chapter 4, *Using PCoC*, describes a sample implementation and configuration of a Service Provider.

3.6 Activities

3.6.1 Overview

Activities are operations provided by Activities Providers (Tools, Service Providers). An Activity is implemented as class and is treated as first class object by PCoC.

Activities correspond to bound method objects as described in Section 2.10 (Program 2.10-3). An Activity can be instantiated by an Activities Provider, and can subsequently be performed. A single method (`doPerform`) of an Activity implements its actual functionality. See Program 4.2-3.

Activities can be retrieved, and added to and removed from Activity Sets at runtime. This is an enhancement to Java reflection, which supports retrieving of methods at runtime, but not adding them to objects at runtime.

Activities support different states, including “Enabled” and “Disabled”.

Detailed concepts and examples of Activities are described in the following sections.

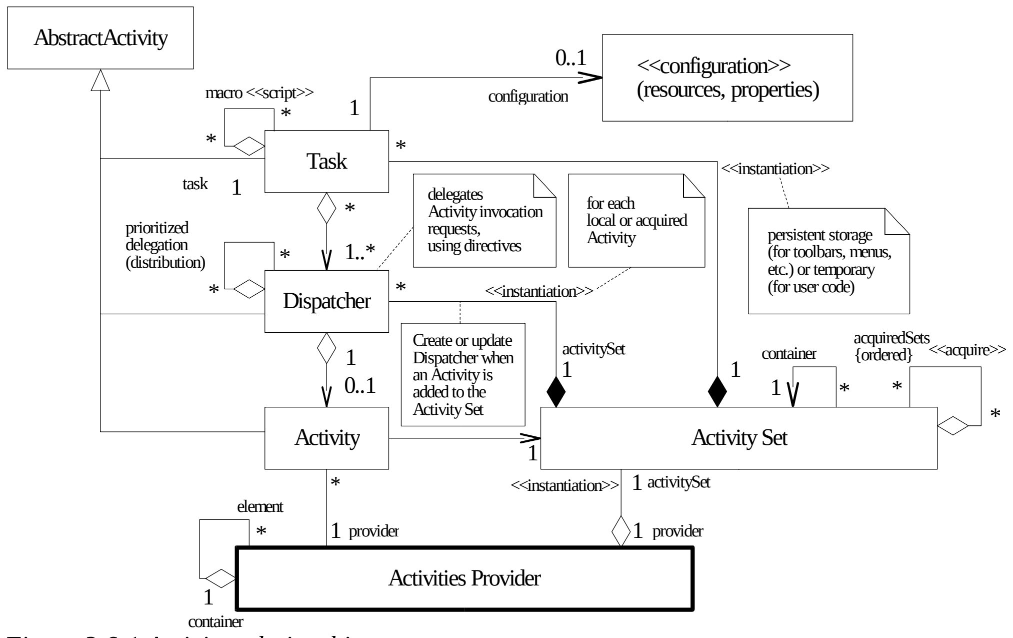

AbstractActivity
<<configuration>> (resources, properties)
0..1
configuration
macro <<script>>
Task
task
prioritized delegation (distribution)
Dispatcher
Activity
delegates Activity invocation requests, using directives
for each local or acquired Activity
<<instantiation>>
persistent storage (for toolbars, menus, etc.) or temporary (for user code)
Create or update Dispatcher when an Activity is added to the Activity Set
activitySet
container
acquiredSets {ordered}
<<acquire>>
Activity Set
element
provider
Activities Provider
container
1..*

Figure 3.6-1 Activity relationships

This picture illustrates how Activity Sets, Activities, Dispatchers, and Tasks are related to each other. See later in this chapter for definitions of the terms Dispatcher and Task.

3.6.2 Class(es)

Base class for all kinds of Activities is `PCCAbstractActivity`. Concrete classes must be derived from `PCCSingleActivity` or `PCCMultiActivity` which are both derived from `PCCActivity`. A `PCCSingleActivity` represents a single operation, e.g. `moveFile` to move a file in a file system. `PCCMultiActivity` represents a group of Activities with the same interface, e.g. `showRecentFile` to reopen a recently opened file in a proper Tool. The Activities of this group are accessed with an index.

Note that there are other classes derived from `PCCAbstractActivity` which semantically differ from single and multiple Activities. Examples are `PCCDispatcher` and `PCCTask` (see Figure 3.6-1).

3.6.3 Remarks

Do not confuse PCoC Activities with COM+ Activities. In the context of COM+, respectively Microsoft Transaction Server (MTS), Activities are logical threads running across different machines. They are used for cooperation of COM+, respectively MTS objects. See also [STAL01], Page 291.

3.7 Materials

3.7.1 Overview

In [ZULLIG98], Page 86, *Material* is defined as an object, which can be an element in a container, and on which Tools can operate in a working environment. For example, documents and files are considered as Materials. We agree with this definition, although the term is more limited in this thesis (see below).

According to [ZULLIG98], on Page 89, an object holding Materials is defined as a container, therefore an argument list would be a container.

In the context of this thesis, a Material is a list of arguments passed to Activities or a list of return values provided by them. A Material can also contain other Materials, but this is an exception to the rule. Usually, a Material is just a container and something that a Tool can operate on. An Activity (for example `cut`) of a Tool can only be performed with a specific Material type (the parameter types must match the Activity interface).

3.7.2 Class(es)

The corresponding class is `PCCMaterial`.

3.8 Dispatchers

3.8.1 Overview

One of the most relevant concepts of PCoC are *Dispatchers*. A Dispatcher forwards or delegates requests for the invocation of a specific type of Activity to Activities Providers. Dispatchers are used for component collaboration and within Tasks. A set of directives specifies how to forward requests, e.g., as single or multicasts.

Dispatchers correspond to indirect method references as described in Section 2.4. Figure 2.4-2 illustrates relationships between Dispatchers, Activities, and Activity Sets.

Dispatchers are similar to the Microsoft .NET delegates (type-safe method-pointers). See Section 6.7.5. The main difference to delegates is that Dispatchers can be used for delegation, whereas delegates can only be used for forwarding. With forwarding, the object where a method originally has been invoked (the receiver) is not known to the method holder (where the method is actually executed), and therefore the method holder cannot use it for subsequent method calls.

Dispatchers are automatically generated when an Activity is added to a component, respectively to its associated Activity Set, or acquired from another. Their type safety is ensured at runtime, whereas the type safety of .NET delegates is ensured at compile time.

The set of Dispatchers available in an Activity Set builds a kind of method dictionary. A Dispatcher has a reference to an Activity (if one with the same name exists in the same Activity Set), and a list of references to “acquired” Dispatchers (see Program 2.6-1). When a Dispatcher is invoked (for example, by using its `perform` method), it gathers all Activities by following the directly or indirectly connected Dispatchers and invokes them, depending on specified directives. Directives determine how Activities are invoked. This includes single casts or multicasts. We say that these Activities are in the dispatch or acquisition branch of the Dispatcher.

See Program 2.6-1 and Program 2.11-3.

3.8.2 Class(es)

The corresponding class is `PCCDispatcher`. An instance of this class serves as proxy for an `PCCDispatcherImpl` object, i.e. it forwards method calls to its associated DispatcherImpl object, which does the actual forwarding to Activities. A reference to a Dispatcher can be kept, even if the corresponding DispatcherImpl is not available (any more). As soon as the DispatcherImpl becomes available (after loading or activating the corresponding Activities Provider and its Activities), its Dispatcher becomes functional. See 5.8.3.

3.8.3 Remarks

See Sections 5.4 and 5.6 for more details about Dispatchers.

3.9 Tasks

3.9.1 Overview

A *Task* is a meaningful unit of work, composed of a series of steps that lead to some well-defined goal. In the context of this thesis, it is a complex Activity composed of a series of Dispatchers including directives and arguments, and/or other Tasks. Specified conditions must be met to enable it for invocation.

A simple Task is the opening of a text editor component with a subsequent loading of a text file and the positioning of the caret at the beginning of the corresponding text view.

The term Task corresponds to the term automaton of the WAM-metaphor. [ZULLIG98] defines the term automaton on Page 89 as a machine that is used to perform a specific task. It can autonomously operate on given Materials. It has preconditions that must be met, and if they are, then the automaton performs a series of steps to finish the task. Note that the term automaton is mostly used with another meaning—for state machines (for example, deterministic and non-deterministic finite automaton).

The main view of this thesis is not the automatism, which allows performing a Task, but the Task itself. The ability of autonomously executing a Task is a feature of a Tool or Service Provider in the context of this thesis, rather than a separate operative object (automaton).

3.9.2 Class(es)

The corresponding class is `PCCTask`.

3.9.3 Remarks

See also Section 4.5 for usage of Tasks, and Section 5.7 for more detailed concepts of this design element.

3.10 Case Insensitivity in PCoC

Activity, Dispatcher, and Activity Set names are always handled case insensitively. This is for the convenience of framework users. No one intuitively distinguishes, for example, between `copy`, `Copy`, and `cOpy`, even if most programming languages force developers to do that.

Activity Sets and Activities can be looked up case-insensitively. For example, an Activity created with name “cOpy” can be looked up as “copy”. All Activities with the same case-insensitive name in the system must have the same interface (argument and result types) at runtime. If an Activity with name “Copy” and runtime interface `void Copy(Selection)` is added, and another Activity “cOpy” with interface `void cOpy()`, an error will be thrown.

3.11 Class Hierarchy

Figure 3.11-1 shows the hierarchy of PCoC classes.

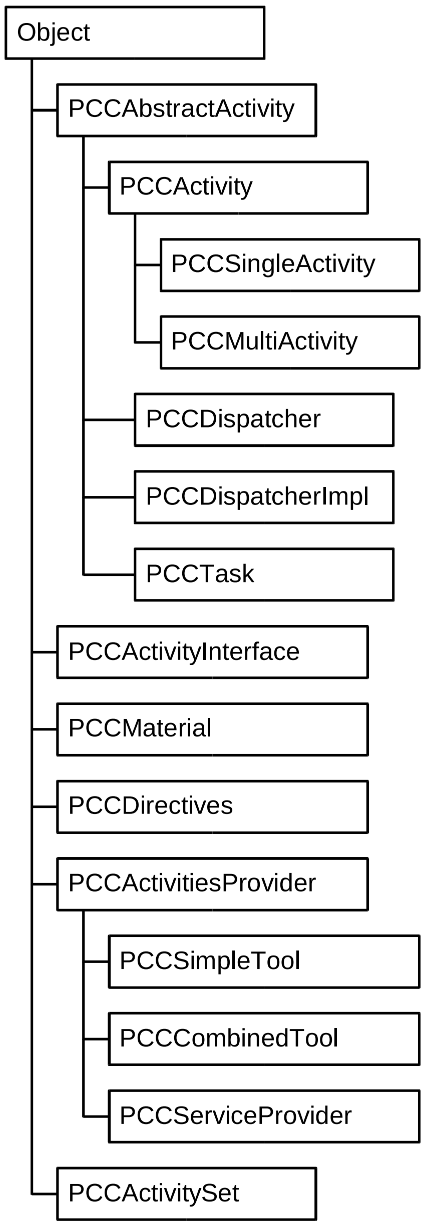


Figure 3.11-1 PCoC class hierarchy

These classes are named as described in Chapter 3, where detailed explanations to these elements are provided.

Note that all PCoC classes have the prefix “PCC”. In most of the figures in this thesis the prefixes have been omitted for simplicity.

4 Using PCoC

This chapter describes the implementation and customization of components based on PCoC.

4.1 Overview

PCoC components can be easily created in a few steps:

- Derive a class from a small base class: For interactive components (components with graphical user interface) derive from `PCCSimpleTool`, and for non-interactive components from `PCCServiceProvider`. See Section 4.2.
- Set up a set of Activities. See Section 4.3.
- Use Dispatchers or Tasks to invoke Activities. See Sections 4.4 and 4.5.

4.2 The First Tool

Program 4.2-1 and Program 4.2-3 show a first Simple Tool. The class `SimpleTool1` derives from `PCCSimpleTool`, overriding `makePresentation` to return a visual representation of itself. All presentation objects of PCoC Tools currently require Java Swing (respectively ET++ in C++). In this case a `JScrollPane` containing a list of four strings is displayed. `makePresentation` is subject to be overridden by each Simple Tool.

Program 4.2-1 SimpleTool1.java (1)

```
package example1;
import pcoc.tools.*;
import javax.swing.*;

/** A Tool providing and displaying a list of strings.
  */
public class SimpleTool1 extends PCCSimpleTool {
   JList fList;

   /** Constructor. Base classes do some initialization.
     */
   public SimpleTool1() {}

   /** Set up the Tool's GUI
     * @return A JComponent representing the GUI of this Tool
     */
   public JComponent makePresentation() {
      String[] data = "Hello World!";
      fList = new JList(data);
      return new JScrollPane(fList);
   }

   /** Set up the Tool's Activity Set which holds and manages an
     * Activities-Provider's Activities.
     */
   protected void setupActivitySet() {
      super.setupActivitySet();
      addActivity(new ClipboardPasteActivity());  // add an Activity and set this
   }                                              // as its provider (fProvider)
...
```

`setupActivitySet` instantiates and adds Activities (operations) to this Activities Provider. In this case we add an instance of `ClipboardPasteActivity`. See Program 4.2-3 for the implementation of this class.

Program 4.2-2 SimpleTool1.java (2)

```
...
   /** Do some extra initialization. doActivate is called after basic
     * initialization of the Activities Provider. */
   public void doActivate() {
      ... // do something, for example synchronize data models between siblings
   }

   /** Do some extra deinitialization. doTerminate is called before basic
     * deinitialization of the Activities Provider. */
   public void doTerminate() { ... // do something }
...
```

`doActivate` and `doTerminate` can be overridden to perform required tasks in the startup and termination phase of a component. Examples for required tasks are data synchronization, sharing of data, adding references to other objects, or removing references.

Program 4.2-3 SimpleTool1.java (2)

```
   /** A concrete Activity which is an implementation of a specific operation-in
     * this case paste from clipboard.
   class ClipboardPasteActivity extends PCCSingleActivity {
      /** Activity constructor. It sets the "dynamic" type of this operation.
        * Type declaration: name "clipboardPaste", category "Edit",
        *   result type (void: ""), parameter types (void: "")
      ClipboardPasteActivity() { super("clipboardPaste", "Edit", "", ""); }

      /** Perform an Activity. It throws an exception if an unexpected error
        * occurs during its execution.
        * @param si Sender and receiver context (involved Activity Sets)
        * @param arg The argument list provided for the execution
        * @return The result(s) of the performed Activity
        */
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial arg)
         throws PCCPerformException {
         String str = Clipboard.getAsString();  // add clipboard content
         if (str != null && !str.equals("")) {  // to list
            ((SimpleTool1)fProvider).addElement(str);
         }
         return null;
      }

      /** Get the current state; is triggered by a direct call of update or
        * indirectly by a call of updateActivity of the provider of the Activity.
        * @return The current state of this Activity
        */
      protected String doGetState() {
         return defaultState(true); // is always "Enabled" in this case
      }
   }
   ... // other Activity classes
} // end of SimpleTool1
```

Program 4.2-3 shows a simple implementation of an Activity. In the constructor, we specify its “dynamic” type (its name, category, and interface). The dynamic type of Activities is described in Section 4.3.4.

`doPerform` implements the core functionality of this Activity defined in the context of the component in Program 4.2-1. It is the most relevant method and must be overridden. It is called indirectly when we call the `perform` method of this Activity.

The `PCCSenderInfo` object holds references to the Activity Set where the Activity has been invoked (the receiver), and to the caller (the sender). The receiver is not necessarily the Activity Set where the Activity finally is executed. The structure usually is filled by PCoC.

`doGetState` returns the current state of this Activity. This method must also be overridden. It can return any state represented as a string. For the most common states, there are predefined methods: `defaultState(bool)`, `disabledState()` (corresponds to `defaultState(false)`), `enabledState()` (corresponds to `defaultState(true)`). In the disabled state an Activity is not executable, i.e., `doPerform` will not be executed when the Activity is invoked. In the enabled or another state, the Activity is executable, i.e., a call of `perform` will lead to an invocation of the `doPerform` method.

Note that in the given example, an Activity is explicitly defined. There are easier ways to create Activities. See the following code fragment.

Program 4.2-4 Semi-automatic generation of Activities

```
public class SimpleTool1 extends PCCSimpleTool {
   protected void setupActivitySet() {
      super.setupActivitySet();
      addActivity("doPaste", "getPasteState");
   }
   void doPaste(String, String) {...}
   String getPasteState() {...}
}
```

With this code fragment, an Activity is implicitly created by the framework, and added to the Activity Set of `SimpleTool1`. The framework uses the Java reflection mechanism to retrieve the methods `doPaste` and `getPasteState` via their names and associate them with the methods `doPerform` and `doGetState` of the newly created Activity. When the Activity `doPaste` is invoked (by calling `perform`), the call is delegated to the corresponding method `doPaste` using reflection. When the state of the Activity is retrieved, the corresponding method `getPasteState` is invoked. This approach is explained in detail in Section 5.9.3.

A Tool’s visual representation is automatically embedded in a window by PCoC’s frame manager. How this is done depends on the application’s configuration. Program 4.2-5 shows, how `SimpleTool1` may be embedded into a Combined Tool (`CombinedTool1`). After the Tool is launched, it is shown as child window of the application main window.

Program 4.2-5 Component configuration

```
<?xml version = "1.0" encoding = "UTF-8"?>
<ResConfig name="Example1" xsi:schemaLocation="www.windriver.com/ResConfig
file:///pwe/rome/config/tools/ResConfig.xsd" xmlns="www.windriver.com/ResConfig"
xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
<RCCombinedTool>
   <name>FrameManager</name>
   <menuBarName>FrameManagerMenuBar</menuBarName>
</RCCombinedTool>
<RCSimpleTool>
   <name>SimpleTool1</name>
   <className>example1.SimpleTool1</className>
</RCSimpleTool>
<RCCombinedTool>
   <name>CombinedTool1</name>
   <windowTitle><dockedTitle>My Combined Tool</dockedTitle></windowTitle>
   <part><simpleToolName>SimpleTool1</simpleToolName></part>
</RCCombinedTool>
...
```

This configuration contains all necessary definitions to set up an application to be able to launch our first Tool. It contains definitions of our Simple Tool, and a Combined Tool containing it.

Definitions for a menu bar (`FrameManagerMenuBar`), a menu (`Example1ToolsMenu`), and a menu entry (`showCombinedTool1`) follow below.

Program 4.2-6 Component configuration: menu items and tasks

```
...
<RCMenuBar>
   <name>FrameManagerMenuBar</name>
   <menuNames>
      <menuName>Example1ToolsMenu</menuName>
   </menuNames>
</RCMenuBar>
<RCMenu>
   <name>Example1ToolsMenu</name>
   <mnemonic>T</mnemonic>
   <shortDescription>Tools</shortDescription>
   <items>
      <item name="showCombinedTool1" type="task"/>
   </items>
</RCMenu>
<RCTask>
   <name>showCombinedTool1</name>
   <mnemonic>1</mnemonic>
   <shortCut>control 1</shortCut>
   <dispatcher>|Application|launchCombinedTool1</dispatcher>
   <RCStates>
      <RCState name="Enabled">
         <shortDescription>Show CombinedTool1</shortDescription>
      </RCState>
      <RCState name="Disabled">
         <shortDescription>Show CombinedTool1</shortDescription>
      </RCState>
   </RCStates>
</RCTask>
</ResConfig>
```

Now, we can use the menu item `showCombinedTool1` to launch our Tool. `showCombinedTool1` causes the Dispatcher `launchCombinedTool1` to be invoked which delegates calls to the associated Activity. The framework generates such an Activity for each Combined Tool. The name of the Activity is that of the Combined Tool prefixed by “`launch`”.

An `RCTask` defines the look and feel and behavior of a Task. The structure defines the mnemonic key `Alt/1`, and the keyboard shortcut `Ctrl/1` for invoking the task. The `RCStates` section describes the look of the menu or toolbar item for the current state of the associated Task.

A Task consist of a Dispatcher and the arguments necessary for the execution of the Task. Each argument can itself be a Dispatcher or Task returning an object as argument for the main Dispatcher in the Task. A Task has also a list of references to other Tasks to be used as a macro (script).

We can simply add a menu item for our Activity `ClipboardPasteActivity` to menu `Example1ToolsMenu` by inserting the following line into its `items` section.

Program 4.2-7 Component configuration: using RCTask definitions

```
<item name="clipboardPaste" type="task"/>
```

In addition, we have to define a `RCTask` and associate it with the menu item. See Program 4.2-8. The corresponding Task object (Activity) will be generated by the framework when the configuration is loaded. When a menu or toolbar item associated with this Task is selected, the method `perform` of the associated Dispatcher `clipboardPaste` in Activity Set `Application` is invoked. The Dispatcher delegates the request to the corresponding Activity of the Activity Set that has the focus. That is, the method `doPerform` of this Activity is finally executed.

Program 4.2-8 Component configuration: RCTask definition

```
<RCTask>
   <name>clipboardPaste</name>
   <shortCut>control v</shortCut>
   <dispatcher>|Application|clipboardPaste</dispatcher>
   <RCStates>
      <RCState name="Enabled">
         <shortDescription>Paste</shortDescription>
      </RCState>
      <RCState name="Disabled">
         <shortDescription>Paste</shortDescription>
      </RCState>
   </RCStates>
</RCTask>
```

See Sections 4.5 and 5.7 for details about Tasks.

4.3 Using Activities

In the sections before we saw how an Activities Provider looks like and how its body can be implemented. Now we concentrate on the implementation of Activities, sending requests for performing Activities, and dynamic coupling and control of Activities Providers.

4.3.1 Activity Class Interface

This section describes the most relevant methods of Activities.

Program 4.3-1 Class interface of PCCAbstractActivity (1)

```
public PCCAbstractActivity(String name, String category,
                           String resultType, String argumentType) {...}

/** Get activity name specified in the constructor.
  */
public String getName() {...}

/** Get activity category specified in the constructor.
  */
public String getCategory() {...}

/** Perform an Activity
  * It throws an exception if an unexpected error occurs during its execution.
  * Calls the method doPerform.
  * @param arg The argument list provided for the execution
  * @return the result(s) of the performed Activity
  */
public PCCMaterial perform(PCCMaterial arg) throws PCCPerformException {...}

/** Returns the current state; is triggered by a direct call of update
  * or indirectly by a call of updateActivity of the provider of the Activity.
  * Calls the method doGetState, if the state was previously
  * invalidated with update.
  * @return The current state of this Activity
  */
public String getState(PCCMaterial arg) {...}

/** Invalidate the state of this Activity. A subsequent call of getState leads
  * to a call of doGetState. If update has not been called, the cached state is
  * returned by getState.
  */
public void update() {...}
```

Program 4.3-1 shows the most relevant methods of a `PCCAbstractActivity`. The class provides default implementations for Activities, Dispatchers, and Tasks. The class `PCCActivity` is the base class for Activities which are subject to be derived in client code.

Actually, we derive from its subclasses—the classes `PCCSingleActivity` and `PCCMultiActivity`. When deriving from these classes, we have to override at least the methods `doPerform` and `doGetState` (which are called from the template methods `perform` and `getState`).

A `PCCSingleActivity` represents a single operation, e.g. `moveFile` to move a file in a file system. `PCCMultiActivity` represents a group of Activities with the same interface, e.g. `showRecentFile` to reopen a recently opened file in a proper Tool. The Activities of this group are accessed with an index parameter (the first argument in `arg`).

Although Activities should rather be used together with Dispatchers (see Section 4.4), they may also be used directly. However, in this case it makes more sense to use ordinary methods instead of Activities.

The following methods are less frequently used, but nevertheless important.

Program 4.3-2 Interface of PCCAbstractActivity (2)

```
/** Get the activity path. It consists of the path of the Activity Set to which
  * this Activity has been added, and the Activity name.
  */
public String getPath() {...}

/** Set context specific data, for example, additional information to the current
  * selection in the GUI. Context data is used, for example, for context sensitive
  * menu entries. This method must be used in doGetState of this Activity
  * @param key The key for a new entry in the context data table of this Activity
  * @param value The value to be associated with the given key
  */
public void setContextData(String key, String value) {...}

/** Get context-specific data, for example, additional information on the current
  * selection in a text editor.
  * @param key The key for a new entry in the context data table of this Activity
  * @param value The value to be associated with the given key
  */
public String getContextData(PCCMaterial arg, String scope, String key) {...}

/** add listener to this Activity
  * @param the listener to add
  */
public void addListener(PCCActivityListener listener) {

/** remove listener from this Activity
  * @param the listener to remove
  */
public void addListener(PCCActivityListener listener) {
```

See Section 4.3.7 for a detailed description of Activity states and context data. See Section 4.3.2 for a description of the listener interface.

Context data denotes a set of strings, representing data in addition to the return value and state of an Activity.

4.3.2 Activity Listener Interface

The following interface is used to send notifications for state changes of Activities, and when Activities are added to or removed from a Dispatcher, Task, etc.

Program 4.3-3 ActivityListener interface (1)

```
/** ActivityListener interface
  * Used for Activities, Dispatchers, and Tasks
  */
public interface PCCActivityListener {
   /** Activity has changed its state. This method is called in a.update().
     * Retrieve the current state by using a.getState()
     * @param a the Activity whose state has changed
     */
   public void activityChanged(PCCAbstractActivity a);

   /** Activity is added to a. a must be Dispatcher, a Task, or a
     * MappedActivity (an Activity associated with a Task)
     * @param a the Activity to which another Activity has been added
     */
   public void activityAdded(PCCAbstractActivity a);

   /** Activity is removed from a. a must be Dispatcher, a Task, or a
     * MappedActivity (an Activity associated with a Task)
     * @param a the Activity from which another Activity has been removed
     */
   public void activityRemoved(PCCAbstractActivity a);
...
```

The following methods are called in the `perform` method of the given Activity `a`.

Program 4.3-4 ActivityListener interface (2)

```
...
   /** Activity is going to be performed. This method is called before a.doPerform
     * @param a the Activity to be performed
     */
   public void beforePerform(PCCAbstractActivity a, PCCMaterial args);

   /** Activity has been performed. This method is called after a.doPerform
     * @param a the Activity which has been performed
     */
   public void afterPerform(PCCAbstractActivity a, PCCMaterial args,
                            PCCMaterial result);
}
```

4.3.3 PCCMaterial: Container for Arguments and Return Values

`PCCMaterial` is a typed container for arguments and return values of Activities. The types of objects contained in a `PCCMaterial` must match the interface specified in the constructor of an Activity.

See also Section 5.5.1 for details about Activity interface specification and checks (type-safety).

The `PCCMaterial` class as illustrated in Program 4.3-5 offers constructors for primitive data types, for an object, and a default constructor. For single values, the constructors should be sufficient. For more complex lists there are methods for adding values to, or setting values in, the container. The interface can be retrieved as string by using the `getInterface` method.

We use an `ArrayList` to hold the objects added to a Material, which is enough for our purpose. We may use `HashMap` instead, for accessing arguments by name, instead of by index. However, this would make the specification of Activity interfaces and arguments via Materials even more complicated, so we decided to use `ArrayList`.

Program 4.3-5 PCCMaterial.java (excerpt)

```
public class PCCMaterial {
   final public static int NPOS = -1;
   ArrayList fArgs;

   /* Default Constructor. Empty list. */
   public PCCMaterial() {...}

   /* Constructors for convenience. Basic data type values are automatically
    * converted to appropriate objects */
   public PCCMaterial(Object value) {...}
   public PCCMaterial(String value, boolean splitString) {...}
   public PCCMaterial(boolean value) {...}
   public PCCMaterial(char value) {...}
   public PCCMaterial(int value) {...}
   public PCCMaterial(double value) {...}

   /* Get a value */
   public Object getObject(int index) {...}
   public boolean booleanValue(int index) {...}
   char charValue(int index) {...}
   public int intValue(int index) {...}
   public double doubleValue(int index) {...}

   /* Add a value */
   public void add(Object element) {...}
   public void add(boolean element) {...}
   public void add(char element) {...}
   public void add(int element) {...}
   public void add(double element) {...}

   /* Set a value */
   public Object set(int index, Object element) {...}
   public Object set(int index, boolean element) {...}
   public Object set(int index, char element) {...}
   public Object set(int index, int element) {...}
   public Object set(int index, double element) {...}

   /* Remove values */
   public void remove(int index) {...}
   public void clear() {...}

   /* Content checks */
   public int size() {...}
   public boolean isNull() {...}
   public String getInterface() {...}  // get types of contained objects as
                                       // interface string "String,Boolean"
   public Class getType(int index) {...}
   public boolean isSubclass(int index, Class cl) {...}
   ...
}
```

Note that primitive data types are converted to instances of the corresponding class. For example, an `int` value `17` is converted to an `Integer` instance with value `17`.

The concrete implementation of this class is not shown, since it is trivial.

4.3.4 Dynamic Activity Type Explained

Activities have a static type which is its class, and a dynamic type which consists of its name, its interface, and category. For example, an Activity might have the name `clipboardPaste`, the interface "`void clipboardPaste(void)`", and might belong to category `Edit` (cf. Program 4.2-3 and Program 4.3-6). The name specified in the constructor ("`clipboardPaste`") may be different to the class name (`ClipboardPasteActivity`).

The result type and the list of parameter types are specified as comma-separated lists of class names. PCoC creates and shares instances of the class `PCCActivityInterface` for the specified type lists. See Section 5.5.1 for more details about `PCCActivityInterface`.

Program 4.3-6 An Activity with no return value and parameters

```
/** Activity constructor. It sets the "dynamic" type of this operation.
  * Type declaration: name "clipboardPaste", category "Edit",
  *   result type (void: ""), parameter types (void: "")
ClipboardPasteActivity() { super("clipboardPaste", "Edit", "", ""); }
```

In this example, the Activity does not need parameters and does not return a result.

An Activity with a more complex interface is illustrated below:

Program 4.3-7 An Activity with a return value and parameters

```
/** Activity constructor. It sets the "dynamic" type of this operation.
  * Type declaration: name "moveFile", category "File", returns a Boolean object,
  *   needs two strings, the original file path and the new one
MoveFileActivity() { super("moveFile", "File", "Boolean", "String,String"); }
```

The Activity takes two `String` objects as parameters and returns a `Boolean` object as result in the template method `perform`, respectively in the method `doPerform` which implements its concrete behavior.

Activity Categories, such as “`File`” in the example above, can be used, for example, for collectively invalidating states, or triggering the invocation of activities. Generally, categories have been introduced only for convenience. A category name is part of the (dynamic) type of an Activity.

An Activity name is bound to a category. For example, it is not allowed to add an Activity `copy` with category `Edit`, and one with category `Clipboard` to the system. This would cause a clash in an Activity Set that acquires two others, one containing a Dispatcher to one of these Activities, and one containing a Dispatcher to the other Activity (see Section 5.4.3). The Dispatcher in the acquiring Activity Set would have to delegate requests to Dispatchers with different dynamic types—in this case different categories. Such a clash of Dispatcher types is always reported by the framework.

Remember, Dispatchers always have the same dynamic type as the associated Activities, respectively its acquired Dispatchers. We say, an Activity Set `A` or Dispatcher `DA` acquires (“dynamically inherits”) a Dispatcher `DB`, when `A` acquires Activity Set `B` containing `DB`, where `DA` and `DB` have the same name.

A clash would also exist if an Activity `copy` with interface `void copy(selection)` and one with `boolean copy(void)` was added.

See Section 5.5.1 for a detailed explanation of the usage and concepts of result and parameter types of Activity.

4.3.5 Creating and Setting Up Activities

In the constructor of an Activity we specify its dynamic type. This includes its name, category, return value and parameter list interfaces. See Section 4.3.4 for a detailed description of the dynamic Activity type.

See also Section 5.5.1 for a description of the Activity Interface class. Instances of this class are used to represent the parameter and result type of an Activity.

Note that Activities should always be an inner class of an Activities Provider to have a clear design.

Program 4.3-8 PCCSingleActivity constructor

```
public class PCCSingleActivity {
   /** Activity constructor. It sets the dynamic type of this Activity and
     * must be called in the constructors of derived classes.
     * @param name The name of the Activity. All Activities with the same
     *    name must have the same dynamic type (category, resultType, argumentType)
     * @param category A virtual category to which this Activity belongs. This
     *    can be used to update the states of all Activities of a category at once,
     *    or to remove them from their Activity Sets.
     * @param resultType Result type. See PCCActivityInterface.
     * @param argumentType Parameter types. See PCCActivityInterface.
   public PCCSingleActivity(String name, String category,
                            String resultType, String argumentType);
}
```

The following methods of `PCCActivitiesProvider` are used for adding and removing Activities. Usually, these methods are used in `setupActivitySet`, which is called after a component has been initialized and while it is being activated. They can also be used during the whole lifetime of a component, i.e. until the component is terminated.

Program 4.3-9 Adding and removing Activities in PCCActivitiesProvider

```
public class PCCActivitiesProvider {
   /** Add an Activity to this Activities Provider, respectively its Activity Set.
     * @param activity an Activity instance
     * @return true, if the Activity was successfully added; false, otherwise
     *  (e.g., if an Activity with the same name already exists in this provider).
   public boolean addActivity(PCCActivity activity) { ... }

   /** Removes an Activity from this Activities Provider, respectively its
     * Activity Set.
     * @param name The name of the Activity to be removed
     * @return the Activity removed from this Activities Provider, or null.
   protected PCCActivity removeActivity(String name) { ... }
}
```

The following example is an excerpt from Program 4.2-1 and Program 4.2-3. `SimpleTool1` implements an Activity `ClipboardPasteActivity`. This Activity is added in `setupActivitySet`. Note that when Activities are not removed explicitly, this is done automatically when the component terminates.

Program 4.3-10 Adding an Activity to a Simple Tool

```
public class SimpleTool1 extends PCCSimpleTool {
   /** Set up the Tool's Activity Set which contains and manages an
     * Activities-Provider's Activities.
     */
   protected void setupActivitySet() {
      super.setupActivitySet();
      addActivity(new ClipboardPasteActivity());
   }

   /** A concrete Activity which is an implementation of a specific operation--in
     * this case paste from clipboard.
   class ClipboardPasteActivity extends PCCSingleActivity {
      /** Activity constructor. In sets the dynamic type of this operation.
        * Type declaration: name "clipboardPaste", category "Edit",
        *   result type (void: ""), parameter types (void: "")
      ClipboardPasteActivity() { super("clipboardPaste", "Edit", "", ""); }
      ...
   }
   ...
}
```

See the following sections and Section 5.5.1 for more details about Activities. Program 4.2-3 has a more detailed implementation of the Activity shown above.

The example below shows how an Activity can be removed.

Program 4.3-11 Removing Activities

```
public class SimpleTool1 extends PCCSimpleTool {
   /** Remove some Activities.
     */
   void removeSomeActivities() {
      removeActivity("clipboardPaste");
      // assuming, the Activity is stored in a member variable
      // clipboardPaste of this component, the following is also possible:
      // removeActivity(clipboardPaste);
   }
   ...
}
```

Activities can also be added to and removed from an Activities Provider different from their current providers (their creators). This feature can be used to implement a component which enhances functionality of another by providing Activities for it.

When such Activities are performed at their actual provider, they have access to the context (environment) in which they have been added.

Program 4.3-12 Adding and removing Activities through another PCCActivitiesProvider

```
public class PCCActivitiesProvider {
   /** Add an Activity to this Activities Provider.
     * @param activity an Activity instance
     * @param toPath the Activity Set where the Activity should be added
     * @return true, if the Activity was successfully added; false, otherwise
     *  (e.g., if an Activity with the same name already exists in this provider).
   public boolean addActivity(String toPath, PCCActivity activity);

   /** Removes an Activity from this Activities Provider.
     * @param name the name of the Activity to be removed
     * @param fromPath the Activity Set where the Activity should be removed
     * @return the Activity removed from this Activities Provider, or null.
   protected PCCActivity removeActivity(String fromPath, String name);
}
```

These methods can be used to add Activities to or remove from an Activities Provider different from its actual provider. The target Activities Provider, respectively its Activity Set is specified as path.

Once added to a different Activities Provider, the new “owner” can be retrieved via the Activity Set currently associated with this Activity.

Program 4.3-13 Retrieving the current Activity owner

```
class MyActivity extends PCCSingleActivity {
   M2Activity() { super("MyActivity", "Methods", "", ""); }
   protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
      // retrieve the Activity Set, respectively the associated provider where
      // this Activity is currently added to. Note that getProvider() delivers
      // the actual creator of the Activity, which may be different to the current
      // "owner" getActivitySet().getProvider().
      PCCActivitiesProvider currentProvider = getActivitySet().getProvider();
      ...                                    // do something
      return null;
   }
}
```

4.3.6 Implementing Activities

4.3.6.1 Subclassing

Program 4.3-14 shows the two methods, `doPerform` and `doGetState`, that must be overridden in concrete Activity classes, for example, those derived from `PCCSingleActivity`.

Program 4.3-14 Subclass interface of PCCActivity

```
/** Perform an Activity.
  * It throws an exception if an unexpected error occurs during its execution.
  * @param si Sender and receiver context (involved Activity Sets)
  * @param arg The argument list provided for the execution
  * @return the result(s) of the performed Activity
  */
protected abstract PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial arg)
  throws PCCPerformException;

/** Returns the current state; is triggered by a direct call of update or
  * indirectly by a call of updateActivity of the provider of the Activity. Is
  * called by getState, if the state was previously invalidated with update.
  * @return The current state of this Activity
  */
protected abstract String doGetState();
```

The following example shows an rough implementation of an Activity class for a file system Service Provider.

Program 4.3-15 An Activity with parameters

```
/** A concrete Activity which is an implementation of a specific operation-in
  * this case moveFile.
class MoveFileActivity extends PCCSingleActivity {
   /** Activity constructor. In sets the dynamic type of this operation.
     * Type declaration: name "moveFile", category "File", returns a Boolean
     *   object, needs two strings, the original file path and the new one
   MoveFileActivity() { super("moveFile", "File", "Boolean", "String,String"); }

   /** Perform an Activity. It throws an exception if an unexpected error
     * occurs during its execution.
     * @param si Sender and receiver context (involved Activity Sets)
     * @param arg The argument list provided for the execution
     * @return the result(s) of the performed Activity
     */
   protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args)
      throws PCCPerformException {
      boolean done = moveFile(
               (String)args.getObject(0),
               (String)args.getObject(1)));
      return new PCCMaterial(done);
   }

   /** Get the current state; is triggered by a direct call of update
     * or indirectly by a call of updateActivity of the
     * provider (component) of the Activity.
     * @return The current state of this Activity
     */
   protected String doGetState() {
      return defaultState(true); // is always "Enabled" in this case
   }
   ...
}
```

Get more about concepts and interfaces in the sections below, and in Section 5.5.

4.3.6.2 Performing Activities

The following example shows the use of the most relevant methods of Activities.

Program 4.3-16 Invoking Activity methods

```
PCCMaterial args = new PCCMaterial();
args.add("fromfile.txt");
args.add("tofile.txt");

PCCActivity activity = new MoveFileActivity();
String currentState = activity.getState(args);
boolean readyToPerform =
   activity.canPerform(args); // is equal to getState(...) != disabledState()
Boolean result = (Boolean)activity.perform(args);
```

We create an Activity instance and provide arguments for the invocation of Activity methods. `getState` returns the current state of the `MoveFileActivity` instance. `canPerform` is used to check if an Activity is executable for a subsequent invocation with `perform`.

Note that `perform` and `getState` are template methods. The concrete behavior is defined in derived classes by overriding the methods `doPerform` and `doGetState`. See also Section 4.3.1.

4.3.7 State Handling of Activities

The state of an Activity can be changed at any time. It represents a special attribute of an Activity which can be used to determine whether an Activity is executable at a specific point in time, or not.

The state can be any string. There are some predefined states:

- "`Enabled`": Can be used to specify that the corresponding Activity can currently be performed.
- "`Disabled`": A specific state in which the Activity cannot be performed. The framework ignores requests for performing Activities which are in this state.
- "`Default`": This state is actually only available for the configuration. Definitions for this state in `RCTasks` are used if they should be shared by concrete states. For example, if the description of a menu item should always be "`Browse Selection`" no matter which state its associated Activity has, the slot description can be defined for state "`Default`". If there is no definition for a specific state, "`Default`" is used instead.

Program 4.3-17 State definition in the configuration

```
<RCTask>
   <name>Copy</name>
   <dispatcher directives="First">|Application|Copy</dispatcher>

   <!-- Default state. Definitions can be overridden by concrete states-->
   <RCState name="Default">
      <description>Copy Selection<description>
      <icon>displaySelectionIcon</icon>
   </RCState>
</RCTask>
```

This excerpt of the configuration of a menu item shows how the definitions of the `Default` state is used for any state of the Dispatcher `copy` associated with `RCTask` "`Copy`". The Dispatcher represents all `copy` Activities of this application (path "`Application`").

See Section 5.7 for a more detailed description of Tasks, and Section 5.4 for details about Dispatchers.

Program 4.3-18 Retrieving an Activity state

```
public class GetSelectionActivity extends PCCSingleActivity {
   public GetSelectionActivity() {
      super("getSelection", "SelectionCategory", "Selection", "");
   }
   protected String doGetState() {
      return defaultState(hasSelection()); // returns "Enabled" or "Disabled"
   }
   ...
}
```

The state of an Activity can be invalidated by calling `update` for this Activity or by calling `updateActivity` in the Activities Provider where it was added.

Program 4.3-19 Invalidating an Activity state

```
PCCActivity activity = new GetSelectionActivity();
addActivity(activity);
activity.update();
updateActivity("getSelection");
```

The last two lines have the same effect. In fact, the framework retrieves in `updateActivity` the given Activity and calls its `update` method. Listeners of `activity`, if there are any, are notified in `update`.

Program 4.3-20 Validating an Activity state in a listener

```
public class MyListener implements PCCActivityListener {
   public void activityChanged(PCCAbstractActivity a) {
      String currentState = activity.getState();
   }
   public void activityAdded(PCCAbstractActivity a) {}
   public void activityRemoved(PCCAbstractActivity a) {}
   ...
}
```

See Section 4.3.2 for a description of the `PCCActivityListener` class.

The next time when we call `getState`, the state is validated (in `doGetState`). Once the state is validated, it is cached and used for subsequent calls of `getState`, until `update` is called again.

Program 4.3-21 Invalidating an Activity state

```
public class PCCActivity extends PCCAbstractActivity {
   ...
   public String getState(PCCMaterial arg) {
      if(fState == null) { fState = doGetState(arg); }
      return fState;
   }
   protected abstract String doGetState(PCCMaterial arg) {}
   /** Update / invalidate the state of this Activity.
     */
   public void update() {
      resetState();
      super.update();
   }
   void resetState() { fState = null; }
}
```

This code fragment illustrates the implementation of the methods `update` and `getState`. See also Section 4.4.2.

4.3.8 Fetching Context Data of Activities

The following code fragment describes how to set context data for an Activity.

Program 4.3-22 Setting context data

```
public class GetSelectionActivity extends PCCSingleActivity {
   public GetSelectionActivity() {
      super("getSelection", "SelectionCategory", "Selection", "");
   }
   ...
   protected String doGetState() {
      setContextData(/*key*/ "selection", /*value*/ getSelectionAsString());
      setContextData(/*key*/ "length", /*value*/ getSelectionAsString().length());
      return defaultState(hasSelection()) ;
   }
}
```

Depending on whether there is a selection or not, the state “`Enabled`” or “`Disabled`” is returned. The selection as string is associated with the key “`selection`” and the length of the string selection is associated with key “`length`” in this case. The context data is valid until the next state change. The context data can be explicitly accessed via the Activity’s interface or that of its Dispatchers. The implementation of `setContextData` is shown below.

Program 4.3-23 Setting context data

```
public class PCCActivity extends PCCAbstractActivity {
...
   public void setContextData(String key, String value) {
      if (fContextData == null) { fContextData = new HashMap(1); }
      fContextData.put(key.toLowerCase(), value);
   }
   public String getContextData() {
      String contextData = "";
      if (fContextData != null) {
         contextData = (String) fContextData.get(key.toLowerCase());
      }
      if (contextData == null) { contextData = ""; }
      return contextData;
   }
   HashMap fContextData;
}
```

Context data can be directly fetched from the Activity providing it, as illustrated in the following example.

Program 4.3-24 Fetching context data

```
PCCMaterial args = new PCCMaterial();
PCCActivity activity = new GetSelectionActivity();
String data = activity.getContextData(args, "getSelection", "selection")
```

A context data entry can be referred in a context sensitive menu entry as in Program 4.3-25 (excerpt from an `RCTask` definition).

Context data references are enclosed by curly braces. “`getFileSelection`” is the Activity, respectively the Dispatcher of the Activity, specified as argument for the actual `BrowseFile` Activity in the `RCTask`. “`selection`” is the name which refers to a string defined as context data by the Activity (the key for the context data table). `<30` means that only the left 30 characters of the string representation are shown.

Context data table entries must be set when the state of an Activity changes, i.e., in `doGetState`. See Program 4.3-22.

Program 4.3-25 Using context data in context sensitive menus

```
<RCTask>
   <name>BrowseFile</name>
   <dispatcher directives="First">|Application|BrowseFile</dispatcher>
   <args><dispatcher>getFileSelection</dispatcher></args>
   <RCStates>
      <RCState name="Default">
         <description>Browse File<description>
         <icon>browseFileSelectionIcon</icon>
      </RCState>
      <RCState name="Enabled">
         <description>Browse '{getFileSelection:selection,<30}'<description>
      </RCState>
      <RCState name="Disabled">
      </RCState>
      <RCState name="Active">
         <description>Browse '{getFileSelection:selection,<30}'<description>
         <checkMarked>true</checkMarked>
      </RCState>
   <RCStates>
</RCTask>
```

The basic algorithm for retrieving context data (used with `getContextData`) is quite simple:

Program 4.3-26 Expanding placeholders with context data

```
abstract public class PCCAbstractActivity {
/** Expand a text with placeholders
  * @param orgText A text containing placeholders
  * @param map A map containing data to be filled into the placeholders
  * @return the expanded text
  */
String expand(String orgText) {    // pseudo code
   int lasttoken = 0;
   int token = 0;
   int septoken = 0;
   int maxlen = 0;
   String expTxt;
   // find placeholders in the text
   while ((token = orgTxt.indexOf("{", lasttoken)) < orgTxt.length()) {
      expTxt.append(orgTxt.substring(lasttoken, token-lasttoken));
      lasttoken = orgTxt.indexOf("}", token);
      if  (lasttoken < 0) { lasttoken = orgTxt.length(); }
      // find length specification, if there is any
      septoken = orgTxt.indexOf(",", token);
      if  (septoken < 0 || septoken > lasttoken) {
         septoken = lasttoken;
         maxlen = 0;
      }
      else { maxlen = Integer.valueOf(orgTxt.substring(septoken,
                                                       lasttoken-septoken-1));
      }
      String id = orgTxt.substring(token+1,lasttoken-token-2);
      String data = expTxt.append(getContextData(id)); // get context data
      if (maxlen > 0 && maxlen < data.length()) {  // cut to specified length
         data.substring(0, maxlen-1);
      }
      expTxt.append(data);                         // append expanded placeholder
      lasttoken++;
   }
   // append the rest of the string
   expTxt.append(orgTxt.substring(lasttoken, orgTxt.length()-lasttoken-1))
   return expTxt;
}
...
}
```

Read more about this topic in Section 4.4.3.

4.4 Using Dispatchers

Actually, Activities are never invoked directly, but rather via Dispatchers.

Dispatchers have been introduced in Section 3.8. You can find implementation details in Section 5.6.

4.4.1 Invoking Activities

The most relevant Dispatcher methods are `getState`, `canPerform`, and `perform`. The methods can either be directly called on Dispatchers (Program 4.4-1), or through convenience methods with the same names provided by the class `PCCActivitiesProvider` (Program 4.4-3) or `PCCActivitySet`.

Let us assume, we have an Activity Set with path "`|Application|AllServices|`" that acquires another Activity Set "`|Application|FileSystemService|`" (we also say, it acquires its Dispatchers, respectively its Activities). `FileSystemService` provides two Activities, `MoveFile` and `CopyFile`, and acquires another Activity Set. Requests on Dispatchers in `AllServices` (for simplicity we only use the Activity Set name, since the full path is not relevant here) are delegated to Dispatchers, respectively their associated Activities, in either `FileSystemService` or the indirectly acquired Activity Set.

A state change through method `update` in the Activity `MoveFile` is propagated back the delegation chain. More precisely, Dispatchers listen to state changes of their acquired Dispatchers or their associated Activities. In the example below, first, the `MoveFile` Dispatcher in the same Activity Set is notified and then the `MoveFile` Dispatcher in `AllServices`. Subsequent calls of method `getState` on one of the Dispatchers lead to an update of the state in the `MoveFile` Activity. See also Section 4.3.7.

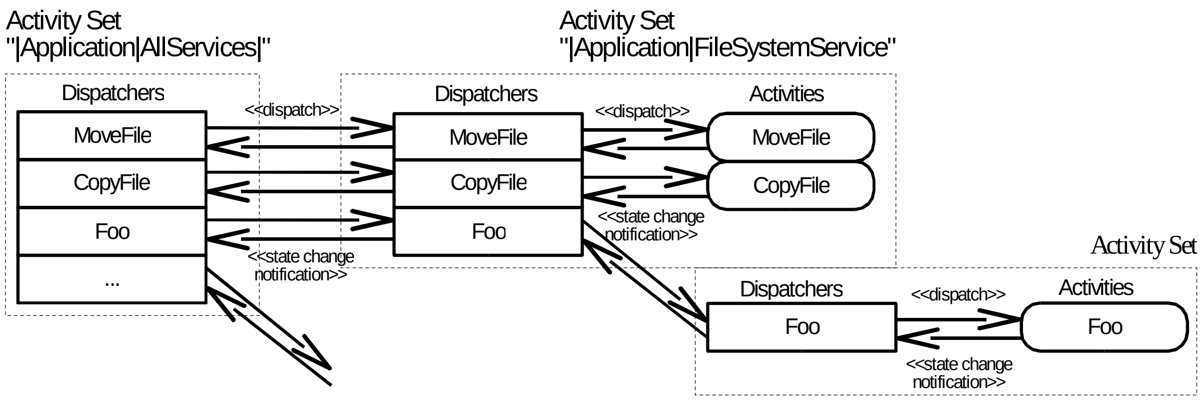


Figure 4.4-1 Invocation of Dispatchers

We assume, that the signature of `MoveFile` is `Boolean MoveFile(String fromFile,String toFile)`, that of `CopyFile` is `Boolean CopyFile(String fromFile,String toFile)`.

The following code fragment shows the use of Dispatchers, using the example above.

Program 4.4-1 Directly calling Dispatcher methods

```
public class AllServices extends PCCServiceProvider {
...
public void foo() {
  PCCMaterial args;
  args.add("fromfile.txt");
  args.add("tofile.txt");
  // Note: The methods below also take an Object array for convenience:
  // Object[] args = {"fromfile.txt", "tofile.txt"};

  PCCDispatcher dispatcher = getDispatcher("|Application|AllServices|MoveFile");
  String currentState = dispatcher.getState(
     args, PCCDirectives.first());

  boolean readyToPerform = dispatcher.canPerform(
     args, PCCDirectives.first());

  Boolean result = (Boolean)dispatcher.perform(
     args, PCCDirectives.first());

  String contextData = dispatcher.getContextData(           // return context data
     args, "MoveFile", "selection", PCCDirectives.first()); // set in doGetState
}
```

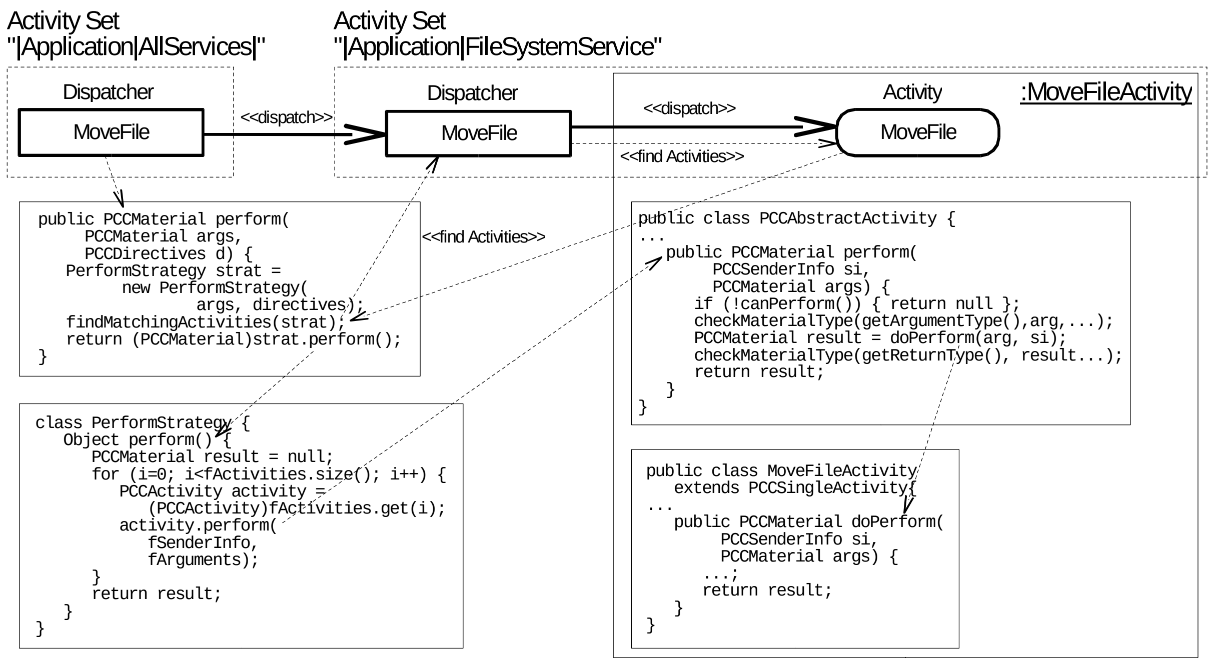
In this case, Dispatchers are retrieved and stored for later use. Note that `getDispatcher(...)` actually calls `getActivitySet().getDispatcher(...)`. Compare the code fragment to Program 4.4-3.

The parameter specified with `getDispatcher` is the path and name of the Dispatcher that is used to delegate the request for performing Activities with the same name. In this case, the execution of `MoveFile` is requested. We use the Dispatcher of Activity Set "`|Application|AllServices`". With the request we deliver an argument list, `args`. The last parameter specifies directives for forwarding the request (e.g. `PCCDirectives.first()`, `PCCDirectives.broadcast()`, etc.) Directives specify, how a Dispatcher delegates requests to Dispatchers of acquired Activity Sets. More precisely, they specify how a Dispatcher gathers Activities with the same name ("`MoveFile`") of directly and indirectly acquired Activity Sets. The specified method, for example, `perform` or `getState`, is invoked on all resulting Activities. See Section 5.6.2 for the implementation of the corresponding method `findAllActivities`. See also the direct acquisition mechanism explained in Section 5.4.

In this example, requests (`perform`, `getState`) are delegated to a `MoveFile` Activity of an Activity Set acquired by `AllServices`. Due to directive `PCCDirectives.first()`, requests for the invocation of `MoveFile` are delegated to the Activity Set with the highest priority providing an Activity with the given name (see direct acquisition in Section 5.4). In our example, the Activity of `FileSystemService` is used. If there was another Activity Set acquired by `AllServices`, that also provides a `MoveFile` Activity, the Activity Set that most recently got the focus would be used. If directive `PCCDirectives.broadcast()` was specified, requests would be delegated to both Activity Sets providing the Activity.

`canPerform` checks if the specified Dispatcher, respectively the corresponding Activity is executable. It has the same effect as `getState(...) != disabledState(...)`.

The following picture illustrates which methods are involved for an Activity request through a Dispatcher.



```
public PCCMaterial perform(
      PCCMaterial args,
      PCCDirectives d) {
   PerformStrategy strat =
         new PerformStrategy(
                 args, directives);
   findMatchingActivities(strat);
   return (PCCMaterial)strat.perform();
}
```

```
class PerformStrategy {
   Object perform() {
      PCCMaterial result = null;
      for (i=0; i<fActivities.size(); i++) {
         PCCActivity activity =
            (PCCActivity)fActivities.get(i);
         activity.perform(
            fSenderInfo,
            fArguments);
      }
      return result;
   }
}
```

```
public class PCCAbstractActivity {
...
   public PCCMaterial perform(
        PCCSenderInfo si,
        PCCMaterial args) {
      if (!canPerform()) { return null };
      checkMaterialType(getArgumentType(),arg,...);
      PCCMaterial result = doPerform(arg, si);
      checkMaterialType(getReturnType(), result...);
      return result;
   }
}
```

```
public class MoveFileActivity
   extends PCCSingleActivity{
...
   public PCCMaterial doPerform(
        PCCSenderInfo si,
        PCCMaterial args) {
      ...;
      return result;
   }
}
```

Figure 4.4-2 Dispatchers delegating requests

The Activity `MoveFile` is an instance of the class `MoveFileActivity` and has been added to the Activity Set `FileSystemService`. The concrete behavior is implemented in the method `doPerform`, respectively `doGetState`. Note that we use the implementation from Program 4.3-15.

When `perform` is called on Dispatcher `MoveFile` in `AllServices` (path "`|Application|AllServices|MoveFile`"), we gather all Activities in the acquisition relationship using `findActivities` (see Section 5.6.2 for a detailed description). In this case, we get a list (as a member of the strategy object `strat`—an instance of class `PerformStrategy`) with only one element—the `MoveFile` Activity of `FileSystemService`. In the `perform` method of `strat`, we invoke the template method `perform` of each Activity in the list. Finally, `doPerform` is called. Note that there are different strategy classes for different requests (`perform`, `getState`, etc.).

We only describe the invocation and delegation of the `perform` request on a `MoveFile` Dispatcher in the picture above. Other requests, such as `getState`, `canPerform`, etc., maybe using other Dispatchers, are handled the same way.

Here an overview of requests that are supported by Activities:

Program 4.4-2 Activity methods (PCCAbstractActivity)

```
public boolean canPerform(PCCMaterial arg);
public String getState(PCCMaterial arg);
public PCCMaterial perform(PCCSenderInfo si, PCCMaterial arg)
  throws PCCPerformException;
public String getContextData(PCCMaterial arg, String scope, String key);
```

Note that the corresponding methods of Dispatchers have an additional parameter holding dispatch directives.

`PCCActivitiesProvider` provides methods that simplify the use of Dispatchers, if they need not be stored for later use. The following code fragment is semantically equal to Program 4.4-1.

Program 4.4-3 Calling Dispatcher methods via PCCActivitiesProvider

```
public class AllServices extends PCCActivitiesProvider {
...
  public void foo() {
    PCCMaterial args = new PCCMaterial();
    args.add("fromfile.txt");
    args.add("tofile.txt");
    // Note: The methods below can also take an Object array for convenience:
    // Object[] args {"fromfile.txt", "tofile.txt"};

    String currentState = getState(
       "|Application|AllServices|MoveFile",
       args, PCCDirectives.first());    // directive first is the
                                        // default and can be omitted

     // canPerform is equal to getState(...) != disabledState()
     boolean readyToPerform = canPerform(
        "|Application|AllServices|MoveFile",
        args, PCCDirectives.first());

     Boolean result = (Boolean)perform(
        "|Application|AllServices|MoveFile",
        args, PCCDirectives.first());

     String contextData = getContextData( // context data is set in doGetState
        "|Application|AllServices|MoveFile",
        args, "MoveFile", "selection", PCCDirectives.first());
  }
}
```

Using these convenience methods, you save to use explicit references to Dispatchers. For example, instead of `getDispatcher(...).perform(...)`, we can simply write `perform(...)`.

4.4.2 State Handling of Dispatchers

See Section 4.3.7 for an introduction in state handling for Activities.

The following code fragment retrieves the current state of an Activity via a Dispatcher.

Program 4.4-4 Getting the state of an Activity via a Dispatcher

```
public class AllServices extends PCCActivitiesProvider {
...
   public void foo() {
      PCCDispatcher dispatcher = getDispatcher("|Application|getSelection");
      String currentState = dispatcher.getState(args, PCCDirectives.first());
   }
}
```

If the state of the Activity has been invalidated, a subsequent call of `getState` leads to a call of method `doGetState`. If the state has not been invalidated, the cached state is used instead (the most recent state).

4.4.3 Fetching Context Data From Dispatchers

See Section 4.3.8 for an introduction into context data retrieval for Activities.

Program 4.4-5 Fetching context data via a Dispatcher

```
public class AllServices extends PCCActivitiesProvider {
...
   public void foo() {
      PCCDispatcher dispatcher = getDispatcher("|Application|getSelection");
      String data = dispatcher.getContextData(null, null, "selection",
         PCCDirectives.first());
      data = getContextData("|Application|getSelection", null, null, "selection",
         PCCDirectives.first());
   }
}
```

In this example, we use Dispatchers to retrieve the context data entry with name "`selection`" of an associated Activity. The last two statements have the same effect.

Program 4.4-6 Fetching context data via a Dispatcher

```
final class PCCDispatcher extends PCCAbstractActivity
                          implements PCCActivityListener {
...
   /** Returns context data of an activity/activities.
     * @param args the arguments for the activities which provide context data
     * @param scope reserved. The scope is mainly used by PCCTask.
     * @param key the name of a data entry in the associated Activities.
     * @param directives the directives for searching activities
     * @return the retrieved context data.
     */
   public String getContextData(PCCMaterial args, String scope, String key,
                                PCCDirectives directives) {
      if (directives.isBroadcast()) {
         GetContextDataStrategy strat = new GetContextDataStrategy(args,
                                                 directives, scope, key);
         findMatchingActivities(strat);
         return (String) strat.perform();
      }
      else {
         PCCActivity activity = findMatchingActivity(args, directives);
         if (activity != null) {
            return activity.getContextData(arg, scope, key);
         }
         return "";
      }
   }
}
```

This code fragment describes how a Dispatcher retrieves context data from associated Activities.

A sample configuration for retrieving context data from a Dispatcher for use with a context sensitive menu item is shown in Program 4.3-25.

4.5 Using Tasks

4.5.1 Overview

In the context of this thesis a Task is a complex Activity which is composed of conditions that must be met for execution, a Dispatcher and its argument list, and/or a list of references to other Tasks. For example, a simple Task could be the launching of a text editor component with a subsequent loading of a text file and the positioning of the caret at the beginning of the corresponding text view.

Tasks are used mainly for menu entries, toolbar buttons, scripts, etc.

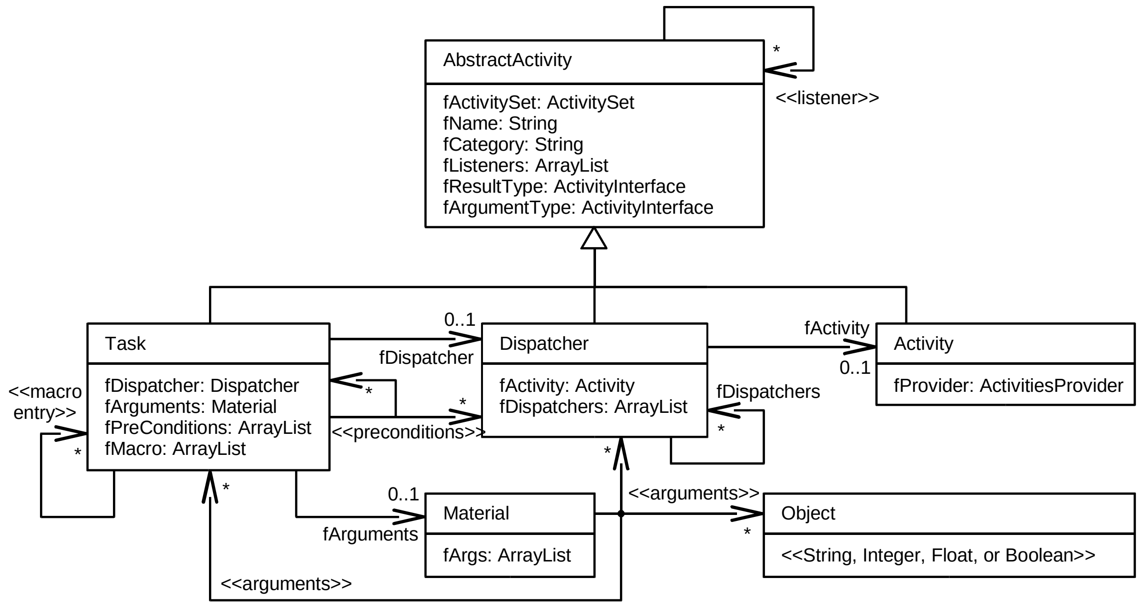


Figure 4.5-1 Task relationships

A Task has usually a reference to a `PCCDispatcher` instance, and a list of arguments represented as instance of the `PCCMaterial` class. The argument list can contain instances of primitive types (`String`, `Integer`, `Float`, `Boolean`, `ArrayList`) as well as of more complex types. It can also contain references to Dispatchers and Tasks which are performed and replaced by their return value, when the container Task is performed.

A Task acts as macro, if a list of references to other Tasks is provided via configuration. See Section 4.5.3.

The list of preconditions can contain references to Dispatchers and Tasks, which must be executable (`canPerform() == true`) in order to be able to perform the Task.

The following section gives an introduction into these features. Section 5.7 has more details.

4.5.2 Simple Tasks

Program 4.5-1 shows a Task configuration in XML.

Program 4.5-1 A Task definition in a configuration

```
<RCTask>
   <name>displaySelectionTask</name>
   <dispatcher directives="First">|Application|displaySelection</dispatcher>
   <args>
      <dispatcher>getSelection</dispatcher>
      <integer>2</integer>
   </args>
   <!-- Default state. Definitions can be overridden for concrete states>
   <RCState name="Default">
      <description>Display Selection<description>
      <icon>displaySelectionIcon</icon>
   </RCState>
   <RCState name="Enabled">
      <description>Display '{getSelection:selection,<30}'<description>
   </RCState>
   <RCState name="Disabled">
   </RCState>
   <RCState name="Active">
      <description>Display '{getSelection:selection,<30}'<description>
      <checkMarked>true</checkMarked>
   </RCState>
</RCTask>
```

Usually, Tasks are invoked through user interaction (selecting a menu entry, etc.) They can also be used explicitly in source code (see Program 4.5-2 and Program 4.5-3).

Program 4.5-2 Performing a Task

```
public class MyActivitiesProvider extends PCCServiceProvider {
...
   public void foo() {
      // Get or create a Task; calls getActivitySet().getOrCreateTask(...);
      PCCTask task = getOrCreateTask("displaySelectionTask");
      if (task.canPerform()) { // if state != disabledState(), and all Dispatchers
         task.perform();       // are available and executable, and all arguments
      }                        // are provided via the configuration, and the
                               // types of the passed arguments match the dynamic
   }                           // type of the Dispatcher
}
```

The Task configuration associated with the given name (“`displaySelectionTask`”) is used to initialize an instance of the class `PCCTask` (see Program 4.5-2). See Section 5.7 for implementation details about this class. After the Task is created, we invoke it using `perform`.

The method `foo2` in the following code fragment is semantically equivalent to method `foo` of Program 4.5-2.

Program 4.5-3 Performing a Task

```
public class MyActivitiesProvider extends PCCServiceProvider {
...
   public void foo2() {
      performTask("displaySelectionTask");
   }
}
```

The Task can also be put into a queue using `performTaskLater`, where it is performed when all other Tasks previously added to the same queue are finished.

Program 4.5-4 Adding a Task definition to a menu

```
<RCMenu>
   <name>ExampleTasksMenu</name>
   <mnemonic>E</mnemonic>
   <shortDescription>ExampleTasks</shortDescription>
   <items>
      <item name="displaySelectionTask" type="task"/>
   </items>
</RCMenu>
```

This example shows the same Task used with a menu item in the menu `ExampleTasksMenu`. Selecting this menu item causes the framework to call `performTask("displaySelectionTask")`.

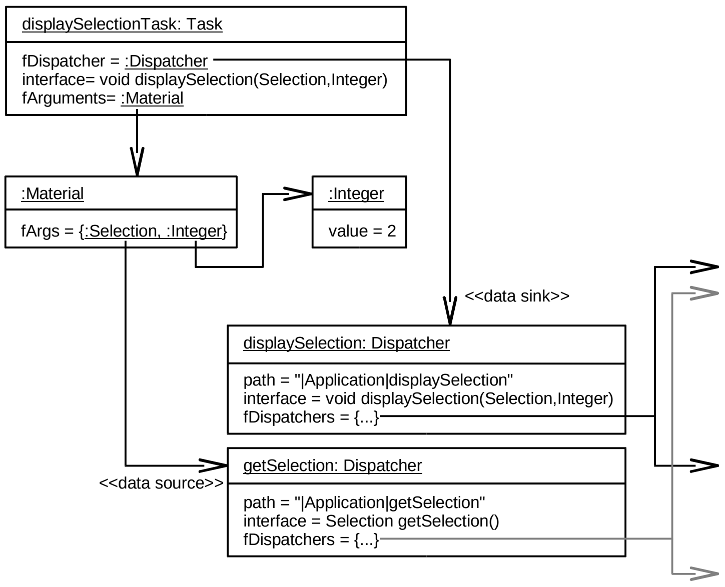

displaySelectionTask: Task
fDispatcher = :Dispatcher
interface= void displaySelection(Selection,Integer)
fArguments= :Material
:Material
fArgs = {:Selection, :Integer}
:Integer
value = 2
<<data sink>>
displaySelection: Dispatcher
path = "|Application|displaySelection"
interface = void displaySelection(Selection,Integer)
fDispatchers = {...}
<<data source>>
getSelection: Dispatcher
path = "|Application|getSelection"
interface = Selection getSelection()
fDispatchers = {...}

Figure 4.5-2 Assembled Task

The Task of this example is associated with a Dispatcher `displaySelection` in Activity Set "`|Application`". This Dispatcher, and therefore also the Task, expects a `Selection` and an `Integer` object as arguments.

When the Task is performed, it first uses the Dispatcher `getSelection` to invoke the Activity `getSelection`, which returns a `Selection` object. Then the Task uses the Dispatcher `displaySelection` of Activity Set `Application` to request the invocation of method `perform` on an "acquired" Activity. It passes the previously retrieved selection object and an `Integer` object with value `2` as arguments. The arguments are passed via an instance of class `PCCMaterial` by the framework.

Since the `getSelection` Dispatcher does not have a path in this example, the Dispatcher of the Activity Set where the current Task object has been created, is used. This means that a component that creates and performs a corresponding Task object, and which provides a `getSelection` Activity, would pass its own selection object to this Task. The target component (the component providing the `displaySelection` Activity) is determined through the acquisition graph beginning with the `Application` Activity Set. A target component, respectively its Activity Set is found if it has directly or indirectly been acquired by the `Application` Activity Set and provides the Activity `displaySelection`, and the acquisition branch to it has a higher priority than those of other Activity Sets possibly providing this Activity.

If no Activities `displaySelection` and `getSelection` are created and acquired, yet, the whole Task is not executable. When the Dispatchers `displaySelection` and `getSelection` are available, respectively corresponding Activities, and `displaySelection` has the state "Enabled", but `getSelection` has the state "Disabled", then the Task `displaySelectionTask` is valid, but cannot be performed due to the "Disabled" state of `getSelection`.

A list of preconditions and the dispatcher directive also regulate whether a Task can be performed. A condition can itself be a Task or a Dispatcher that must be available and executable.

Program 4.5-5 Defining a Task via configuration

```
<RCTask>
   <name>displaySelectionTask</name>
   ...
   <preconditions>
      <dispatcher>|Application|foo</dispatcher>
      <task>myCondition</task>
   </preconditions>
</RCTask>

<RCTask>
   <name>myCondition</name>
   ...
</RCTask>
```

In this case, the Dispatcher `foo` and the Task `myCondition` must be executable (`canPerform` must return `true` for each condition). Only if this is true, the Task `displaySelectionTask` can be performed.

See also Section 5.7 for implementation details of Tasks.

4.5.3 Macros

Tasks can be used as macros. Therefore they provide a slot `do` which can contain a list of references to other Task definitions or inlined Task definitions. When a macro-Task is performed, the Tasks referenced in this list are performed in the given order.

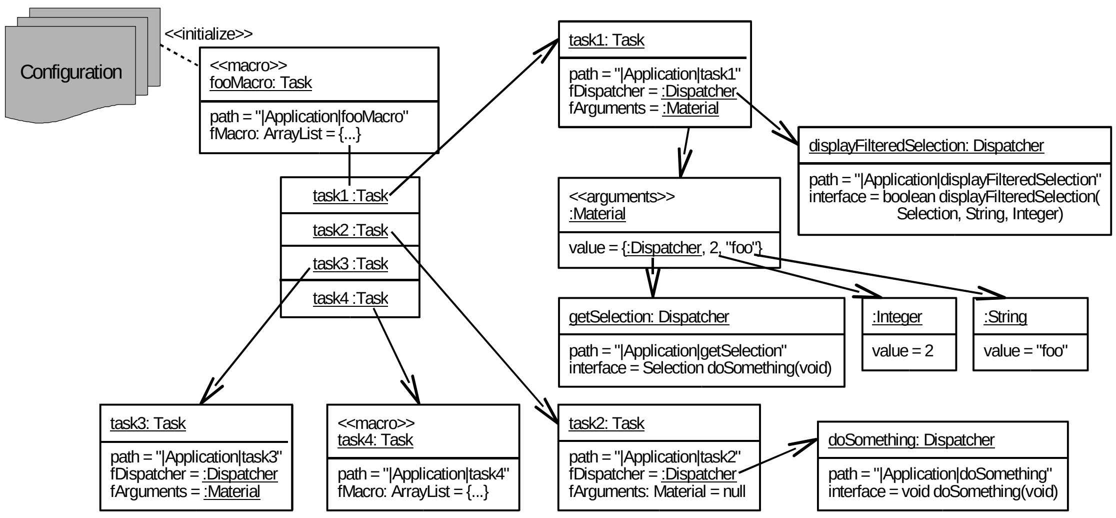

Configuration
<<initialize>>
<<macro>> fooMacro: Task
path = "|Application|fooMacro"
fMacro: ArrayList = {...}
task1 :Task
task2 :Task
task3 :Task
task4 :Task
task1: Task
path = "|Application|task1"
fDispatcher = :Dispatcher
fArguments = :Material
displayFilteredSelection: Dispatcher
path = "|Application|displayFilteredSelection"
interface = boolean displayFilteredSelection(Selection, String, Integer)
<<arguments>> :Material
value = {:Dispatcher, 2, "foo"}
getSelection: Dispatcher
path = "|Application|getSelection"
interface = Selection doSomething(void)
:Integer
value = 2
:String
value = "foo"
task3: Task
path = "|Application|task3"
fDispatcher = :Dispatcher
fArguments = :Material
<<macro>> task4: Task
path = "|Application|task4"
fMacro: ArrayList = {...}
task2: Task
path = "|Application|task2"
fDispatcher = :Dispatcher
fArguments: Material = null
doSomething: Dispatcher
path = "|Application|doSomething"
interface = void doSomething(void)

Figure 4.5-3 Assembling a Task

Figure 4.5-3 shows a macro `fooMacro`. The configuration may look as in Program 4.5-6.

The first two entries in the `do`-list, `task1` and `task2`, are inlined Tasks. The other two, `task3` and `task4`, are references to other Task definitions.

Note that definitions can be distributed over several files.

`task1` is associated with a Dispatcher `displayFilteredSelection` in Activity Set `Application`. A Dispatcher (`getSelection`), a string (“`foo`”) and an integer (`2`) are passed as arguments to `task1`, respectively its associated Dispatcher `displayFilteredSelection`.

`task2` is associated with Dispatcher `doSomething` and has no arguments. Since no Activity Set path is specified for `doSomething`, the Activity Set of the Task is used (`Application`). We assume that `task3` is no macro, and `task4` is a macro-Task.

Program 4.5-6 Defining a macro (script)

```
<RCTask>
   <name>fooMacro</name>
   <do>
      <RCTask>
         <name>task1</name>
         <dispatcher directives="First">
            |Application|displayFilteredSelection</dispatcher>
         <args>
            <dispatcher>|Application|getSelection</dispatcher>
            <string>foo</string>
            <integer>2</integer>
         </args>
      </RCTask>
      <RCTask>
         <name>task2</name>
         <dispatcher directives="First">doSomething</dispatcher>
      </RCTask>
      <task>task3</task>
      <task>task4</task>
   </do>
</RCTask>

<RCTask>
   <name>task3</name>
   ...
</RCTask>

<RCTask>
   <name>task4</name>
   ...
   <do>
      ...
   </do>
</RCTask>
```

When `fooMacro` is performed, `task1` is performed first. On successful completion of `task1`, `task2` is performed, etc. If a Task of the macro cannot be completed successfully, the execution of the macro is stopped per default. That is, the subsequent Tasks in the `do`-list are not performed.

You can change this behavior by setting the `stopOnError`-flag correspondingly.

Program 4.5-7 Defining a macro (script)

```
<RCTask>
   <name>fooMacro</name>
   <stopOnError>0</stopOnError>
   <do>
      <RCTask>
         <name>task1</name>
...
```

Note, that, when a macro-Task is invoked, all paths of referenced Dispatchers are temporarily replaced by the paths of concrete Activities right before it is actually executed. This prevents unexpected behavior, if the acquisition priorities (the ranking) of Activity Sets change during the execution of the macro, for example, if a new component is loaded or removed during the execution, or the ranking is changed explicitly.

Let us assume that the Activity Set with path "`|Application|MyView`" provides an Activity `displayFilteredSelection`. The Activity Set is acquired by `Application` and has the highest ranking in the list of parents in `Application`. In this case, the Dispatcher path "`|Application|displayFilteredSelection`" will be replaced by the path "`|Application|MyView|displayFilteredSelection`".

Here the relationships of some Dispatchers in the macro when it is invoked:

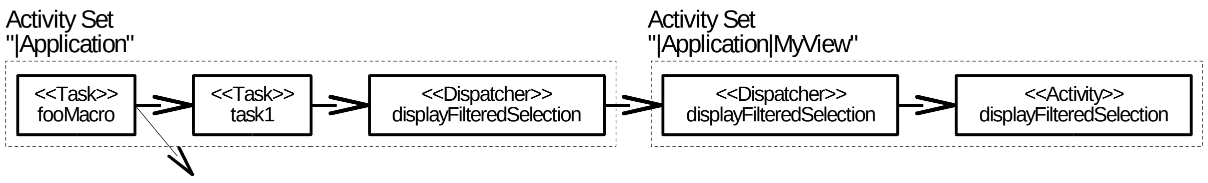


Figure 4.5-4 Replacing Dispatchers paths (1)

`task1` uses the `displayFilteredSelection` Dispatcher of Activity Set `Application`. The following picture illustrates the relationships after replacing the Dispatcher paths before the actual execution of `fooMacro`.

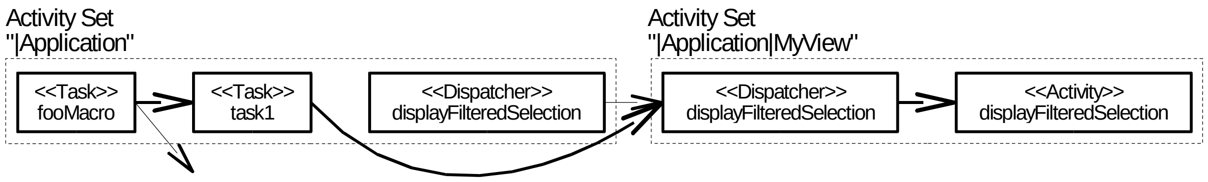


Figure 4.5-5 Replacing Dispatcher paths (2)

Now, `task1` directly uses the `displayFilteredSelection` Dispatcher of Activity Set `MyView.`

The macro may then look as follows:

Program 4.5-8 Defining a macro (script)

```
<RCTask>
   <name>fooMacro</name>
   <do>
      <RCTask>
         <name>task1</name>
         <dispatcher directives="First">
            |Application|MyView|displayFilteredSelection</dispatcher>
         <args>
            <dispatcher>
               |Application|TextEditor[0]|TextEditorTool[0]|getSelection
            </dispatcher>
            <string>foo</string>
...
```

Note that the macro is only changed temporarily until the execution has ended. The change has no impact on the original configuration.

How a Dispatcher determines its currently associated Activity or Activities, is described in 5.6.2. Normally, a Dispatcher gathers acquired Activities (depending on the specified directives) and invokes methods on them, but in this case only their paths are determined and finally used as Dispatcher paths in the macro. So, when the macro is actually executed, the same Dispatchers are used, even if the ranking has changed.

For example, if the Activity Set "`|Application|`**`MyView2`**`|displayFilteredSelection`" gets the highest priority in "`|Application`" during the execution of the macro, the `displayFilteredSelection` Dispatcher will still be used from Activity Set "`|Application|`**`MyView`**". See below.

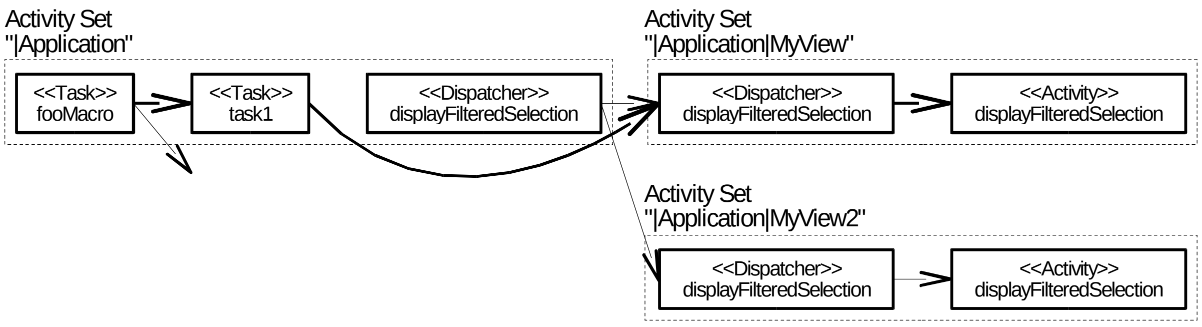


Figure 4.5-6 Replacing Dispatcher paths (3)

4.6 Delegation using PCoC

As we learned in Section 2.2.4, delegation is a dispatch mechanism for object composition.

The overhead for managing (class-based) method dispatch for implementation inheritance is relatively low. The concept is very mature. However, object composition has some advantages compared to implementation inheritance. We get the flexibility to compose object sets at runtime, which increases the level of reusability of single objects, and which adds more dynamics to a system. However, object composition usually does not support (object-based) delegation, but only forwarding. In order to accomplish delegation across different objects, we can make the forwarding mechanism stronger. That is what PCoC tries to achieve—a delegation mechanism (using Dispatchers) that passes information about the Activity Set (the context) in which an Activity is requested, as separate parameter to the Activity.

Program 4.6-1 shows an implementation of a class `C` as PCoC Service Provider. We define class `C` and the Activities `M1` and `M3`. `M1` calls `M2` which may be defined in another provider. `M3` executes some code. Note that the actual implementation of `doPerform` is not relevant here.

Program 4.6-1 Delegation using PCoC (1)

```
public class C extends PCCServiceProvider {
   public C() {
      activate(); // creates an Activity Set with the class name
                  // ("C") as name. Use setType("<typename>") for setting another
   }              // name, or use constructor of the base class:
super("<typename>")

   protected void setupActivitySet() {
      super.setupActivitySet();
      addActivity(new M1Activity());
      addActivity(new M3Activity());
   }

   class M1Activity extends PCCSingleActivity {
      M1Activity() { super("M1", "Methods", "", ""); }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         ...                                  // do something
         // invoke M2 in the original context
         si.getReceiver().perform("M2", null, PCCDirectives.first());
         return null;
      }
   }

   class M3Activity extends PCCSingleActivity {
      M3Activity() { super("M3", "Methods", "", ""); }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         ...  // do something
         return null;
      }
   }
}
```

Now that we have Activities Provider `C` with Activities `M1` and `M3`, we also need an Activities Provider `CB` which defines the Activity `M2`.

Program 4.6-2 Delegation using PCoC (2)

```
public class CB extends PCCServiceProvider {
   public CB() {
     activate(); // creates an Activity Set (Activity Set) with the class name
                 // ("CB") as name. Use setType("<typename>") for setting another
                 // name, or use constructor of the base class: super("<typename>")
   }
   protected void setupActivitySet() {
      super.setupActivitySet();
      addActivity(new M2Activity());
   }

   class M2Activity extends PCCSingleActivity {
      M2Activity() { super("M2", "Methods", "", ""); }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         ...                                   // do something
         // invoke M3 in the original context
         si.getReceiver().perform("M3", null, PCCDirectives.first());
         return null;
      }
   }
}
```

Now the Activity `M2` is also defined. It invokes an Activity `M3` in the context of the original receiver—the Activity Set where `M2` was invoked (`si.getReceiver()`).

The following client code creates a delegation relationship by connecting the Activities Providers from above.

Program 4.6-3 Delegation using PCoC (client code)

```
/** create components */
PCCActivitiesProvider c = new C();
PCCActivitiesProvider cb = new CB();

/** create composite context */
PCCActivitySet context = PCCRegistry.getOrCreateActivitySet("|CombinedSet");
context.acquire(c.getActivitySet());
context.acquire(cb.getActivitySet());

/** perform (invoke) the operation M1 */
context.perform("M1", null, PCCDirectives.first());
```

We instantiate two Activities Providers, `c` and `cb`. Then, we create an Activity Set "`CombinedSet`" that acquires from `c` and `cb`, respectively their Activity Sets. `CombinedSet` represents a common "self" (we say, a common context) for the acquired Activity Sets. The `perform` method of Activity Set `context` is used to invoke Activity `M1`, which is in this case provided by `c`. We say, the request for executing Activity `M1` is delegated to `c`. `M1` invokes `M2` (as shown in Program 4.6-1). This request will be delegated to `cb`. `M2` subsequently invokes `M3` on the Activity Set `context` where the first Activity (`M1`) in the delegation sequence was invoked (see Program 4.6-2). `context` was the original receiver of the `M1` request, and then of the `M2` request.

If `c` directly acquires from `cb`, we need no separate Activity Set to specify a common context. In this case, `c`, or more precisely its Activity Set, provides the common context for `c` and `cb`.

Program 4.6-4 Delegation using PCoC (client code)

```
PCCActivitiesProvider c = new C();
PCCActivitiesProvider cb = new CB();

c.add(cb); // add child; or call c.acquire(cb) directly without adding cb as child
c.perform("M1", null, PCCDirectives.first());
```

In this example, `cb` is added as element to container `c` (i.e., the Activity Set path of `cb` is contained in that of `c`), and is also acquired by `c`. If `cb` needs not be an element of `c`, we can also directly call `c.acquire(cb)`. In both cases, `c` becomes the delegation child of `cb`.

We invoke Activity `M1` by using Activities Provider `c`. The delegation sequence is the same as in Program 4.6-3.

We can use the same mechanism for simple forwarding, simply by not using the sender info parameter. Instead, we can use the current Activity Set (the current owner of the Activity) within an Activity. Compare the following code fragment to Program 4.6-2.

Program 4.6-5 Delegation using PCoC (3)

```
public class CB extends PCCServiceProvider {
...
   class M2Activity extends PCCSingleActivity {
      M2Activity() { super("M2", "Methods", "", ""); }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         ...                                     // do something
         // invoke M3 in the same Activity Set; corresponds to
         // getActivitySet().perform("M3", null, PCCDirectives.first())
         perform("M3", null, PCCDirectives.first());
         return null;
      }
   }
}
```

In this case, the execution of Activity `M3` by Activity `M2` will fail, since `M3` is neither provided, nor acquired by Activity Set `CB` (the current owner of Activity `M2`). However, we can nevertheless request Activities using the registry.

Program 4.6-6 Delegation using PCoC (4)

```
public class CB extends PCCServiceProvider {
...
   class M2Activity extends PCCSingleActivity {
      M2Activity() {
         super("M2", "Methods", "", "");
      }

      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         ...                                     // do something
         // invoke M3 in the given Activity Set
         PCCRegistry.getActivitySet("|CB").perform("M3", null,
                                                 PCCDirectives.first());
         return null;
      }
   }
}
```

Activities can be provided by one Activity Set, but added to another.

Program 4.6-7 Delegation using PCoC (5)

```
public class CB extends PCCServiceProvider {
   public CB() {
      activate(); // creates an Activity Set (Activity Set) with the class name
                  // ("CB") as name. Use setType("<typename>") for setting another
                  // name, or use constructor of the base class:
super("<typename>")
   }
   protected void setupActivitySet() {
      super.setupActivitySet();
      addActivity("C", new M2Activity()); // add Activity to Activity Set "C", but
                                          // its provider is the current ("CB")
      addActivity(new M4Activity());  // add Activity to the current Activity Set
   }
   class M2Activity extends PCCSingleActivity {
      M2Activity() { super("M2", "Methods", "", ""); }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         ...                              // do something
      }
   }
   class M4Activity extends PCCSingleActivity {
      M2Activity() { super("M4", "Methods", "", ""); }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         ...                              // do something
      }
   }
}
```

In this case, the Activity `M2` is added to Activity Set `C` (the Activity Set associated with Activities Provider `C`). However, it will be performed in `CB`. We say, we enhance `C` by an Activity of Activities Provider `CB`. This is useful, for example, if we do not have access to the source code of `C`. See Program 4.3-12 for a description of the `addActivity` method.

Program 4.6-8 Delegation using PCoC (client code)

```
/** create components */
PCCActivitiesProvider c = new C();
PCCActivitiesProvider cb = new CB();

/** perform (invoke) the operation M1 */
c.perform("M1", null, PCCDirectives.first());
```

Since Activity `M2` is already added to `C`, is not necessary that `C` acquires `CB`. Compare to Program 4.6-3 and Program 4.6-4. The advantage of this approach is, as opposed to acquisition, that `M2` has access to both Activity Sets—the one where it has been added, and that of its provider (which may implement other Activities or methods necessary for the execution of `M2`).

Program 4.6-9 Delegation using PCoC (5)

```
public class CB extends PCCServiceProvider {
...
   class M2Activity extends PCCSingleActivity {
      M2Activity() { super("M2", "Methods", "", ""); }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args) {
         // invoke M3 in the Activity Set,where this Activity has been added ("C")
         getActivitySet().perform("M3", null, PCCDirectives.first());
         // invoke M4 in the Activities Provider, respectively Activity Set,
         // where this Activity has been created ("CB")
         getProvider().perform("M4", null, PCCDirectives.first());
         return null;
      }
   }
}
```

Note that, as long as there is a reference to a single Activities Provider, all related objects such as Activities, Dispatchers, (acquired) Activity Sets, etc., are prevented from being garbage collected. In order to clean up and remove an Activities Provider, `terminate` has to be called. Its Activity Set will be unregistered from the PCoC registry and released for later cleanup together with all related objects.

Program 4.6-10 Cleaning up Activities Providers

```
cb.terminate(); // terminates cb, removes its Activities, and discards its
                // Dispatchers from acquiring Activity Sets (in this case,
                // that of c)
c.terminate();  // terminates c
```

Delegation is discussed in more detail in [SZYPE98] on Pages 119ff. Other related pages are 155f (*Events and messages*), 159ff (V*ery late binding: dispatch interfaces and meta-programming*), and 324ff (*On the horizon*). [GRIFF98] also describes common issues of delegation on Pages 254ff (*Event channels*), and 424ff (*Delegation*).

Apart from dynamic and late object composition, the availability of delegation is also a requirement of prototypical programming languages. Read more about delegation in prototypical systems in [LIEBER86].

5 Detailed Concepts and Implementation

This chapter explains the design and most essential concepts of PCoC.

5.1 PCoC-Related Packages and Responsibilities

This section describes dependencies between PCoC and other packages, as well as the responsibilities of these packages.

The packages are not described in detail, and there may be other required features that have not yet been considered. However, this section may help to get an overview of how PCoC is structured.

We have chosen the following set of packages to build a PCoC application:

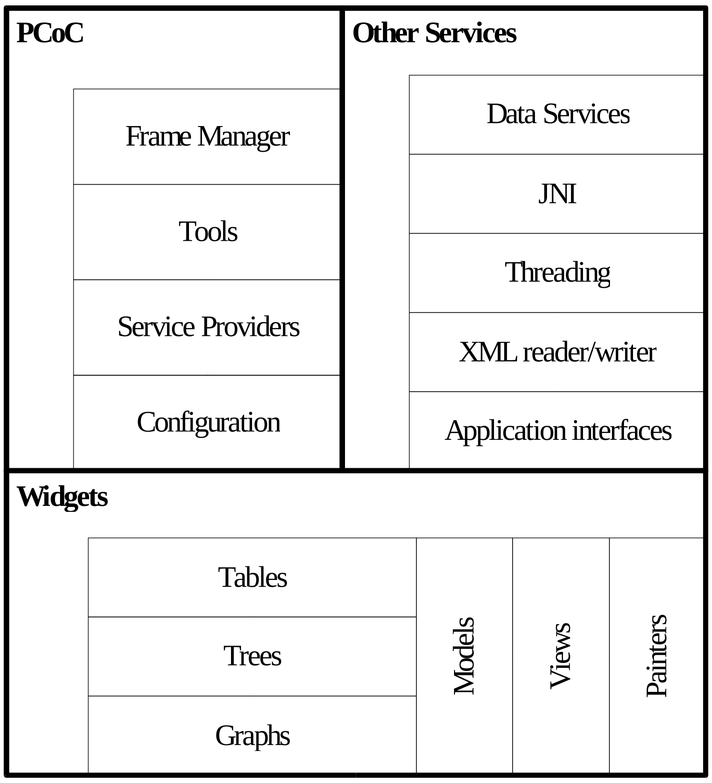


Figure 5.1-1 Overall Architecture

PCoC includes Tools, Service Providers, a configuration library (reading and writing data structures used to customize Activities Providers and a whole application), and a frame manager responsible for menu and toolbar setup, for window management, and for composing, starting up (initialize), and terminating Activities Providers.

Widget libraries include view classes for tables, trees, and graphs, and proper data models and painters.

Other services are data services (general data models and access operations), a threading model, and component and application interfaces.

Figure 5.1-2 illustrates how these packages are related to each other. There may be changes in detail, e.g. the package structure, implementation details, etc. This does not have an impact on the overall design.

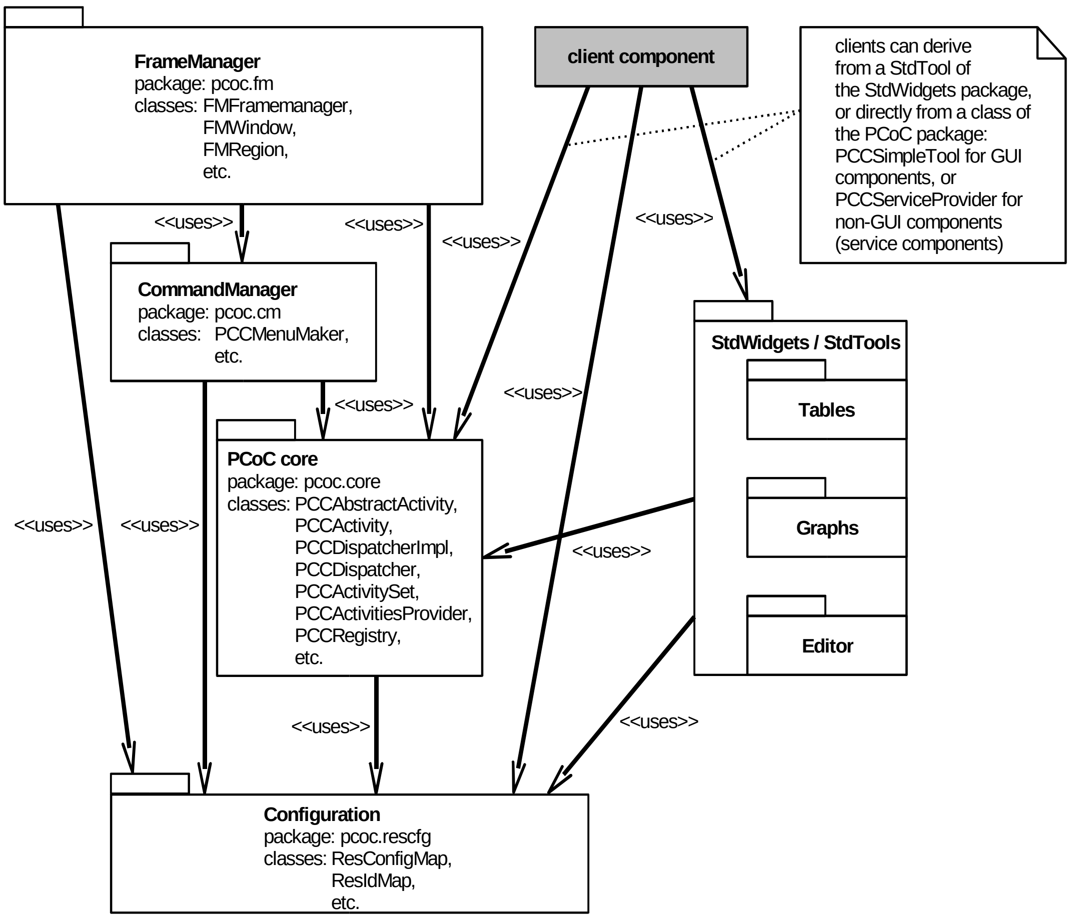

FrameManager
package: pcoc.fm
classes: FMFramemanager, FMWindow, FMRegion, etc.
client component
clients can derive from a StdTool of the StdWidgets package, or directly from a class of the PCoC package: PCCSimpleTool for GUI components, or PCCServiceProvider for non-GUI components (service components)
<<uses>>
CommandManager
package: pcoc.cm
classes: PCCMenuMaker, etc.
StdWidgets / StdTools
Tables
Graphs
Editor
PCoC core
package: pcoc.core
classes: PCCAbstractActivity, PCCActivity, PCCDispatcherImpl, PCCDispatcher, PCCActivitySet, PCCActivitiesProvider, PCCRegistry, etc.
Configuration
package: pcoc.rescfg
classes: ResConfigMap, ResIdMap, etc.

Figure 5.1-2 PCoC and related packages

Packages and classes, as shown in Figure 5.1-2, have only dependencies in one direction. This layered design with only one-way dependencies makes it easy to make modifications (extensions) to packages, or to replace them by other implementations without having an impact on the packages they use.

5.1.1 Configuration

The configuration, respectively the Configuration Manager, is used to customize the look and feel of an application. This includes layout definitions for the whole application and its Tools, definitions for menus and toolbars and their associated Tasks, etc.

A central XML-based configuration file is used as root configuration, and can import others. Preferably each Activities Provider has its own configuration file. The configuration (all configuration files of an application) can be reloaded at runtime. Definitions are loaded from the configuration files into a hash map (`ResConfigMap`) where they can be looked up by name. Another map (`ResIdMap`) allows the lookup with integer keys. For example, menu items may be associated with integer keys instead of strings.

5.1.2 Frame Manager

The Frame Manager is responsible for the window creation, layout management (positioning of windows, cascaded or tiled layout, storing and reloading layout information), region management (repainting of invalidated regions), focus management (set focus and caret according to user input), status line management (switching displayed information on focus changes), drag and drop management.

5.1.3 Command Manager

The command manager is responsible for menu and toolbar management (setting up and updating), accelerator keys handling, and command management (handling do, undo, and redo operations).

5.1.4 PCoC Core

The PCoC core package contains almost all classes relevant for this thesis. This includes Activity classes (abstract base class for all kinds of Activities, and base classes for deriving in client code), Dispatcher and DispatcherImpl (the proxy and its implementation), Task and TaskQueue (a queue for the asynchronous execution of Tasks), Material (argument container class), Activity Set and Activity Set Registry, and the Activities Provider class.

5.1.5 Simple and Combined Tools

The base classes for client components, `PCCSimpleTool`, and `PCCServiceProvider` can be found in the PCoC root package (`pcoc`), as well as the `PCCCombinedTool` class (the generic container for Simple Tools).

5.1.6 Standard Widgets / Standard Tools

Standard Widgets are GUI elements (e.g. edit views, table views, graph views, etc.) that are needed to make templates for most common Tools—so-called *Standard Tools*. A *Standard Tool* is a Simple Tool containing a table, tree, graph, or editor. Such a Tool, respectively its template implementation (base classes), is assembled from a data model, a view, painters, etc.

The main purpose of Standard Widgets and Standard Tools is to provide uniform and basic implementations of common Tools. This reduces implementation and maintenance time and effort for this kind of component. A component developer can concentrate on implementing the actual functionality of his component, instead of creating solutions for common issues from scratch.

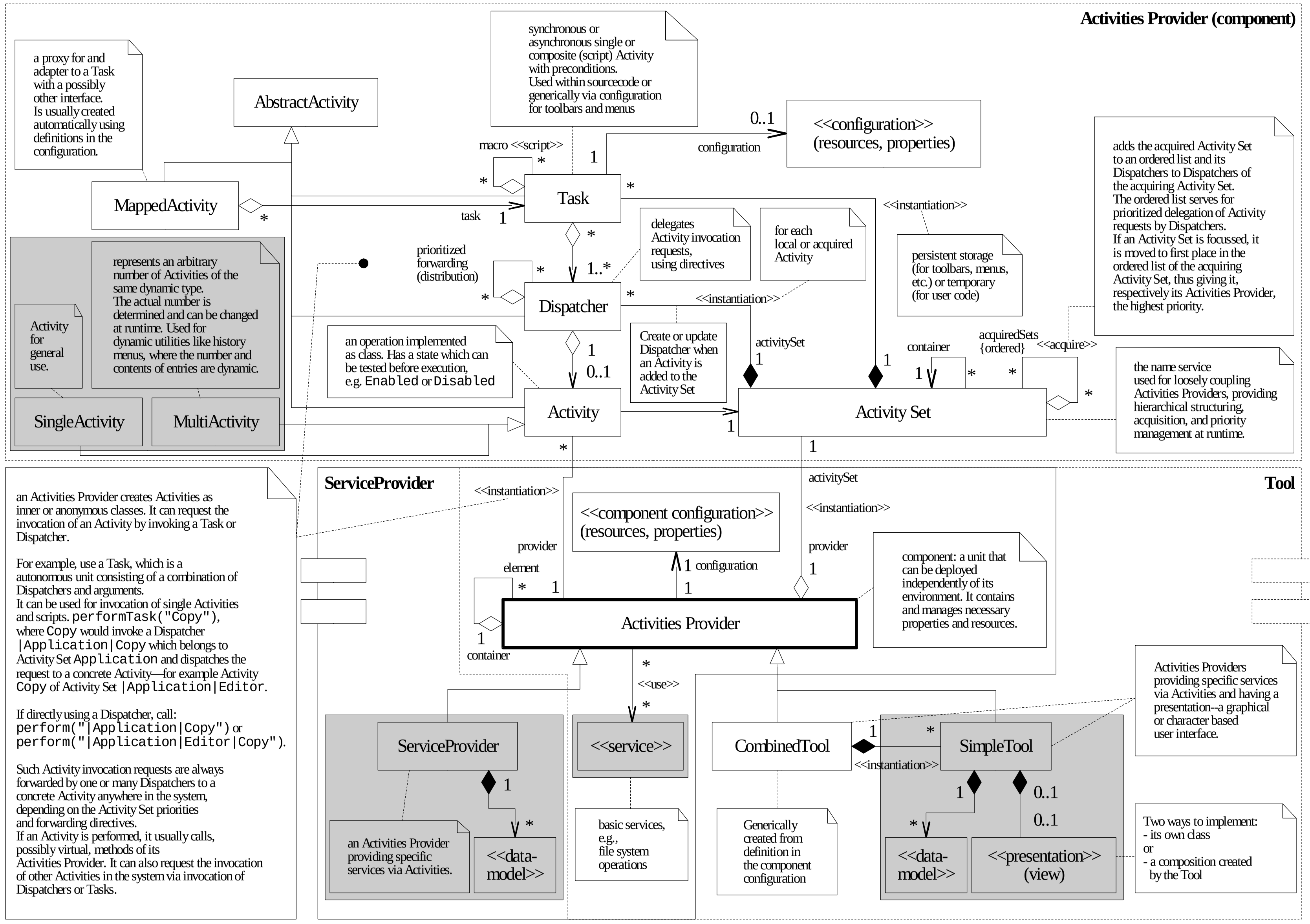
5.1.7 Remarks

We have chosen this architecture, because (a) it allows clear separation of responsibilities and (b) dependencies are uni-directional. This modular (layered) architecture enables us to make modifications to packages, or completely replace them by others, without having an impact on packages they use. See Figure 5.1-1. This is the software design recommended in the technical literature.

Higher layers use those below and “pull” information from them. Packages on a lower layer may provide extensions for higher layer packages, for example, abstract base classes, template methods, listener interfaces, etc. This can keep the dependencies clear and uni-directional and keeps the maintenance effort low.

5.2 Activities Provider

Activities Providers are components based on the PCoC framework. Figure 5.2-1 illustrates how all PCoC classes are related to each other, and describes their meanings. The elements shown in grey are those implemented by component developers. Others are created automatically, such as Dispatchers, or generated depending on the configuration, such as Tasks.



a proxy for and adapter to a Task with a possibly other interface. Is usually created automatically using definitions in the configuration.

synchronous or asynchronous single or composite (script) Activity with preconditions. Used within sourcecode or generically via configuration for toolbars and menus

adds the acquired Activity Set to an ordered list and its Dispatchers to Dispatchers of the acquiring Activity Set. The ordered list serves for prioritized delegation of Activity requests by Dispatchers. If an Activity Set is focussed, it is moved to first place in the ordered list of the acquiring Activity Set, thus giving it, respectively its Activities Provider, the highest priority.

represents an arbitrary number of Activities of the same dynamic type. The actual number is determined and can be changed at runtime. Used for dynamic utilities like history menus, where the number and contents of entries are dynamic.

Activity for general use.

delegates Activity invocation requests, using directives

for each local or acquired Activity

persistent storage (for toolbars, menus, etc.) or temporary (for user code)

an operation implemented as class. Has a state which can be tested before execution, e.g. `Enabled` or `Disabled`

Create or update Dispatcher when an Activity is added to the Activity Set

the name service used for loosely coupling Activities Providers, providing hierarchical structuring, acquisition, and priority management at runtime.

an Activities Provider creates Activities as inner or anonymous classes. It can request the invocation of an Activity by invoking a Task or Dispatcher.

For example, use a Task, which is a autonomous unit consisting of a combination of Dispatchers and arguments.
It can be used for invocation of single Activities and scripts. `performTask("Copy")`, where `Copy` would invoke a Dispatcher `|Application|Copy` which belongs to Activity Set `Application` and dispatches the request to a concrete Activity—for example Activity `Copy` of Activity Set `|Application|Editor`.

If directly using a Dispatcher, call:
`perform("|Application|Copy")` or
`perform("|Application|Editor|Copy")`.

Such Activity invocation requests are always forwarded by one or many Dispatchers to a concrete Activity anywhere in the system, depending on the Activity Set priorities and forwarding directives.
If an Activity is performed, it usually calls, possibly virtual, methods of its Activities Provider. It can also request the invocation of other Activities in the system via invocation of Dispatchers or Tasks.

component: a unit that can be deployed independently of its environment. It contains and manages necessary properties and resources.

Activities Providers providing specific services via Activities and having a presentation--a graphical or character based user interface.

an Activities Provider providing specific services via Activities.

basic services, e.g., file system operations

Generically created from definition in the component configuration

Two ways to implement:
- its own class
or
- a composition created by the Tool

Figure 5.2-1 Activities Provider Architecture

The picture should be self-explanatory. Please read the notes. Note that, when an Activity is created and added to an Activity Set, the Activity Set automatically creates and adds a corresponding Dispatcher.

5.3 Containment Hierarchy

Activities Providers (PCoC components), respectively their Activity Sets can be organized in a logical containment hierarchy, like a package hierarchy in Java or a directory tree in file system. Containment hierarchies may depend (partially) on the hierarchy of GUI elements.

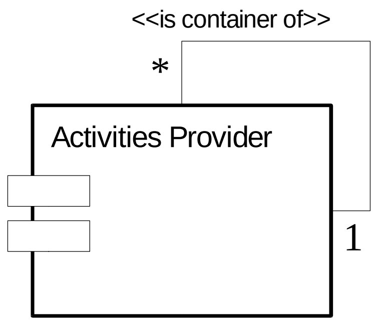


Figure 5.3-1 Containment hierarchy

Figure 5.3-1 shows the containment relationship of Activities Providers. An Activities Provider can be the container of no, one, or many others.

The nodes in a containment hierarchy are distinguished by path. A path is, for example, used to lookup Activity Sets in the PCoC registry, and to lookup Activities, Dispatchers, or Tasks via Activity Sets. Paths are used case insensitively in PCoC.

We use “|” as path separator. Paths starting with character “|” are absolute paths.

Program 5.3-1 Using an absolute Activity Set path

```
PCCRegistry.getActivitySet("|Application").perform("|Application|Copy", null,
                                                    PCCDirectives.first())
```

In this case, we invoke the Dispatcher `Copy` in the Activity Set associated with path “`|Application`”.

If the first character is not “|”, the path is considered relative to the path of the current Activity Set. For example, calling the method `perform` with path “`TextEditor|Copy`” in an Activity Set “`|Application`” is resolved to “`|Application|TextEditor|Copy`”.

Program 5.3-2 Using a relative Activity Set path

```
PCCRegistry.getActivitySet("|Application").perform("TextEditor[0]|Copy", null,
                                                    PCCDirectives.first())
```

5.3.1 Sample Hierarchy

A containment hierarchy is defined by the paths of Activities Providers, respectively their Activity Sets. Compare this to a directory tree in a file system.

Figure 5.3-2 depicts a realistic containment hierarchy for an IDE application. The Activity Sets associated with the given Activities Providers are organized accordingly. See Figure 3.4-2 and Figure 3.4-3 for screen shots.

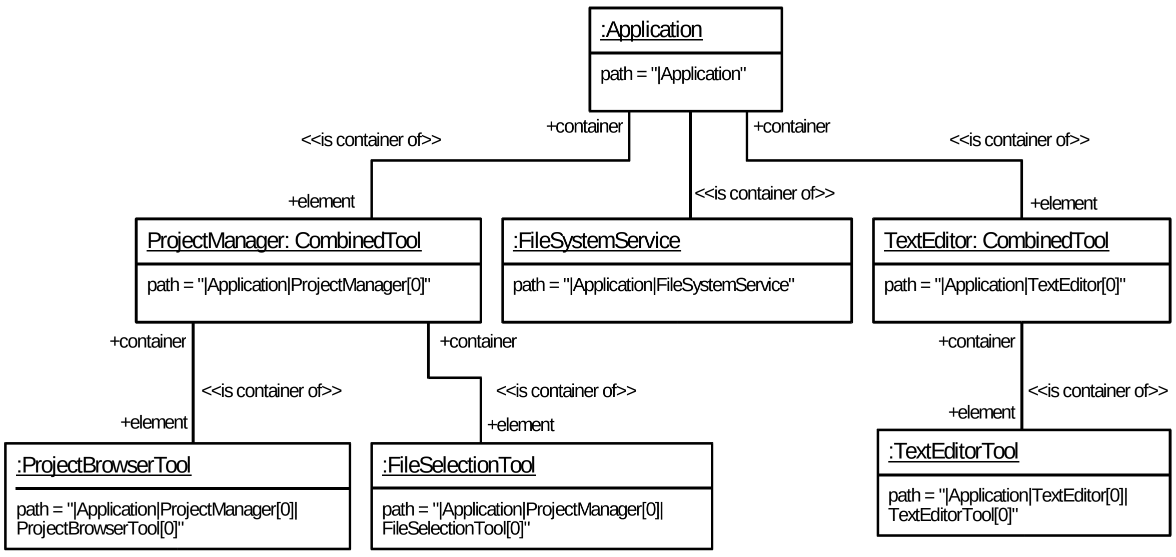


Figure 5.3-2 A possible containment hierarchy

In our example, a text editor component (a class derived from `PCCSimpleTool`, respectively its instance) has the path "`|Application|TextEditor[0]|TextEditorTool[0]`". It is a child of the Combined Tool `TextEditor[0]`.

Service Provider `FileSystemService` is child of the root Service Provider `Application`, therefore it has the path "`|Application|FileSystemService`".

Note that indexes (for example, in `TextEditorTool[0]`) are automatically created by the framework in order to get unique names for the Activity Sets of multiple instances of Activities Providers. Activities Providers have a method `getPath` which returns their current path.

5.3.2 Modifying a Containment Hierarchy

Activities Providers can be added manually to the containment hierarchy by specifying the container object or path.

Program 5.3-3 Methods for setting a container

```
void setContainer(PCCActivitiesProvider container);// from PCCActivitiesProvider
void setContainer(PCCActivitySet container);       // from PCCActivitySet
void setContainer(String container);               // from PCCActivitiesProvider
                                                   // and PCCActivitySet
```

The following code fragment illustrates the use of these methods:

Program 5.3-4 Setting a container

```
PCCActivitiesProvider a = new PCCActivitiesProvider("A");
PCCActivitiesProvider b = new PCCActivitiesProvider("B");
b.setContainer(a);  // is implicitly called when using a.add(b)
```

In this case, the Activities Provider `b` becomes the container of `a`. The containment relation can also be created via the corresponding Activity Sets. This is useful, if we only have the current paths of the Activities Providers, or some Activities Providers are not instantiated yet.

Program 5.3-5 Setting a container via paths

```
PCCActivitySet a = PCCRegistry.getActivitySet("|A");
PCCActivitiesProvider b = new PCCActivitiesProvider("B");
b.activate();  // do some initialization
b.getActivitySet().setContainer(a);  // register "B" as element of "A"
b.setContainer("|A");     // this statement has the same effect as that above
```

Now `b` has the path "`|A|B`". Next, we can retrieve the Activities Provider `b` through the registry:

Program 5.3-6 Retrieving an Activities Provider via the registry

```
PCCActivitiesProvider b = PCCRegistry.getActivitySet("|A|B").getProvider();
// or simpler:
PCCActivitiesProvider b = PCCRegistry.getActivitiesProvider("|A|B");
```

Now, let us take a look at an implementation of a simple Tool.

Program 5.3-7 SimpleTool2.java (1)

```
package example2;
import pcoc.tools.*;
import javax.swing.*;

/** A Tool providing and displaying a list of strings. */
public class SimpleTool2 extends PCCSimpleTool {
   JList fList;

   /** The constructor. Adds a service provider as element. */
   public SimpleTool2() {
      PCCActivitiesProvider childService = new SampleServiceProvider();
      add(childService);     // calls childService.setContainer(this)
   }
...
```

Program 5.3-7 shows how a Simple Tool creates and adds a Service Provider as child. The Tool's visual representation and its Activity Set may be defined as follows:

Program 5.3-8 SimpleTool2.java (2)

```
   /** Set up the Tool's GUI
     * @return A JComponent representing the GUI of this Tool
     */
   public JComponent makePresentation() {
      String [] data = { "Hello World!" };
      fList = new JList(data);
      return new JScrollPane(fList);
   }

   /** Set up the Tool's Activity Set */
   protected void setupActivitySet() {
      super.setupActivitySet();
      ...
   }
   ...
}
```

Since our Tool has no container in this case, it gets the Activity Set path "`|SimpleTool2`". When this Tool is added to the GUI hierarchy, it may get, depending on the configuration (see below), for example, the path
"`|Application|CombinedTool2[0]|SimpleTool2[0]`".

Program 5.3-9 Component configuration

```
<?xml version = "1.0" encoding = "UTF-8"?>
<ResConfig name="Example1" xsi:schemaLocation="www.windriver.com/ResConfig
file:///pwe/rome/config/tools/ResConfig.xsd"
xmlns="www.windriver.com/ResConfig"
xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
...
<RCSimpleTool>
   <name>SimpleTool2</name>
   <className>example2.SimpleTool2</className>
</RCSimpleTool>
<RCCombinedTool>
   <name>CombinedTool2</name>
   <windowTitle><dockedTitle>My Combined Tool</dockedTitle></windowTitle>
   <part><simpleToolName>SimpleTool2</simpleToolName></part>
</RCCombinedTool>
...
```

The following code fragment describes the class `SampleServiceProvider`. It is initialized with the Activity Set name “`SampleService`”, that is, it can be retrieved with this name through the Activity Set registry.

Program 5.3-10 SampleServiceProvider.java

```
package example3;

import pcoc.*;
import javax.swing.*;

/** A Service Provider providing specific services (not implemented here)
  */
public class SampleServiceProvider extends PCCServiceProvider {
   /** The constructor setting the Activities Provider associated with path
     * "|Application|AService" as parent.
   public SampleServiceProvider() {
      super("SampleService");
      activate();
   }
   ...
}
```

The Service Provider added as `childService` in Program 5.3-7 is then accessible with path “`|Application|CombinedTool2[0]|SimpleTool2[0]|SampleService`”.

The container and thus the path of the corresponding Activity Set can be changed dynamically. See below:

Program 5.3-11 SampleServiceProvider.java

```
package example3;

import pcoc.*;
import javax.swing.*;

/** A Service Provider providing specific services (not implemented here)
  */
public class SampleServiceProvider extends PCCServiceProvider {
...
   public void foo() {
      PCCActivitiesProvider containerComponent =
         PCCRegistry.getActivitiesProvider("|Application|AllServices");
      containerComponent.add(this);// calls this.setContainer(containerComponent)
   }
}
```

The Service Provider can then be looked up via the PCoC registry:

Program 5.3-12 Retrieving a Service Provider via the registry

```
PCCServiceProvider sampleService;
sampleService = (PCCServiceProvider)
   PCCRegistry.getActivitiesProvider(
      "|Application|AllServices|SampleService");
sampleService = (PCCServiceProvider)
   PCCRegistry.getActivitySet(
      "|Application|AllServices|SampleService").getProvider();
```

Both statements have the same effect.

5.4 Acquisition: Dynamic Component Inheritance

This section roughly describes the environmental acquisition concept, and provides detailed insight into the PCoC acquisition concept.

5.4.1 Overview

Acquisition is a delegation mechanism that lets objects and components acquire functionality from others at runtime. As opposed to class inheritance, attributes and operations can be distributed over various objects or components, independent of the class hierarchy.

There are two forms of acquisition:

- Environmental acquisition: A mechanism that allows an object to obtain a requested feature from its environment (its parents in a containment hierarchy), if neither its class nor its base class provides the feature. There are two styles of environmental acquisition, implicit (automatic) and explicit acquisition.
- Direct acquisition: A mechanism that allows objects or components to acquire (dynamically “inherit”) features from others independent of the class hierarchy. The difference to environmental acquisition is that direct acquisition is independent of the containment hierarchy of an object.

Both forms of acquisition are needed in certain cases. In the context of this thesis, the term *acquisition* refers to either direct or environmental acquisition, depending on the context.

5.4.2 Environmental Acquisition

Environmental acquisition is important where the specific environment of objects or components provides attributes or a context that has, or can have, an impact on these objects and components.

Think of a directory tree of Web pages. The Web pages may inherit attributes from their parent directories, if these attributes are not found in the same directory. In this case, parent directories provide the environment for the child directories and contained Web pages. This is the main idea of Zope, a web server and content management system (see [ACQU98]).

See also Section 6.1 for more details.

5.4.3 Direct Acquisition

In contrast to environmental acquisition, direct acquisition is usually independent of the containment hierarchy. However, in a containment hierarchy, containers may nevertheless acquire functionality from their children, or, like with environmental acquisition, children may acquire from their containers.

5.4.3.1 Acquisition in PCoC

In an acquisition relationship between Activities Providers, respectively their Activity Sets, the acquiring component has a reference to its acquired components, and vice versa.

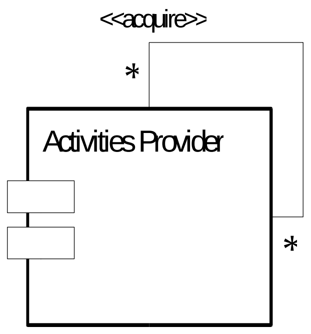


Figure 5.4-1 Acquisition

See also Figure 5.4-3 for an example of an acquisition graph.

Acquisition relationships are managed by Activity Sets and Dispatchers. An Activity Set has a dictionary of Dispatchers and an ordered list of acquired Activity Sets. A Dispatcher has a reference to an Activity and/or Dispatchers of acquired Activity Sets. See also Sections 2.3 and later.

The following picture illustrates two Activity Sets, their Dispatchers, and Activities, for the following acquisition/discard algorithm.

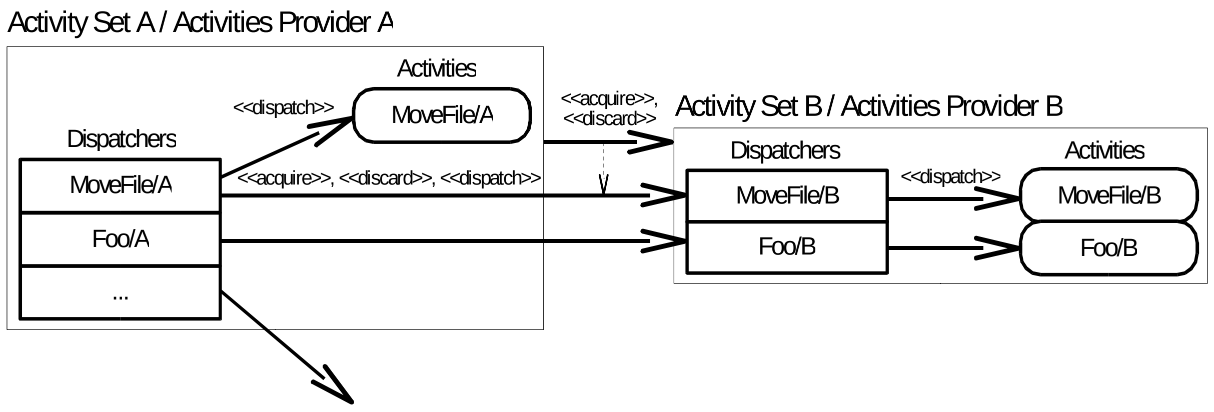


Figure 5.4-2 Acquire / discard Dispatchers

When an Activity is added or removed, or another Activity Set is acquired, the Dispatcher dictionary and the list of acquired Activity Sets are brought up to date accordingly:

- When `A` acquires another Activities Provider `B`, respectively its Activity Set, add a reference to `B` to the list of acquired Activity Sets in `A`. Add a reference to `A` to the list of listeners in `B`.
- When an Activity `MoveFile` is added to an Activity Set `A`, and no Dispatcher `MoveFile/A` with this name exists in `A`, create one with the same name (`MoveFile`) and dynamic type (for example, `Boolean MoveFile(String fromFile, String toFile)`). Add a reference to this Activity to the Dispatcher. Note that Activity and Dispatcher names globally identify a dynamic type. In this case, all Dispatchers and Activities in the system must have the signature `Boolean MoveFile(String, String)`.

- If `B` provides a Dispatcher `MoveFile/B` with the same name and type as the Dispatcher `MoveFile/A` of Activity Set `A`, add a reference to it to the list of acquired Dispatchers in Dispatcher `MoveFile/A`. If no Dispatcher with this name exists, create one. If there is a type mismatch (for example, if `MoveFile/B` returns `void` instead of a `Boolean`), throw an error and do not add `MoveFile/B`.

  Note that in this case, Activity `MoveFile/A` in Dispatcher `MoveFile/A` actually overrides `MoveFile/B`. However, when we invoke Dispatcher `MoveFile/A`, we can specify a directive (`sendDeep`) to send the request through the acquisition branch to a leaf, in this case Activity `MoveFile/B`.

  ```
  Boolean result = (Boolean)perform("|A|MoveFile", args,
     PCCDirectives.first().sendDeep());
  ```

- If `B` provides a Dispatcher `Foo/B`, and `A` does not provide a Dispatcher with the same name (and dynamic type), create a new Dispatcher `Foo/A` in `A` and add a reference to `Foo/B` to the list of acquired Dispatchers in `Foo/A`. If an Activity is added to `B` after the acquisition relationship from `A` to `B` has been established, notify `A` (as well as all other listeners of `B`) in order to update the Dispatchers in `A`.

Discard algorithm:

- When Activity Set `A` discards `B`, that is, when it breaks the acquisition relationship, remove the reference to `B` from the list of acquired Activity Sets in `A`. Remove `A` from the list of listeners in `B`.
- Remove all references to Dispatchers of `B` (`MoveFile/B` and `Foo/B`) from the lists of acquired Dispatchers in the Dispatchers of `A` (remove `MoveFile/B` from `MoveFile/A` and `Foo/B` from `Foo/A`).
- When an Activity `Foo/B` is removed from an Activity Set, remove it from the corresponding Dispatcher `Foo/B`.
- If a Dispatcher `Foo/A` becomes empty, that is, if it does not contain references to an Activity or acquired Dispatchers (`Foo/B`) anymore, remove it from `A`.

These steps are partly recursive. For example, an Activity Set `A` might acquire `B` and `B` might acquire `C` and so forth. When an Activity is added to an Activity Set `C`, a corresponding Dispatcher is created in `C`, then in `B`, and finally in `A`.

When an Activity is requested (by invoking a Dispatcher), the request is delegated to that Activity Set providing the Activity.

See Sections 5.6 and later for a detailed explanation of Dispatchers. See Section 5.8 for a detailed description of Activity Sets.

5.4.3.2 Sample Acquisition

An acquisition graph as shown in Figure 5.4-3 denotes a delegation relationship. Each Activity Set holds an ordered list of acquired Activity Sets which defines a priority ranking. If one of the acquired Activity Sets gains focus, for example through user interaction, the lists of acquiring Activity Sets are rearranged, such that this Activity Set becomes the first entry, and therefore gets the highest priority.

Figure 5.4-3 illustrates how Activities Providers acquire others, respectively their Dispatchers, and how messages are dispatched through the given acquisition branches.

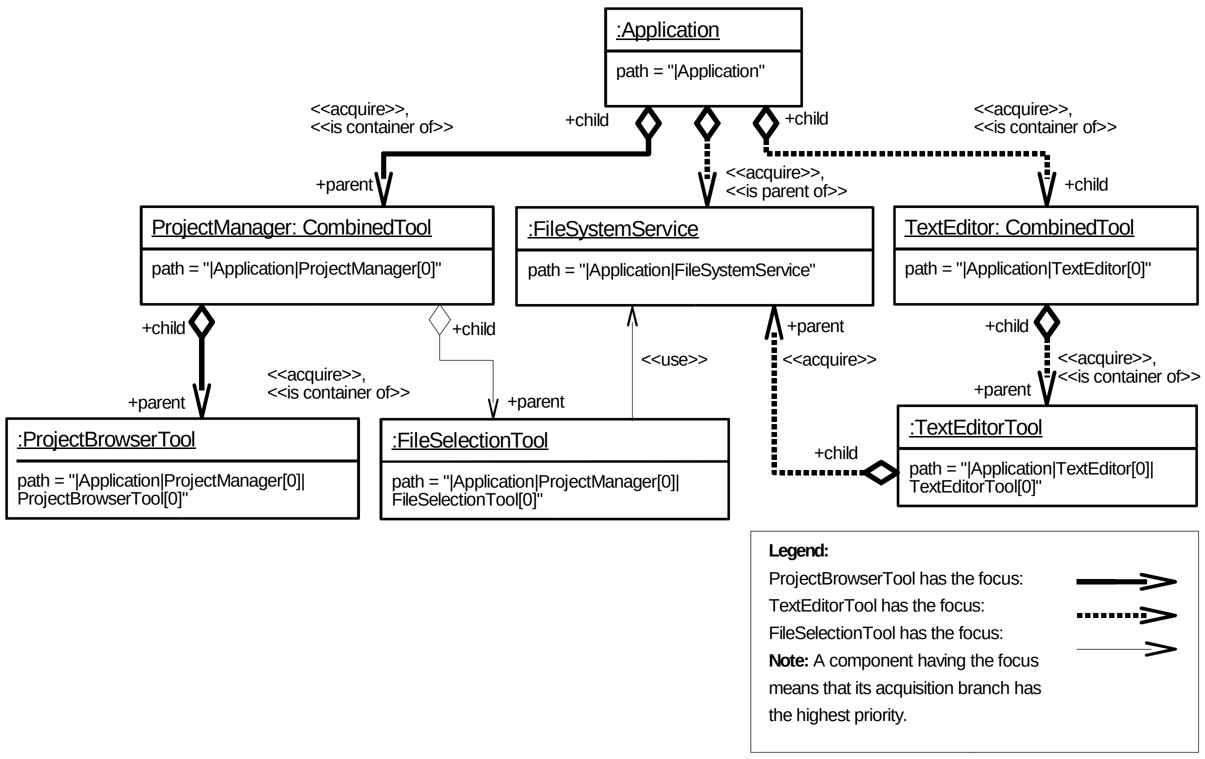

:Application
path = "|Application"
<<acquire>>,
<<is container of>>
+child
+child
<<acquire>>,
<<is container of>>
+parent
<<acquire>>,
<<is parent of>>
+child
ProjectManager: CombinedTool
path = "|Application|ProjectManager[0]"
:FileSystemService
path = "|Application|FileSystemService"
TextEditor: CombinedTool
path = "|Application|TextEditor[0]"
+child
+child
<<use>>
+parent
<<acquire>>
+child
<<acquire>>,
<<is container of>>
+parent
+parent
<<acquire>>,
<<is container of>>
+parent
:TextEditorTool
path = "|Application|TextEditor[0]|TextEditorTool[0]"
+child
:ProjectBrowserTool
path = "|Application|ProjectManager[0]|ProjectBrowserTool[0]"
:FileSelectionTool
path = "|Application|ProjectManager[0]|FileSelectionTool[0]"
Legend:
ProjectBrowserTool has the focus:
TextEditorTool has the focus:
FileSelectionTool has the focus:
Note: A component having the focus means that its acquisition branch has the highest priority.

Figure 5.4-3 An Acquisition Graph

The Service Provider `Application` acquires the Tools `TextEditor` and `ProjectManager`, and a Service Provider `FileSystemService`. More precisely, its Activity Set acquires the Dispatchers of their Activity Sets. This is similar to a multiple inheritance. If the execution of an Activity is requested in `Application`, then the request is delegated to the Activity Set providing the Activity. If more than one provides it, the most recently focussed is used, or in the case of a broadcast, all are used.

The priority used for delegating Activity requests is determined by the order in the list of acquired Activity Sets. In our example, the order of acquired Activity Sets in `Application` is `ProjectManager` (index `0` in the list), `FileSystemService` (index `1`), and `TextEditor` (index `2`). However, the node (in the acquisition branch) where a Dispatcher is invoked in order to delegate an Activity request, has always the highest priority. Note that indexes in Activity Set names (for example, in `TextEditorTool[0]`) are automatically created by the framework in order to get unique names for the Activity Sets of multiple instances of Activities Providers. They do not reflect the priority ranking in the acquisition graph.

In this case, `Application` has the highest priority. If itself does not provide a requested Activity, the request is delegated via the Activities Provider, respectively Activity Set, with the highest priority in `Application`, which is in this case `ProjectManager`. The request is then delegated further to `ProjectBrowserTool`.

We say, the branch `Application-ProjectManager-ProjectBrowserTool` has the highest priority. That is, the `ProjectManager` Activity Set is at the first position (index `0`) in the list of acquired Activity Sets in `Application`, and that of `ProjectBrowserTool` at the first position in `ProjectManager`. Let as assume that the priority changes through user interaction, more precisely through a mouse-click into `TextEditorTool`. In this case, the branch `Application-TextEditor-TextEditorTool` gets the highest priority. `FileSystemService` gets in this case the highest priority relative to `TextEditorTool`, but may still have a lower priority relative to `Application`.

Section 5.6 describes the dispatch mechanism in detail.

5.4.3.3 Acquiring / Discarding a Component

We assume an acquisition relationship like illustrated in Figure 5.4-3, where `Application` is the container of `FileSystemService` and acquires it.

Program 5.4-1 FileSystemService.java

```
package example4;
import pcoc.*;

/** A Service Provider for file system operations */
public class FileSystemService extends PCCServiceProvider {
   /** The constructor setting the name and id to "MyFileSystemService".
     * If the constructor of the base class is not explicitly called with a
     * string argument, the class name is used as name.
     */
   public FileSystemService() {
      super("MyFileSystemService");
      activate();
   }

   /** Set up the Activity Set */
   protected void setupActivitySet() {
      super.setupActivitySet();
      addActivity(new MoveFileActivity());  // add an operation
   }
   ...

   /** An Activity for moving a file on the file system.
   class MoveFileActivity extends PCCSingleActivity {
      MoveFileActivity() {
         super("moveFile", "File", "Boolean", "String,String");
      }
      ...
   }
   ...
}
```

In Program 5.4-1 we see a file-system Service Provider which adds the Activity `MoveFileActivity` in `setupActivitySet`. This Activity gets the name "`moveFile`". It overrides the method `doPerform` which defines the actual behavior of this Activity, and `doGetState` which returns the current state. See a more detailed implementation in Program 4.3-15.

Note that the direct base class of `FileSystemService` (`PCCServiceProvider`) or indirect ones may already have added Activities.

Program 5.4-2 is another implementation of a Service Provider. It serves as root component of a PCoC application. It instantiates `FileSystemService` and adds it as element.

Program 5.4-2 Application.java

```
package example4;

import pcoc.*;

/** A root component in the acquisition graph of an application
*/
public class Application extends PCCServiceProvider {
   public Application() {
      activate();
      PCCActivitiesProvider fsProvider = new FileSystemService();
      add(fsProvider);     // calls acquire(fsProvider)
   }
   ...
}
```

With the current implementation, by default, a container acquires all its elements, and thus acts as their delegation child. This behavior can be changed individually by either using another call `add(fsProvider, false)`, or subsequently removing the acquisition relationship to the child using the method `discard`. For the example above we would call `discard(fsProvider)`.

Activities Providers can acquire others independent of their containment relationship by using `acquire(<parent>)`. See the example below.

Program 5.4-3 AllServices.java

```
package example4;
import pcoc.*;
import javax.swing.*;

/** A Service Provider combining other Service Providers through acquisition
  */
public class AllServices extends PCCServiceProvider {
   /** The constructor acquiring from known Service Providers.
     */
   public AllServices() {
      activate();
      acquire(   // calls getActivitySet().acquire(...)
         PCCRegistry.getActivitySet("|Application|MyFileSystemService"));
      acquire(
         PCCRegistry.getActivitySet("|Application|MyOtherService"));
      ...
   }
   ...
}
```

The Activities Provider `AllServices`, respectively its Activity Set, becomes an acquisition child of the Activity Sets `MyFileSystemService` and `MyOtherService`. We use the registry to look up the Activity Sets.

In order to break up an acquisition relationship, we can use the method `discard`. When discarding an Activities Provider or Activity Set, it is not necessarily destroyed. Other objects may still have references to `fileSystemService` and prevent it from being garbage collected.

Program 5.4-4 Discarding acquisition relationship

```
PCCActivitySet fileSystemService =
   PCCRegistry.getActivitySet("|Application|MyFileSystemService");
PCCActivitySet allServices =
   PCCRegistry.getActivitySet("|Application|AllServices");
allServices.discard(fileSystemService); // discard fileSystemService from
                                        // allServices

// semantically equivalent:
allServices.getProvider().discard(fileSystemService.getProvider());
```

5.4.3.4 Changing the Priorities (the Ranking) of Acquired Activity Sets

The following code fragment illustrates how the priority ranking of an acquisition relationship can be changed. We use the acquisition graph of Figure 5.4-3.

Program 5.4-5 Changing the priority ranking in an acquisition graph

```
PCCActivitySet app = PCCRegistry.getActivitySet("|Application");
PCCActivitySet pm = PCCRegistry.getActivitySet("|Application|ProjectManager[0]");
PCCActivitySet te = PCCRegistry.getActivitySet("|Application|TextEditor[0]");

app.moveFirst(pm); // ProjectManager[0] has/gets the highest priority;
                   // corresponds to app.moveTo(pm, 0)
app.moveFirst(te); // change the focus to TextEditor[0]
app.moveLast(te);  // give TextEditor[0] the lowest priority
app.moveTo(te, 2); // move TextEditor[0] to position 2 in the priority ranking

app.perform("copy", PCCDirectives.first());  // directive first is the default
                                             // and can be omitted
app.perform("moveFile");
```

The most common case is that an Activity Set is moved to the first position in the priority ranking of an acquiring Activity Set. For example, a frame manager (a layout manager for Tools in the GUI of an application), may call `moveFirst` for the Activity Set of a Tool when the user interactively activates the Tool. A Dispatcher delegates Activity requests to the Activity Set with the highest priority ranking (the foremost Activity Set in the ordered list of acquired Activity Sets) which provides the requested Activity. If a text editor Tool has the focus and Dispatcher directive `PCCDirective.first` is used, `copy` is delegated to this Tool. `moveFile` is nevertheless delegated to `FileSystemService`, although it has not the highest priority. However, since there is no Activity Set with a higher priority that provides a `moveFile` Activity, that of `FileSystemService` is used. Section 5.6.2 describes the dispatch algorithm in more detail.

5.4.4 Remarks

Both, environmental and direct acquisition support delegation. Objects or components can be acquired at runtime, either automatically or explicitly. In contrast to environmental acquisition, direct acquisition allows also discarding of acquisition relationships.

PCoC supports both kinds of acquisition, but uses mainly direct acquisition. Acquisition can be applied to Activities Providers, or more precisely their Activity Sets only.

5.5 Activities and their Interfaces

We assume that the reader has read the general definitions in Chapter 3, and the basics of Activities in Section 4.3.5.

This section describes the Activity interface—an interface that is specified and checked at runtime, and its use with Activities. It is a combination of a parameter and a return value interface of an Activity. These interfaces specify sets of types and are stored as instances of the class `PCCActivityInterface`. This class ensures type safety for arguments and return values of Activities at runtime. See Section 4.3.4 for a description of the dynamic Activity type, and Section 4.3.3 for a description of the argument container class of PCoC.

5.5.1 Interface Specification

The interface of an Activity is specified in its constructor.

Program 5.5-1 PCCSingleActivity constructor

```
public class PCCSingleActivity {
   /** Activity constructor. Sets the dynamic type of this operation and
     * is called in subclasses.
     * @param name The name of this Activity. All Activities with the same
     *    name must have the same interface (resultType,
     *    argumentType) and category.
     * @param category A virtual category to which this Activity should belong.
     *    This can be used to update the states of all Activities of a category
     *     at once, or to remove them from their Activity Sets.
     * @param resultType Return value interface. See PCCActivityInterface.
     * @param argumentType Parameter interface. See PCCActivityInterface.
   public PCCSingleActivity(String name, String category,
     String resultType, String argumentType);
}
```

Activities must be derived from `PCCSingleActivity` or `PCCMultiActivity`. The latter is not described here, since it does not add any relevant concepts. The only difference to `PCCSingleActivity` is that its instances represent sets of Activities which are distinguished by a separate index parameter. The index parameter is always the first parameter in the Material passed to a multi-Activity. An Activity class must pass an interface specification to this base class.

Program 5.5-2 A User Implemented Activity

```
public class MoveFileActivity : PCCSingleActivity {
   /** Activity constructor. Sets the dynamic type of this operation.
     * Type declaration: name "moveFile", category "File", returns a Boolean
     *   object, needs two strings, the original file path and the new one
   MoveFileActivity() {
      super("moveFile", "File", "Boolean", "String,String");
   }
   ...
}
```

In this example an Activity is specified as having a `Boolean` return value, and two `String` objects as arguments. PCoC converts these `String` specifiers to `PCCActivityInterface` objects. These are managed by the framework in a lookup table to share them with Activities with the same interface.

5.5.2 Retrieving Interfaces

The following methods of `PCCAbstractActivity` retrieve the return value and parameter types of a `PCCAbstractActivity`.

Program 5.5-3 Parameter and Return Value Interface of PCCAbstractActivity

```
public PCCActivityInterface getArgumentType();
public PCCActivityInterface getResultType();
```

These interfaces are used by the framework in order to check the types of the arguments passed to an Activity when it is invoked, and its return value.

5.5.3 Activity Interface Class

The interfaces specified in the constructors of Activities are managed by the class `PCCActivityInterface`. The interface declaration strings specified in Activity constructors are passed to the `fromString` method of `PCCActivityInterface`. The most relevant methods of `PCCActivityInterface` are listed in Program 5.5-4.

Program 5.5-4 PCCActivityInterface.java (excerpt)

```
/** PCCActivityInterface. This class holds interfaces of Activities and can
  * be retrieved by using the Activity methods getResultType and getArgumentType.
class PCCActivityInterface {
   /** Create an activity interface.
     * The passed string has the following interface:<br><pre>
     * Interface = ([InterfaceEntry] {"," InterfaceEntry} ["," "..."]) | "...".
     * InterfaceEntry = Classname ["=" DefaultValue] [MayBeNull].
     * DefaultValue = "(" Value ")".
     * DefaultAccessor = "{" ActivityName "}".
     * MayBeNull = "[0]".
     * </pre>
     * Classname ... a fully qualified classname or java.lang. - Datatype
     *   (String, Integer, Boolean, etc.)<br>
     * A trailing "..." stands for an arbitrary number of arbitrary arguments<br>
     * @param iSpec a string in the given EBNF format
     */
   static public PCCActivityInterface fromString(String iSpec) {...}
   private PCCActivityInterface(String iSpec) {...}
   String asInterfaceString(Class[] typeList) {...}

   /** Get the size of this interface.
     * @return The number of defined interface entries.
     */
   int size() {...}

   /** Get the variable argument status.
     * @return true, if this interface allows a variable number of arguments
     */
   boolean hasVariableArgs() {...}

   /** Check if a PCCMaterial matches this Interface.
     * @param material the argument list or return value to check
     * @return true, if material matches this interface; false, otherwise
     */
   boolean isMaterialOK(PCCMaterial material) {...}
}
```

5.5.4 Type Safety / Implementation of perform

Activity interfaces are used to ensure type safety for arguments and return values of Activities. Materials passed as arguments and returned must match the Activity interface represented through `PCCActivityInterface` objects.

The method `isMaterialOK` (see Program 5.5-4) is called for each invocation of an Activity. If the type check for the passed arguments fails, the execution is stopped by throwing an exception of type `PCCPerformException`. The same is done if the return value type does not match the specified interface.

Program 5.5-5 Type Checks in PCCAbstractActivity

```
public class PCCAbstractActivity {
   ...
   public PCCMaterial perform(PCCSenderInfo si, PCCMaterial arg)
     throws PCCPerformException {
      if (!canPerform(arg)) {
         throw new PCCPerformException("Cannot perform.");
      }
      checkMaterialType(getArgumentType(), arg, "argument");
      notifyListenersBeforePerform(this, arg);
      PCCMaterial result = doPerform(si, arg);
      checkMaterialType(getResultType(), result, "return value");
      notifyListenersAfterPerform(this, arg, result);
      return result;
   }
}
```

The template method `perform` is responsible for checking argument and return value types using method `checkMaterialType`. `checkMaterialType` calls `isMaterialOK` on the specified Activity interface if the interface is not `null`. Before checking the argument types, the Activity is checked as to whether or not it is in an executable state (a state other than “`Disabled`”).

The following code fragment shows some erroneous return statements that lead to exceptions because of type errors, and a correct statement.

Program 5.5-6 Checking the return value type

```
public class MoveFileActivity : PCCSingleActivity {
   /** Activity constructor. In sets the dynamic type of this operation.
     * Type declaration: name "moveFile", category "File", returns a Boolean
     *   object, needs two strings, the original file path and the new one
   MoveFileActivity() { super("moveFile", "File", "Boolean", "String,String"); }
   protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args)
      throws PCCPerformException {
      boolean done = moveFile(
               (String)args.getObject(0),
               (String)args.getObject(1)));
      // return new PCCMaterial();      // exception: returns no value
      // return new PCCMAterial(17);    // exception: returns an Integer object
      // return new PCCMAterial("foo"); // exception: returns a String object
      return new PCCMaterial(done);     // ok:returns a Boolean object; equal to:
                                        //     PCCMaterial retv;
                                        //     retv.add(done);
}
```

The framework-internal type check reports no errors if a `Boolean` object is returned, as expected.

The following code fragment shows some incorrect usage of `MoveFileActivity` by passing arguments of wrong types.

Program 5.5-7 Checking the argument types

```
void testActivity() {
   PCCActivity activity = new MoveFileActivity();
   PCCMaterial result = null;
   PCCMaterial args = new PCCMaterial();
   String argumentInterface;

   PCCActivityInterface argumentTypes = activity.getArgumentType();
   System.out.println(argumentTypes.getInterface());   // prints "String,String"

   PCCActivityInterface resultType = activity.getResultType();
   System.out.println(resultType.getInterface());      // prints "Boolean"

   System.out.println(args.getInterface());            // prints ""
   result = activity.perform(null, args);   // exception: passed an empty
                                            //            argument list
   args.add("/tmp/fromFile.tmp");           // add a String object
   System.out.println(args.getInterface());            // prints "String"
   result = activity.perform(null, args);   // exception: passed only one String,
                                            //            two expected
   args.add(17);               // add an Integer object (a plain int value is
                               // automatically converted to an Integer object)
   System.out.println(args.getInterface());            // prints "String,Integer"
   result = activity.perform(null, args);   // exception: passed a String and an
                                            //            Integer, but two String
                                            //            objects are expected
   args.set(1, "/tmp/toFile.txt");          // replace the second argument
                                            // (index 1) with a String
   System.out.println(args.getInterface());            // prints "String,String"
   result = activity.perform(null, args);   // ok: passed two Strings
}
```

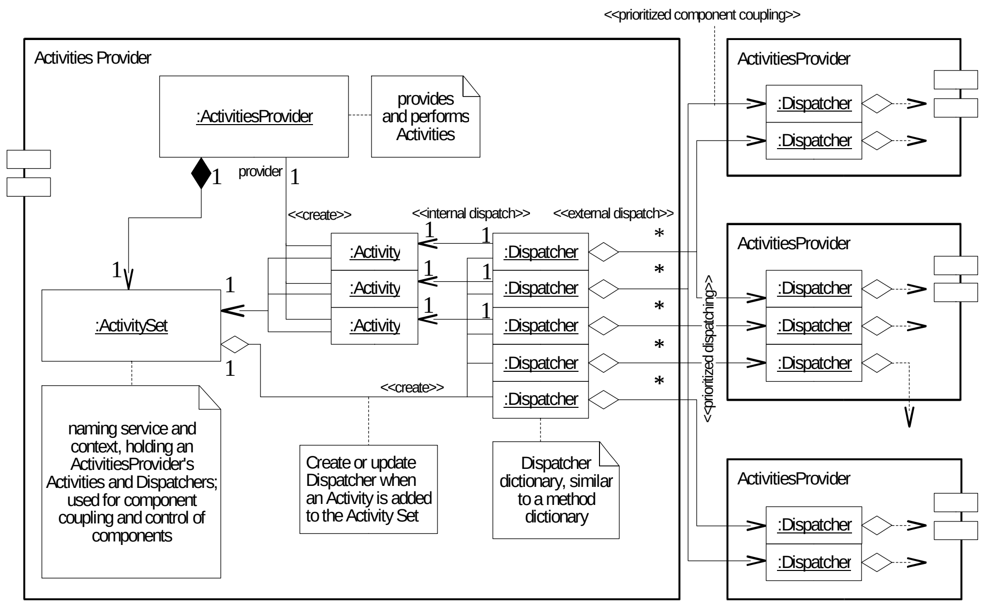
Only the last `perform` statement works, since only in this case the types of the arguments match the specification in the constructor. Print statements are used to print current interfaces.

5.6 Dispatchers and Activity Sets: Dispatching Requests

One of the most important concepts of PCoC are Dispatchers. They are used for delegating requests for the invocation of Activity requests through an acquisition graph. See Section 5.4.3.2 for an acquisition example. We assume, the reader has read Section 4.4.

5.6.1 Activity and Dispatcher Tables

An Activity Provider, respectively its Activity Set has a set of Activities and Dispatchers managed in dictionaries (key/value-maps).



<<prioritized component coupling>>
Activities Provider
:ActivitiesProvider
provides and performs Activities
provider
1
<<create>>
<<internal dispatch>>
<<external dispatch>>
:Activity
:Dispatcher
:ActivitySet
naming service and context, holding an ActivitiesProvider's Activities and Dispatchers; used for component coupling and control of components
Create or update Dispatcher when an Activity is added to the Activity Set
Dispatcher dictionary, similar to a method dictionary
ActivitiesProvider
<<prioritized dispatching>>
*

Figure 5.6-1 Component Coupling using Dispatchers

Figure 5.6-1 shows how Activities Providers can be connected via their Dispatchers. Activities Providers create Activities which are then stored in the Activity dictionary of its Activity Set. The dictionary is used, for example, for introspection. For each Activity, a Dispatcher is either created or, in the case that the Dispatcher already exists, updated (see Section 5.4.3). Dispatchers are stored in the Dispatcher dictionary. The key for looking up a Dispatcher is its (lower-case) name (or its signature, in an extended version of the framework). Remember, a Dispatcher holds a list of references to acquired Dispatchers, and an optional reference to an Activity (see Program 2.6-1).

Figure 5.6-2 illustrates the Activity and Dispatcher dictionaries of a specific Activities Provider. The tables are managed by its Activity Set. See Section 5.4.3 for a description of how Activities are added to an Activity Set, and the algorithm for creating corresponding Dispatchers.

When a Dispatcher is invoked, it uses its own Activity reference and/or gathers all Activities from directly and indirectly acquired Dispatchers in a distinct and ordered list and delegates the request to one or many Activities of the resulting set (depending on the specified directives). Note that we do not directly store acquired Activities in Dispatchers, but only references to their associated Dispatchers, in order to save memory and to keep updates simpler.

In the given example, `foo` is acquired from another Activity Set, and thus requests for performing it are in each case delegated to this Activity Set, whereas `copy` requests may be performed locally (for example, when using directive `PCCDirectives.first()`).

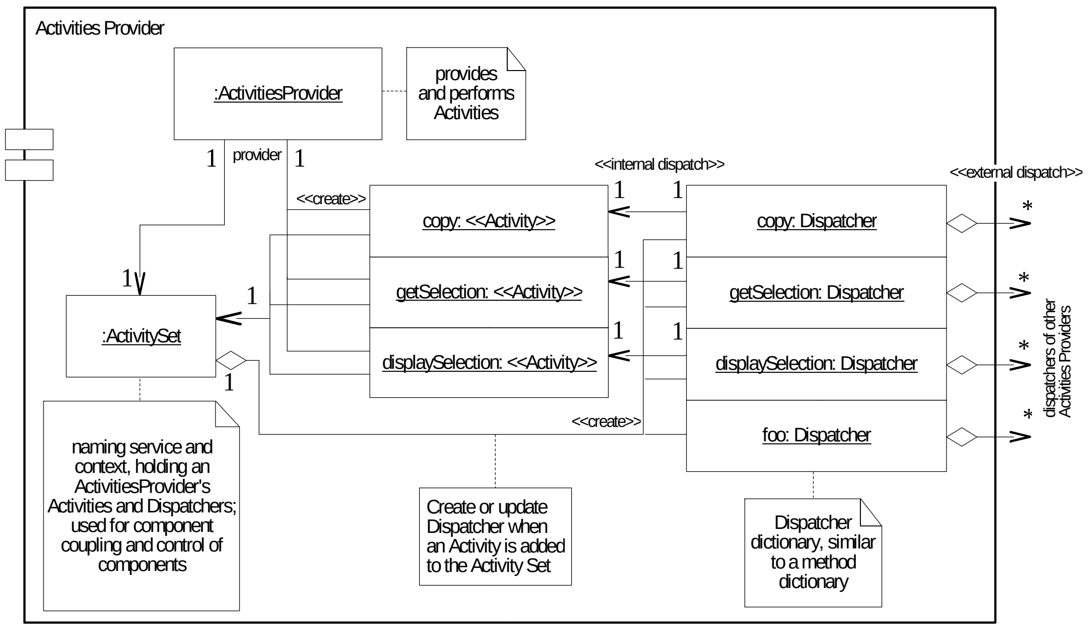

Activities Provider
:ActivitiesProvider
provides and performs Activities
provider
1
1
<<create>>
<<internal dispatch>>
<<external dispatch>>
copy: <<Activity>>
getSelection: <<Activity>>
displaySelection: <<Activity>>
copy: Dispatcher
getSelection: Dispatcher
displaySelection: Dispatcher
foo: Dispatcher
:ActivitySet
dispatchers of other Activities Providers
naming service and context, holding an ActivitiesProvider's Activities and Dispatchers; used for component coupling and control of components
Create or update Dispatcher when an Activity is added to the Activity Set
Dispatcher dictionary, similar to a method dictionary

Figure 5.6-2 Activity and Dispatcher Tables of a Component

5.6.2 Dispatching Activity Requests

Figure 5.6-3 depicts the coupling of four Activities Providers by their Dispatchers. `Application` acquires `AP1` and `AP2`. `AP1` acquires `AP11`.

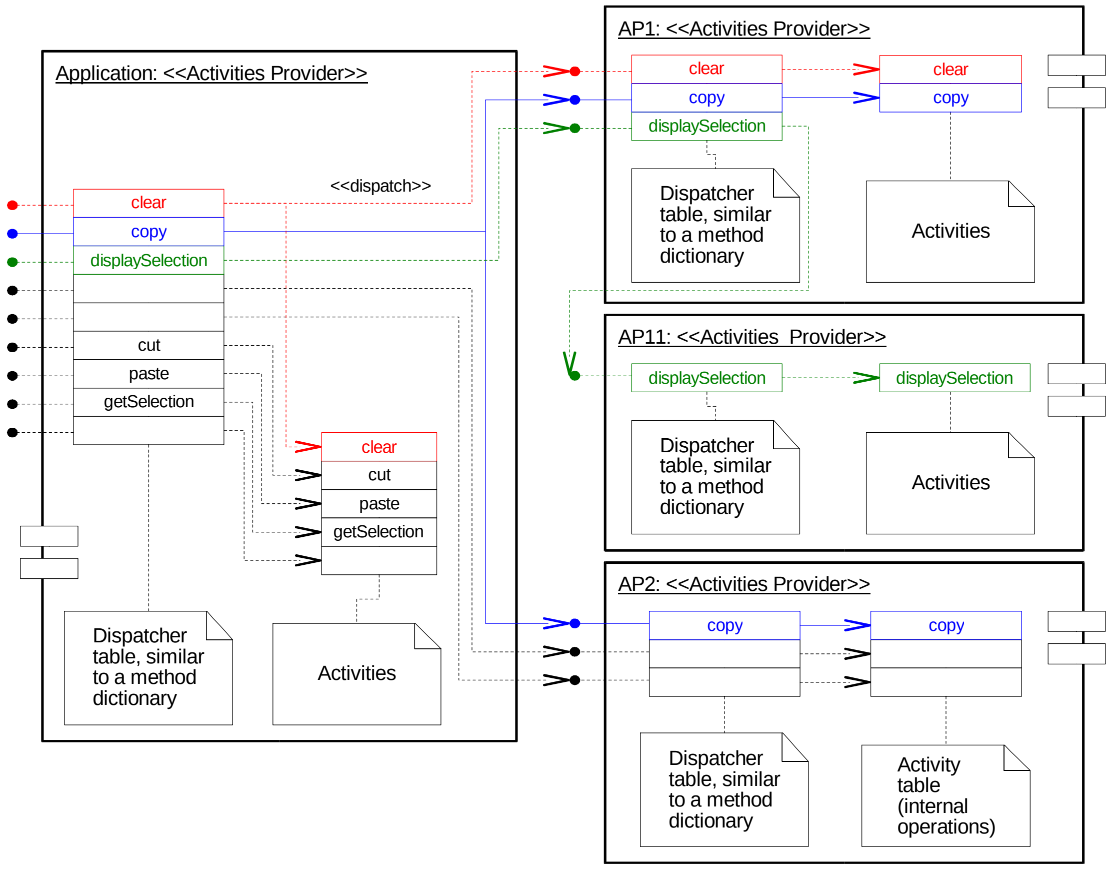

AP1: <<Activities Provider>>
Application: <<Activities Provider>>
AP11: <<Activities Provider>>
AP2: <<Activities Provider>>
<<dispatch>>
clear
copy
displaySelection
cut
paste
getSelection
Dispatcher table, similar to a method dictionary
Activities
Activity table (internal operations)

Figure 5.6-3 Dispatching Activity Requests

Some of the components in Figure 5.6-3 provide a `copy` Activity, therefore each of them has automatically created a Dispatcher for `copy`.

A request for performing `copy` in `Application` is sent via its Dispatcher to the Activities Provider, respectively Activity Set, that most recently had focus and that provides a `copy` Activity.

Program 5.6-1 Invoking a Dispatcher

```
Application app = new Application();
app.invoke("copy", null, PCCDirectives.first());
app.getActivitySet().invoke("copy", null, PCCDirectives.first());
```

This code fragment shows the invocation of the `copy` Dispatcher. The last two statements are semantically equivalent.

In this case, the request might be sent to `AP1`, i.e., the Dispatcher algorithm follows its first branch to `AP1`, gets the `copy` Activity there, and invokes its `perform` method in a second step. We say that a `perform` request is sent to the `copy` Activity of `AP1` in this case. On the other hand, `getSelection` would be performed directly in `Application`.

A request for performing `displaySelection` would always be sent to `AP11` over `AP1`, since `AP11` is the only component that provides this Activity.

Program 5.6-2 Dispatch algorithm

```
final public class PCCDispatcher extends PCCAbstractActivity
                                 implements PCCActivityListener {
   /** finds activities starting the search at the current dispatcher and
     * using a given strategy. Found activities are stored in the strategy object.
     * This framework-internal method is mainly used for broadcasts.
     * @param strat the strategy used for searching activities
     */
   void findMatchingActivities(ActStrategy strat) {
      PCCDirectives directives = strat.getDirectives();
      if (fActivity != null) {
         strat.addActivity(fActivity);
         if (!directives.sendsDeep()) { return; }
      }
      PCCDispatcherImpl activity = null;
      if (directives.isBroadcast()) {
         int sz = fDispatchers.size();
         int i = 0;
         int endv = sz;
         int offset = 1;
         if (directives.sendsBackward()) {
            i = sz-1;
            endv = 0;
            offset = -1;
         }
         while (i != endv) {
            activity = (PCCDispatcherImpl) fDispatchers.get(i);
            if (activity != null &&
              (!directives.sendsToActiveOnly() ||
              activity.getActivitySet().isActive())) {
                activity.findMatchingActivities(strat);
            }
            if (i == endv) {  // i is an int, therefore we must check the value
               break;         // before getting an underrun with i--
            }
            i += offset;
         }
      }
   }
...
}
```

This code fragment shows the algorithm for gathering Activities through all acquisition branches of a Dispatcher.

Basically, the dispatch algorithm of PCoC consists of two phases: the search and the execution.

- Follow one or many acquisition branches of the given Dispatcher in the given order (priority) and collect all Activities belonging to the given Dispatcher in a distinct and ordered list. This can be done independently of Activity Sets, since Dispatchers store the references to Activities and acquired Dispatchers with the same name. The resulting set has no duplicate Activities. In the case of a single request, for example using specifier `first`, only the branch with the highest priority is visited. In the case of broadcasts all acquisition branches are visited in the given order.
- Invoke a method on each Activity in the retrieved set, e.g. `getState`, `perform`, etc.

The directives are used to filter the search while iterating over the list of acquired Dispatchers (`fDispatchers`). If an Activity (`fActivity`) is found, it is added to the resulting strategy object.

Program 5.6-3 is an excerpt of the base class for strategy objects passed to the dispatch algorithm (cf. Program 5.6-2).

Program 5.6-3 Dispatch-strategy class (1)

```
abstract class ActStrategy implements PCCActivityListener {
   ActStrategy(boolean robust, PCCMaterial arg, PCCDirectives directives,
     String filter, ArrayList activities) {
      ...
   }
   /** performs this strategy on each activity found by
     * <code>findMatchingActivities</code> using this strategy
     */
   Object perform() {
      int i = 0;
      while (i < fActivities.size()) {
         PCCActivity activity = (PCCActivity) fActivities.get(i);
         if (activity != null) { doPerform(activity); }
         i++;
      }
      performed();
      return fResult;
   }
}
...
```

The method `perform` is called by client code and represents the actual strategy. Subclasses must override the method `doPerform`.

Program 5.6-4 Dispatch-strategy class (2)

```
class PerformStrategy extends ActStrategy {
   PerformStrategy(PCCMaterial arg, PCCDirectives directives, String filter) {
      super(true, arg, directives, filter, null);
   }
   /** invokes a method corresponding to this strategy class on a single activity.
     * @param activity the activity to be performed
     */
   protected void doPerform(PCCSenderInfo si, PCCActivity activity) {
      try { fResult = activity.perform(si, (PCCMaterial) fArg); }
      catch (PCCPerformException pe) { fResult = pe; }
   }
}
```

In this case, the `perform` method of every single Activity is invoked. Beside `PerformStrategy`, there are several other strategy classes, one for each Activity request that a Dispatcher can delegate: `getState`, `countActivities`, `getContextData` (see Section 4.4), etc.

Program 5.6-5 Using a dispatch-strategy class

```
final public class PCCDispatcher extends PCCAbstractActivity
                                 implements PCCActivityListener {
   public void perform() {
      PerformStrategy strat = new PerformStrategy(arg, directives, filter);
      findMatchingActivities(strat);
      Object result = strat.perform();
   }
}
```

Having different strategy classes, the Dispatcher can reuse the same algorithm for different things.

Using strategies instead of duplicating an algorithm is a good way to reduce the danger of making mistakes during maintenance or extension of the algorithm. It also reduces the maintenance effort, since the really difficult code is just in one place and has only to be maintained there.

5.7 Tasks in Detail

We assume, the reader has read Section 4.5 as introduction to the concept of Tasks. Now we go more into detail.

5.7.1 Structure of Tasks

Tasks are usually defined via configuration. The following excerpt of a configuration file shows a definition of a Task (basically equivalent to Program 4.5-1).

Program 5.7-1 Defining a Task via configuration

```
<RCTask>
   <name>displaySelectionTask</name>
   <dispatcher directives="First">|Application|displaySelection</dispatcher>
   <args>
      <dispatcher>|Application|getSelection</dispatcher>
      <integer>2</integer>
   </args>
   <preconditions>
      <dispatcher>|Application|foo</dispatcher>
      <task>myCondition</task>
   </preconditions>
...
</RCTask>
```

See Section 4.5 for a description of this sample configuration.

Program 5.7-2 Using a Task

```
public class MyActivitiesProvider extends PCCServiceProvider {
...
   public void foo() {
      // Get or create a Task; calls getActivitySet().getOrCreateTask(...);
      PCCTask task = getOrCreateTask("displaySelectionTask");
      if (task.canPerform()) { // if state != disabledState(), and all
         task.perform();       // Dispatchers are available and executable, and
      }                        // all arguments are provided via configuration,
                               // and the types of the passed arguments match
   }                           // the dynamic type of the Dispatcher
}
```

In this example, we use the method `getOrCreateTask`, provided by `PCCActivitiesProvider` and `PCCActivitySet`, to create a task.

PCoC creates an instance of the `PCCTask` class and initializes the object with the given `RCTask` definition.

Program 5.7-3 Using a Task

```
public final class PCCActivitySet implements PCCActivitySetListener {
...
   /** Create new or get existing Task associated with the given name. If no
     * Task instance with this name exists in this Activity Set, it is created
     * and initialized with the corresponding RCTask definition from the
     * configuration. References to already created Tasks are stored in a
     * hash map (fTaskMap) in this Activity Set.
     * @param name The name of the item used to get or create a task.
     */
   public PCCTask getOrCreateTask(String name) {
      PCCTask task = (PCCTask) fTaskMap.get(name.toLowerCase());
      if(task == null) {
         task = createAndRegisterTask(ResConfigMap.getTask(name));
      }
      return task;
   }

   public PCCTask createAndRegisterTask(RCTask item) {
      if (item == null) { return null; }
      PCCTask task = new PCCTask(this, item);
      if(task.isUnusable()) {  // check if the arguments match the dynamic type
         task.cleanup();
         task = null;
      }
      else { fTaskMap.put(item.getName().toLowerCase(), task); }
      return task;
   }
}
```

Note that `ResConfigMap` is a singleton that holds a representation of the whole XML-configuration of a PCoC application. All definitions are available as objects in `ResConfigMap` and can be looked up using their name. The name is the value enclosed in the corresponding name-tag of the XML-configuration. See Program 5.7-1 (`<name>displaySelectionTask</name>`).

Program 5.7-4 PCCTask constructor

```
/** create a PCCTask from RCTask
  * @param as the Activity Set the Task belongs to
  * @param rcTask defines the Task (name, associated dispatcher, arguments, etc.)
  */
PCCTask(PCCActivitySet as, RCTask rcTask) {
   String name = rcTask.getName();
   String associatedDispatcher = null;
   String directives = null;
   RCDispatcherReference dispRef = rcTask.getDispatcher();
   if(dispRef != null) {
      associatedDispatcher = dispRef.getContent(); // set dispatcher, can be null
                                                    // for macros
      directives = dispRef.getDirectives();        // set dispatcher directives
   }
   PCCMaterial args = null;
   RCArray rcArgs = rcTask.getArgs();
   args = convertRCArrayToMaterial(rcArgs);        // set arguments
   setupConditions(rcTask);                        // set up preconditions
                                                    // for performing this Task
   // initialize member variables, and macro (if this Task represents a macro)
   init(as, name, associatedDispatcher, args, directives, rcTask);
   ...
}
```

The implementation of the `PCCTask` constructor is shown above. This code fragment should help to get a rough overview of how the configuration and instances of `PCCTask` are related to each other. We do not show the whole implementation of the `PCCTask` class, since it is too huge to be described here, and it does not add relevant information to get more insight into the concepts of the framework.

The member variables (`fActivitySet`, `fName`, `fArguments`, etc.) of the Task are set in `init`. The values are taken from the configuration passed to the constructor.

Arguments are converted in a method `convertRCArrayToMaterial` from a string representation to corresponding primitive data-type objects. For example, a value following an `<integer>` tag in the configuration (Program 5.7-1) is converted to an `Integer` object, etc. The method also checks whether the argument types match the dynamic type of the main Dispatcher of the Task (if there is any). If an argument is a reference to a Dispatcher or Task, the referenced Dispatcher or Task is performed and temporarily replaced by its return value when the current Task is performed.

For each precondition (if there is any), a corresponding reference to a Task object or Dispatcher is stored in an `ArrayList` (`fPreConditions`) using the method `setupConditions`. When the Task is performed, for each Task and Dispatcher in this list the method `canPerform` is called in order to check whether the Task can be performed.

5.7.2 Dispatching Requests using Tasks

Figure 5.7-1 illustrates how Dispatchers of a Task delegate Activity requests to different acquired Activities Providers. Note that the prefix `PCC` of class names are omitted in this diagram (`PCCDispatcher`, `PCCActivity`, etc.)

We assume that the Activity Set of an Activities Provider `Application` acquires three others, `AP1`, `AP2`, and `AP3`.

The Task `displaySelectionTask` is defined in the context of the `Application` Activities Provider. The two Dispatchers associated with this Task, `displaySelection` and `getSelection`, delegate requests for retrieving states and performing Activities to the acquired Activity Sets.

Requests are always delegated to the Activities Provider, respectively Activity Set with the highest priority capable of handling the request. In the case of a broadcast (specified as Dispatcher directive), all capable Activities Providers are addressed in the order of their priority ranking.

Let us assume that we have the following priority ranking in the list of acquired Activity Sets in `Application`: `AP2`, `AP3`, `AP1`.

In this case, `displaySelection` requests are delegated to `AP2` while `getSelection` requests are delegated to `AP3`. Note that the priority ranking can change at any time.

The return value of `getSelection` is used as argument for the invocation of the `displaySelection` Dispatcher. We say, `getSelection` is a data source, and `displaySelection` the data sink for `displaySelectionTask`.

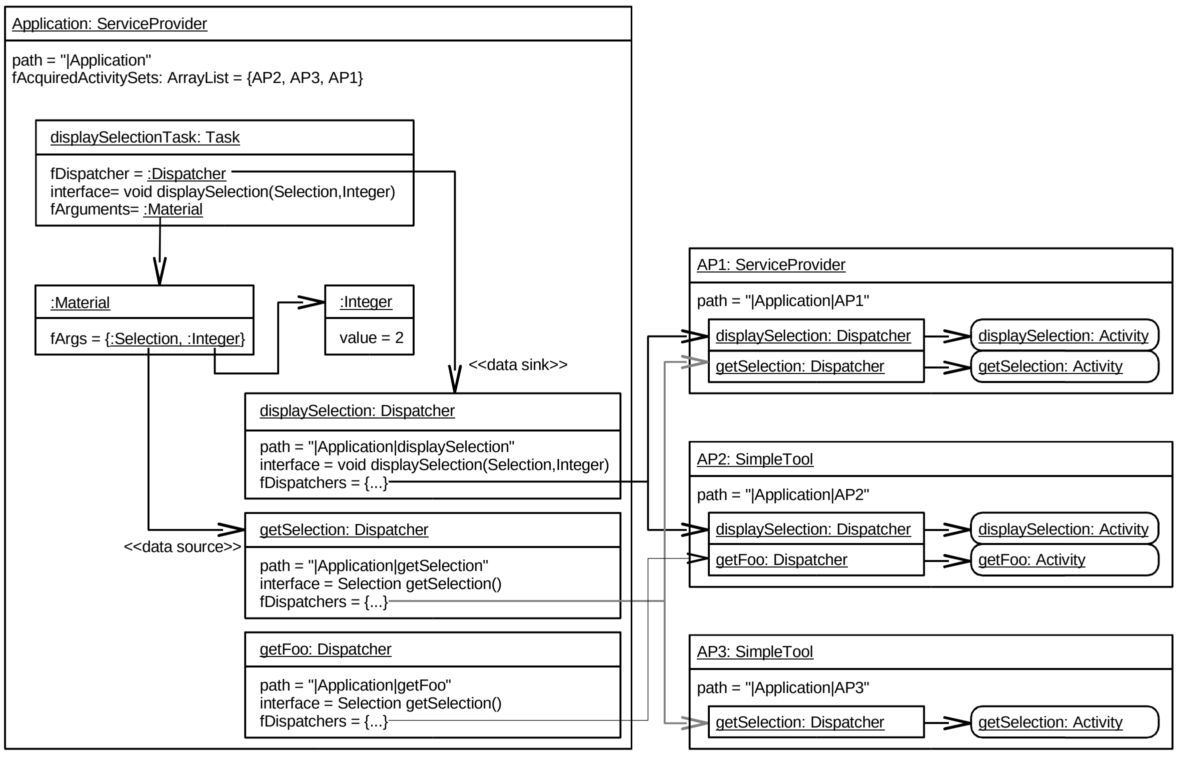

Application: ServiceProvider
path = "|Application"
fAcquiredActivitySets: ArrayList = {AP2, AP3, AP1}
displaySelectionTask: Task
fDispatcher = :Dispatcher
interface= void displaySelection(Selection,Integer)
fArguments= :Material
:Material
fArgs = {:Selection, :Integer}
:Integer
value = 2
<<data sink>>
displaySelection: Dispatcher
path = "|Application|displaySelection"
interface = void displaySelection(Selection,Integer)
fDispatchers = {...}
<<data source>>
getSelection: Dispatcher
path = "|Application|getSelection"
interface = Selection getSelection()
fDispatchers = {...}
getFoo: Dispatcher
path = "|Application|getFoo"
interface = Selection getSelection()
fDispatchers = {...}
AP1: ServiceProvider
path = "|Application|AP1"
displaySelection: Dispatcher
displaySelection: Activity
getSelection: Dispatcher
getSelection: Activity
AP2: SimpleTool
path = "|Application|AP2"
displaySelection: Dispatcher
displaySelection: Activity
getFoo: Dispatcher
getFoo: Activity
AP3: SimpleTool
path = "|Application|AP3"
getSelection: Dispatcher
getSelection: Activity

Figure 5.7-1 A Task Coupling Components With Dispatchers

5.8 Activity Sets: Putting Everything Together

We have seen how Activities Providers expose Activities, how Activities are invoked via requests from Dispatchers, and how Activities Providers collaborate using these elements. We have also seen how Activities Providers are organized in dynamic containment hierarchies and acquisition relationships.

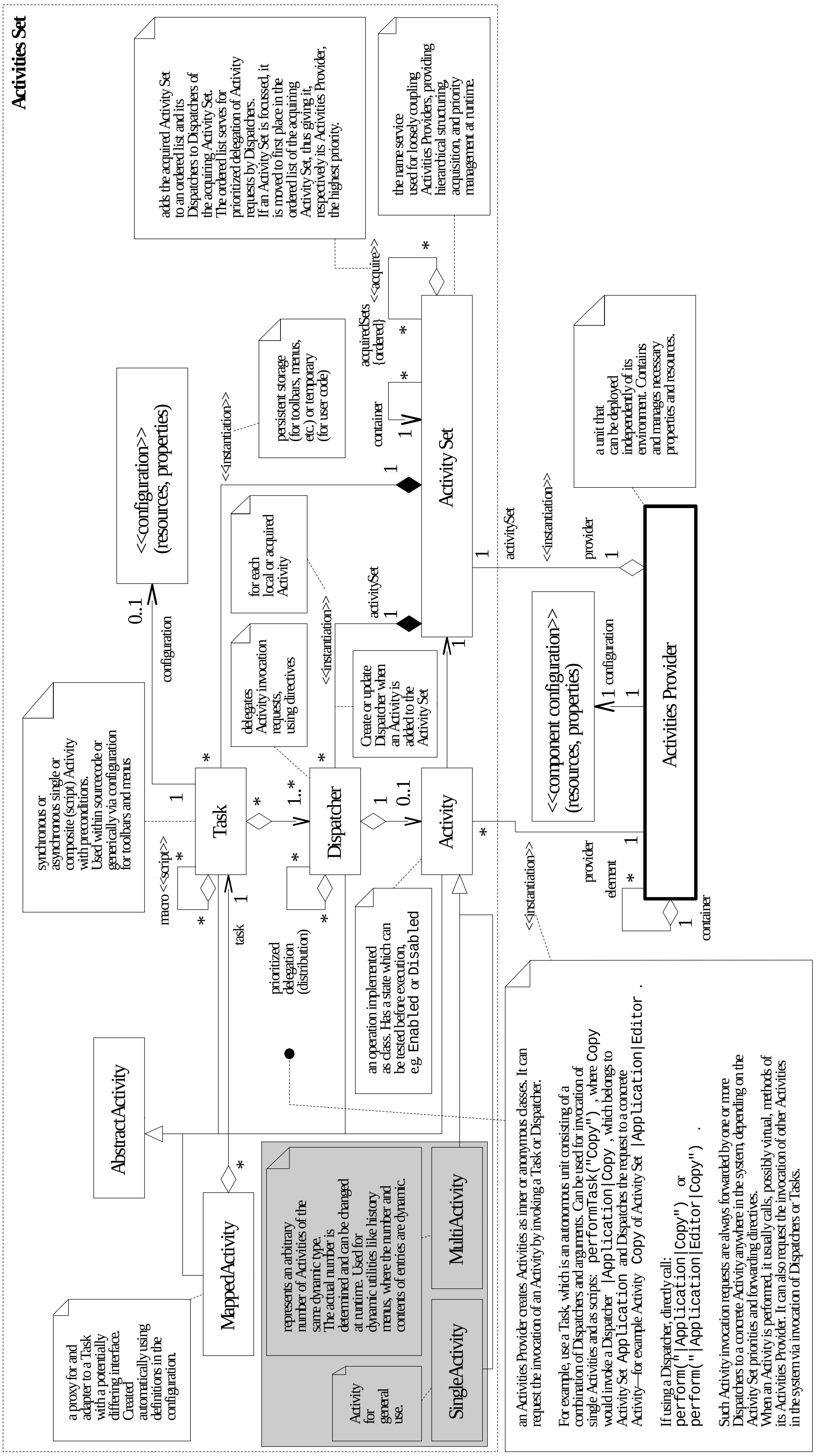
Activity Sets are responsible for keeping all dynamic elements and structural information together. They are created automatically and invisibly to component developers by Activities Providers when they are initialized. They can also be created independently of Activities Providers, simply providing sets of operations.

Please read Section 3.3 for a short introduction into Activity Sets.

5.8.1 Architecture Overview

Figure 5.8-1 illustrates how Activity Set, Activities Provider, Activity, and Dispatcher classes are related to each other, and roughly explains the meaning of the classes.

See Sections 5.3 and 5.4.3 for the most relevant facilities supported by Activity Set: containment hierarchy and prioritized acquisition.



a proxy for and adapter to a Task with a potentially differing interface. Created automatically using definitions in the configuration.

Activity for general use.

represents an arbitrary number of Activities of the same dynamic type. The actual number is determined and can be changed at runtime. Used for dynamic utilities like history menus, where the number and contents of entries are dynamic.

synchronous or asynchronous single or composite (script) Activity with preconditions. Used within sourcecode or generically via configuration for toolbars and menus

delegates Activity invocation requests, using directives

for each local or acquired Activity

Create or update Dispatcher when an Activity is added to the Activity Set

an operation implemented as class. Has a state which can be tested before execution, e.g. `Enabled` or `Disabled`

persistent storage (for toolbars, menus, etc.) or temporary (for user code)

adds the acquired Activity Set to an ordered list and its Dispatchers to Dispatchers of the acquiring Activity Set. The ordered list serves for prioritized delegation of Activity requests by Dispatchers. If an Activity Set is focussed, it is moved to first place in the ordered list of the acquiring Activity Set, thus giving it, respectively its Activities Provider, the highest priority.

the name service used for loosely coupling Activities Providers, providing hierarchical structuring, acquisition, and priority management at runtime.

a unit that can be deployed independently of its environment. Contains and manages necessary properties and resources.

an Activities Provider creates Activities as inner or anonymous classes. It can request the invocation of an Activity by invoking a Task or Dispatcher.

For example, use a Task, which is an autonomous unit consisting of a combination of Dispatchers and arguments. Can be used for invocation of single Activities and as scripts: `performTask("Copy")` , where Copy would invoke a Dispatcher `|Application|Copy` , which belongs to Activity Set `Application` and Dispatches the request to a concrete Activity—for example Activity `Copy` of Activity Set `|Application|Editor` .

If using a Dispatcher, directly call:
`perform("|Application|Copy")` or
`perform("|Application|Editor|Copy")` .

Such Activity invocation requests are always forwarded by one or more Dispatchers to a concrete Activity anywhere in the system, depending on the Activity Set priorities and forwarding directives.
When an Activity is performed, it usually calls, possibly virtual, methods of its Activities Provider. It can also request the invocation of other Activities in the system via invocation of Dispatchers or Tasks.

Figure 5.8-1 Activity Set Architecture

5.8.2 Dispatching Activity Requests in Activity Sets

Program 5.8-1 Activities Provider methods

```
public class PCCActivitiesProvider {
   /** Lets a Dispatcher perform an Activity using forwarding
     * directives.
     * @param dispatcher path and name of an activity Dispatcher,
     *        e.g. "|Application|Copy"
     * @param arg The argument for the Dispatcher to perform.
     * @param directives the forwarding directives for the Dispatcher,
     *        e.g. PCCDirectives.defaultDirectives()
     * @return The result of the performed activity
     */
   public PCCMaterial perform(String dispatcher, PCCMaterial arg,
                              PCCDirectives directives)
          throws PCCPerformException {
      if (!isInitialized()) { return null; }
      return getActivitySet().perform(dispatcher, arg, directives);
   }
}
```

`PCCActivitiesProvider` provides a `perform` method which is used to send a request for the execution of an Activity through the acquisition graph, starting at a specific Dispatcher. The method calls the corresponding method in its Activity Set if it is created and initialized.

Program 5.8-2 Activity Set methods

```
public final class PCCActivitySet implements PCCActivitySetListener {
   /** Lets a Dispatcher invoke (forward) an Activity using forwarding directives.
     * @param dispatcher path and name of a Dispatcher, e.g. "|Application|Copy"
     * @param arg The argument for the Dispatcher to invoke.
     * @param directives the forwarding directives for the Dispatcher,
     *        e.g. PCCDirectives.defaultDirectives()
     * @return The result of the performed Activity
     */
   public PCCMaterial perform(String dispatcher, PCCMaterial arg,
                              PCCDirectives directives)
     throws PCCPerformException {
      PCCDispatcherImpl dispatcherImpl = getDispatcherImpl(dispatcher);
      if (dispatcherImpl != null) {
         return dispatcherImpl.perform(new PCCSenderInfo(this), arg,
                                       directives, null);
      }
      else {
         throw new PCCPerformException("No Dispatcher '" + dispatcher + "'");
      }
   }
...
}
```

The user passes a path that describes a Dispatcher. This includes the path of the Activities Provider, respectively its Activity Set, where the Dispatcher should be retrieved, and its name. For example, "`|Application|copy`" means Dispatcher "`copy`" in the Activities Provider with path "`|Application|`".

Note, if no "|" is specified as first character, the path is considered relative to the path of the current Activity Set. For example, calling `perform` with path "`TextEditor[0]|copy`" in an Activity Set "`|Application`" is resolved to "`|Application|TextEditor|copy`".

5.8.3 Dispatcher Proxy Management

In Program 5.8-2 a `DispatcherImpl` instance is used to forward or delegate requests for executing Activities. A `DispatcherImpl` instance provides the behavior of a Dispatcher, whereas instances of the class `Dispatcher` are proxies to `DispatcherImpl` instances.

Beside using the `perform` method of an Activities Provider or Activity Set

Program 5.8-3 Invoking an Activity

```
perform("|Application|copy", null, PCCDirectives.first());
```

we can use Dispatcher proxies to invoke Activities

Program 5.8-4 Invoking an Activity using a Dispatcher proxy

```
Dispatcher dispatcher = getDispatcher("|Application|copy");
dispatcher.perform(null, PCCDirectives.first());
```

Program 5.8-3 and Program 5.8-4 are equivalent. The Dispatcher proxy (Program 5.8-4) can be stored for later use regardless of whether the underlying `DispatcherImpl` instance exists or not. So we can attach listeners to the Dispatcher proxy without caring about the lifetime of the corresponding `DispatcherImpl` instance. A Dispatcher proxy sends notifications whenever its `DispatcherImpl` instance is created or destroyed, and it forwards all notifications that its `DispatcherImpl` provides, e.g., state changes of Activities.

Program 5.8-5 Dispatcher lookup (class PCCActivitySet)

```
/** Gets a Dispatcher.
  * @param dispatcher the name or path of the requested Dispatcher
  * @return the Dispatcher; null if not found
  */
public PCCDispatcher getDispatcher(String path) {
   int lastsep = path.lastIndexOf(fcPathSeparatorChar);
   if (lastsep >= 0) {
      PCCActivitySet as = null;
      String asName = path.substring(0, lastsep);
      as = getOrCreateChild(asName); // get Activity Set associated with given path
      if (as != null) { return as.getDispatcher(path.substring(lastsep+1)); }
      return null;
   }
   String lowername = path.toLowerCase();
   DispatcherTag dispatcherTag = (DispatcherTag) fDispatcherMap.get(lowername);
   if (dispatcherTag == null) {
      dispatcherTag = new DispatcherTag();
      fDispatcherMap.put(lowername, dispatcherTag); // create new Dispatcher entry
   }
   PCCDispatcher dispatcher = null;
   if (dispatcherTag.fDispatcher != null) {   // get the corresponding Dispatcher
      dispatcher = (PCCDispatcher)dispatcherTag.fDispatcher.get();
   }                                          // (proxy), if any
   if (dispatcher == null) {                  // create new one if necessary
      dispatcher = new PCCDispatcher(this, path);
      dispatcherTag.fDispatcher = new WeakReference(dispatcher);
      if (!dispatcherTag.fRemoved) {          // add a reference to the
                                              // DispatcherImpl, if not
         dispatcher.activityAdded(dispatcherTag.fDispatcherImpl);
      }                                       // removed since the last change
   }                                          // propagation of this Activity Set
   return dispatcher;
}
```

For example, for some reason, we might want to use and store a Dispatcher "`|Application|FileSystemService|getDirectories`" in one of our classes, whilst the corresponding Activities Provider (`FileSystemService`) and Activity (`getDirectories`) have not been loaded, yet. When the Activities Provider and its Activity are instantiated, a `DispatcherImpl` object is created and associated with the Activity and a possibly existing Dispatcher. The Dispatcher is then notified by its `DispatcherImpl`. Finally, the Dispatcher notifies its listeners. Let us assume that our object holding a reference to the

Dispatcher is a listener of it. The object might instantly invoke the Dispatcher after the Activity is created (for example, in order to display the resulting directories in a list view, or to do some initialization or synchronization of data between various Activities Providers and the newly loaded one, etc.).

`Dispatcher` instances are stored in the same table as `DispatcherImpl` instances. They are stored and looked up using their lower case names (see Program 5.8-5).

If no Dispatcher with the given name has ever been looked up before, an instance is created. It is stored in a `WeakReference`, therefore it will be cleaned up by the garbage collector as soon as it is not used by client code anymore. We assume that the reader is familiar with the `WeakReference` class of Java.

If there is a corresponding `DispatcherImpl` for the requested `Dispatcher` instance, the framework adds a reference to it using `activityAdded`. If there is no `DispatcherImpl` instance, then the reference is set whenever an instance is created and added to the Activity Set.

5.9 Implementation Issues

5.9.1 Java or C++?

As mentioned earlier in this thesis, the PCoC framework was implemented in C++ and later ported to Java. The reason for the migration to Java was that Java provides a platform that is available on many operating systems. This includes a sophisticated standard set of class libraries. Usually, class libraries either have to be bought and integrated with others in an application, or implemented from scratch. There are also open-source class libraries, which can be adapted, integrated, and used.

However, the selection, integration, implementation, and maintenance of class libraries are expensive and ongoing tasks. Particularly, the training of new developers is expensive and time-consuming. Hence we decided to move to Java, which provides a commonly known and accepted set of libraries, respectively packages.

5.9.2 Supporting Implicit Generation of Activities

Activities can be generated implicitly from method definitions. We had not to implement Activities, since, for example, all public method would be automatically exposed as Activities. The implementation of the generic Activity class could look like Program 5.9-1.

The constructor takes a method object as argument and initializes the Activity with the method name and its result- and parameter-types.

`asInterfaceString` generates interface specification strings for the parameter type list and the return value type of the given method. The implementation of this method is omitted, since it does not add relevant information to this example. See Sections 4.3.4 and 5.5.1 for descriptions of the dynamic Activity type concept and the `PCCActivityInterface` class.

When the Activity is invoked, its arguments are unpacked and passed to the method. The method is invoked by using dynamic method invocation (`perform`).

Program 5.9-1 A generic Activity

```
class GenericActivity extends PCCSingleActivity {
   Method performMethod;
   Method stateMethod;
   GenericActivity(Method m, Method stateMethod) {
      super(m.getName(), "Generated",
            PCCActivityInterface.asInterfaceString(m.getReturnType()),
            PCCActivityInterface.asInterfaceString(m.getParameterTypes()));
      this.performMethod = m;
      if (stateMethod != null) { // check method interface
         Class[] parameterTypes = stateMethod.getParameterTypes();
         Assert.assert("Wrong state method interface. No parameters expected.",
            (parameterTypes == null || parameterTypes.length == 0));
         Class returnType = stateMethod.getReturnType();
         Assert.assert("Wrong state method interface. Must return a string.",
            (returnType != null && returnType == String.class));
         }
      }
      this.stateMethod = stateMethod;
   }
   protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args)
     throws PCCPerformException {
      Object result = null;
      Class cl = fProvider.getClass(); // provider is set in addActivity
      Object[] arguments = args.toArray();
      try { result = performMethod.invoke(fProvider, arguments); }
      catch ...      // NoSuchMethodException, IllegalAccessException,
                     // InvocationTargetException
      return new PCCMaterial(result);
   }
   protected String doGetState() {
      if (stateMethod != null) {
         String result = "";
         Class cl = fProvider.getClass()
         try { result = (String)stateMethod.invoke(fProvider, new Object[] {}); }
         catch ...   // catch exceptions as above
         return result;
      }
      return defaultState(true); // is always "Enabled" in this case
   }
...
}
```

The framework retrieves methods by using `getMethods` of the Java reflection package:

Program 5.9-2 Automatically generating Activities

```
public class PCCActivitiesProvider {
   ...
   bool addMethodsAsActivities() {
      // create Activities from public methods
      Method[] methods = getClass().getMethods();
      int i=0;
      while (i < methods.length) {
         PCCActivity activity = new GenericActivity(methods[i], null);
         addActivity(activity);  // calls also activity.setProvider(this);
         i++;   // search the first method with the given name
      }
   }
}
```

The framework iterates over all public methods of the Activities Provider, and adds them as Activities by using `addActivity`.

This approach looks comfortable for developers, since it is not necessary to implement, instantiate, and add Activities explicitly. However, we threw away this approach. There are several reasons for not generating Activities automatically:

- We need a rule to determine which methods should be exposed as Activities and which not, in order to limit the number of allocated objects. We usually only want methods to be exposed, which are reasonable for component collaboration and control.

  One solution is to let the framework only generate Activities from public methods with a certain name prefix (*programming by convention*), but this might not be appropriate for all use cases. See Program 5.9-3.
- It might cause security problems if all methods were available as Activities, since they would be automatically accessible via all application interfaces. All Activities would be exposed, for example, via XMLRPC. This might include also file system operations. Java provides a security mechanism, but if a method is forwarded via JNI to native code in C++, then there is definitely a security problem.
- If we do not create Activities for each method, then heuristics or other, maybe semi-automatic mechanisms must be used to select the methods for which Activities have to be created. This makes it almost impossible for the component developer to get and retain an overview of which Activities will be available at runtime.
- “There is a lot of magic inside”. Automatic mechanisms can make the implementation and maintenance of components more difficult. Even without automatic mechanisms, a dynamic approach such as PCoC may make the development of client code more difficult without proper utilities. For example, searching for errors in the behavior of a component is almost impossible, if we do not have an overview of which Activities and Activity Sets are created at runtime. However, there are two approaches that can help us with this problem:
  - The framework provides an introspection mechanism that allows the browsing of Activity Set hierarchies and acquisition relationships, including Activities, Dispatchers, Tasks, etc., per Activity Set. This helps us to get an overview of available Activities at runtime.
  - If we implement each Activity in a separate class, we can browse all Activities at development time with source code and class browsers. The implementation effort is slightly higher in this case.
- We could use JavaBeans, but they do not have a built in support for state handling. Activities have a state which can be set at runtime. Examples are “Enabled”, “Disabled”, “Active”, etc. This does not mean that JavaBeans cannot be used together with Activities; on the contrary, an Activity can be used to control a JavaBean or vice versa.

The advantage of supporting implicit generation of Activities is the reduced implementation effort.

Program 5.9-3 Automatically generating Activities, using a naming convention

```
public class PCCActivitiesProvider {
   ...
   bool addMethodsAsActivities() {
      // create Activities from public methods
      Method[] methods = getClass().getMethods();
      int i=0;
      while (i < methods.length) {
         if (methods[i].getName().substring(0,3).equals("pcc")) {
            PCCActivity activity = new GenericActivity(methods[i], null);
            addActivity(activity);  // calls also activity.setProvider(this);
         }
         i++;   // search the first method with the given name
      }
   }
}
```

As mentioned, a better approach is to limit the implicit generation of Activities through a naming convention (using a name prefix), but it nevertheless is not satisfying. For completeness, the corresponding implementation is illustrated above.

The framework iterates over all public methods with prefix “pcc” of the Activities Provider, and adds them as Activities by using `addActivity`.

5.9.3 Explicit and Semi-automatic Generation of Activities

After many discussions with framework users we came to the conclusion that, although the implicit generation of Activities is comfortable, the cross-relations in the source code are easier to understand if the creation of Activities is driven by the developer.

The following aspects must be considered:

- Explicit mechanisms are more accepted by PCoC users than implicit ones. Implicit features are fine as long as they do exactly what is desired. As soon as there is unexpected or unwanted behavior, it is a huge effort to find out how to change it. For example, it is difficult to find out at design time which Activities or better Activity instances will be generated at runtime. Other examples are special features (such as personalized menus) newly introduced to an operating system. For a user, it is difficult to switch a feature off, if he does not know yet, how the feature is actually called and where he can find proper information (without reading a huge user manual).
- Current PCoC users are content with the explicit definition of Activities. There is little reason why we should change a successful system.

The solution we chose was to support explicit definition of Activities (see Program 5.9-4) and user-driven generation of Activities (see Program 5.9-5 and Program 5.9-6).

The following code fragment shows the explicit definition of an Activity:

Program 5.9-4 An Activity (explicit definition)

```
public class FileSystemProvider extends PCCServiceProvider {
   ...
   public void foo() {
      addActivity(new MoveFileActivity());
   }

   class MoveFileActivity extends PCCSingleActivity {
      MoveFileActivity() {
         super("moveFile", "File", "Boolean", "String,String");
      }
      protected PCCMaterial doPerform(PCCSenderInfo si, PCCMaterial args)
         throws PCCPerformException {
         boolean done = moveFile(
                  (String)args.getObject(0),
                  (String)args.getObject(1)));
         return new PCCMaterial(done);
      }
      protected String doGetState() {
         return defaultState(true); // is always "Enabled" in this case
      }
   }
}
```

Activities can be generated from a method using reflection:

Program 5.9-5 Semi-automatic generation of Activities

```
public class FileSystemProvider extends PCCServiceProvider {
   ...
   public void foo() {
      addActivity("moveFile");
   }
   public boolean moveFile(String, String) { ... } // some code
}
```

The method will be called in the `doPerform` method of the Activity.

If state handling is required for a generated Activity, a separate method can be specified:

Program 5.9-6 An Activity(with an additional state-retrieval method)

```
public class FileSystemProvider extends PCCServiceProvider {
   ...
   public void foo() {
      addActivity("moveFile", "getMoveFileState");
   }
   public boolean moveFile(String, String) { ... } // some code
   public String getMoveFileState() {
      return enabledState();  // equivalent to "Enabled"
   }
}
```

The additional method will be invoked in the `doGetState` method of the Activity.

The advantage of this approach is that a component developer can explicitly specify the methods to be exposed as Activities. For each Activity, no matter if implicitly generated or explicitly defined, a Dispatcher is created.

`addActivity` is a convenience method of `PCCActivitiesProvider`. It generates an Activity from a specified method, adds it to the Activities Provider’s Activity Set, and does some additional checks. See the implementation below:

Program 5.9-7 Explicitly generating Activities

```
public class PCCActivitiesProvider {
   ...
   bool addActivity(String performMethodName, String stateMethodName) {
      // create Activities from public methods
      Method[] methods = getClass().getMethods();
      int i = 0;
      Method performMethod = null;
      Method stateMethod = null;
      while (i < methods.length) {
         String methodName = methods[i].getName();
         if (methodName.equals(performMethodName)) {
            performMethod = methods[i];
         }
         else if (methodName.equals(stateMethodName)) {
            stateMethod = methods[i];
         }
         if (performMethod != null && stateMethod != null) {
            break;
         }
         i++;   // search the first method with the given name
      }
      PCCActivity activity = new GeneratedActivity(performMethod, stateMethod);
      addActivity(activity);  // calls also activity.setProvider(this);
   }
}
```

The implementation of class `GeneratedActivity` can be found in Program 5.9-1.

For most cases the framework-supported generation of Activities (Program 5.9-5, Program 5.9-6, and Program 5.9-7) will do. However, in some cases one may want to do some special things where information is needed about the Activity (its environment, etc.), or one wants to group or encapsulate functionality in one Activity. In such a case, it makes sense to explicitly define Activities as in Program 5.9-4.

6 Comparison with Related Approaches

This chapter describes other approaches and facilities that are similar or at least related to those of PCoC.

6.1 Environmental Acquisition

This section gives an overview over the conventional acquisition mechanism called *environmental acquisition* and how a simple implementation could look like. It explains differences between implicit (automatic) and explicit acquisition.

We assume that the reader has read Section 5.4 as general introduction into the acquisition concept.

The following sources relating to this topic are strongly recommended: [GIL96], [ACQU00], [ACQU01], and some other acquisition documents [ACQU98].

6.1.1 Overview of Environmental Acquisition

Environmental acquisition is a mechanism that allows an object to obtain a requested feature from its environment (containment hierarchy). It is used where the environment of objects or components provides attributes or a context that has or can have an impact on the objects and components.

Let us assume we are implementing an interactive component, for example, a source-code editor. It may consist of different replaceable parts. One part could be the editor view, where we can edit source code. Another might be a visual component showing line numbers. A third could be a status information pane that shows the caret position and other status information. All of these components have something in common: the file buffer and caret information. It makes sense to make this information automatically accessible to all of the components so that they can use it as if they themselves owned it.

Generally we can say that environmental acquisition is necessary to share data among components in a containment hierarchy. The mechanism allows components to redefine attributes they acquired from their environment. For some components it might make sense to manage information or redefine some methods independently of their environment, or to wrap shared information or functionality.

6.1.2 Environmental Acquisition in Literature

[GIL96] explains the motivation for introducing environmental acquisition as follows:

To give the reader a taste of the motivation for this research, consider the following example which may occur in an automobile industry application: An object of a class `Car` depicted in Figure 6.1-1 is a composite which comprises components such as objects of class `Door`.

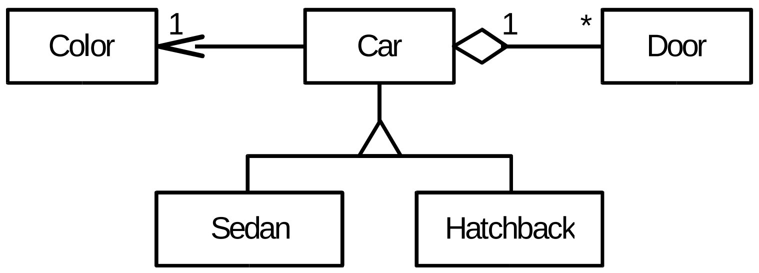


Figure 6.1-1 Environmental acquisition in the problem space: Class `Door` acquires its color from `Sedans` and `Hatchbacks` alike

Suppose that it is known that a car is colored red, then we are likely to infer that its doors are red as well. However, although the door “inherits” its color from the car of which it is part, it would be wrong to derive `Door` from `Car`. `Color`-inheritance is related to the “in-a” (reverse-“has-a”) link which binds doors to cars.

This becomes also clear when examining `Car` subclasses, say `Sedan` and `Hatchback`, which are distinguished among other things by the number of doors. Class `Door` inherits its color from `Sedan` and `Hatchback` alike, just as it would from another hypothetical class `Airplane` which has no “is-a” (“a-kind-of”) relationship with `Car`.

We call this kind of inheritance *environmental acquisition* and distinguish it from *class-based inheritance* (inheritance for short) in its common meaning in object oriented programming.

[ACQU98], [ACQU00], and [ACQU01] are helpful for understanding the basics of environmental acquisition.

6.1.3 Simple Implementation

The following code fragment shows a simple implementation of environmental acquisition.

Program 6.1-1 Acquisition (Java)

```
public class AcquisitionBase {
   protected Object parent;
   public AcquisitionBase() { parent = null; }
   public printResult(String methodName) {
      Object o = perform(methodName);
      if (o != null) { System.out.println("Result: " + o); }
   }
   public Object perform(String methodName) {
      Object result = null;
      try {
         Method m = getClass().getMethod(methodName, null);
         result = m.invoke(this, null);
      }
      catch (Exception ex) {
         if (parent != null) { result = parent.perform(methodName); }
         else { System.err.println("no such method: "+methodName); }
      }
      return result;
   }
}
```

In this example, we can dynamically invoke methods by using the method `perform`. The specified method is looked up in the current object first, respectively in its class and its base classes, and then in the parent object.

The example is based on Java reflection. See [ENGLA97] on Pages 40-42 for a short introduction into Java reflection.

Program 6.1-2 Acquisition (Java)

```
public class A extends AcquisitionBase {
   public A() {
   }
   public A(Object p) {
      parent = p;
   }
   public String getAValue() {
      return "A";
   }
}

public class B extends AcquisitionBase {
   public A a;
   public B() {
      a = new A(this);
   }
   public String getValue() {
      return "B";
   }
}

public class C extends AcquisitionBase {
   public A a;
   public C() {
      a = new A(this);
   }
   public String getValue() {
      return "C";
   }
}
```

Program 6.1-1 and Program 6.1-2 show a simple implementation of an environmental acquisition mechanism for invoking methods. `AcquisitionBase` is a class introducing the method `perform` for dynamically invoking methods using the Java reflection mechanism and providing implicit (automatic) environmental acquisition.

If a method cannot be found in the current object, the request for invoking a method named `methodName` is delegated to its container. `perform` succeeds if an object is found providing a corresponding method. It fails if the root object in the containment hierarchy is found without finding a method with the given name.

Note that with the given implementation it is only possible to invoke methods without parameters using the `perform` method, but it should be enough to provide a rough overview of how acquisition works.

Let us take a look at a use case for our example above:

Program 6.1-3 Using Acquisition (Java)

```
A a = new A();
B b = new B();
C c = new C();

b.a.printResult("getValue");
--> "Result: B"
c.a.printResult("getValue");
--> "Result: C"
a.printResult("getValue");
--> "no such method: getValue"
a.printResult("getAValue");
--> "Result: A"
```

If method `getValue` is invoked on `b.a` or `c.a`, it is found in the container `AcquisitionBase`. If it is invoked directly on `a`, which does not provide the method and no container which could provide it, the call results in an error.

The Python programming language (see [ROSSUM90]) provides much more flexible usage of acquisition than Java, since no method or attribute declarations are necessary. The following code fragment shows the built-in acquisition mechanism of Python.

Program 6.1-4 Acquisition (Python)

```
class AcquisitionBase:
  def __getattr__(self, attribName):
    if self.parent:
      return getattr(self.parent, attribName)

class A(AcquisitionBase):
  def __init__(self, parent=0):
    self.parent=parent
  def getAValue(self):
      return "A"

class B(AcquisitionBase):
  def __init__(self):
    self.a=A(self)
  def getValue(self):
    return "B"

class C(AcquisitionBase):
  def __init__(self):
    self.a=A(self)
  def getValue(self):
    return "C"
```

If Python does not find a requested attribute or operation either in the class of the given object or in one of its base classes, it tries to find the method `__getattr__`. In this method we can define what Python should do in this case, for example, search for the method in the parent object, as in the example above.

The example code for utilizing acquisition looks similar to that in Java:

Program 6.1-5 Using Acquisition (Python)

```
a = A();
b = B();
c = C();

print 'Result: ' + b.a.getValue()
--> "Result: B"
print 'Result: ' + c.a.getValue()
--> "Result: C"
print 'Result: ' + a.getValue()
--> runtime error (TypeError: 'NoneType' object is not callable)
print 'Result: ' + a.getAValue()
--> "Result: A"
```

If a method `getValue` is invoked on `b.a` or `c.a`, it is found in the container. If it is invoked directly on `a`, which does not provide the method and no container which could provide it, the call results in an error.

Note that acquisition is not only possible with methods, but also with attributes.

6.1.4 Implicit Acquisition

We have seen what environmental acquisition is in general. Now, let us take a look at the differences between implicit and explicit acquisition.

Implicit acquisition automatically searches for attributes and methods in the environment whenever they cannot be obtained directly from an object or through class inheritance.

According to the section above, we call it *implicit* or *automatic acquisition* of method calls, whenever we invoke methods using `perform` of `AcquisitionBase`. If a method is called this way, it is first searched in the class or base classes of the object itself, and then of the parent object, and so on (cf. Program 6.1-1 and Program 6.1-2).

6.1.5 Explicit Acquisition

When explicit acquisition is used, attributes are not automatically obtained from the environment. Instead, an `acquire` method must be used.

Let us assume, we invoke methods as usual (ordinary method calls), without using acquisition. In some cases, though, it is useful to use the acquisition mechanism. In these cases we can explicitly use method `perform` (see Program 6.1-1) which searches and invokes specified methods in the environmental context of the specified object.

Program 6.1-6 Using Explicit Acquisition (Java)

```
a = A();
b = B();
c = C();

System.out.println("Result: " + b.a.getAValue());
--> "Result: A"
System.out.println("Result: " + b.getValue());
--> "Result: B"
System.out.println("Result: " + b.a.perform("getValue"));
--> "Result: B"
System.out.println("Result: " + c.a.perform("getValue"));
--> "Result: C"
```

In the example above, the methods `b.a.getAValue()` and `b.getValue()` are called directly. The other calls use environmental acquisition, e.g., for `b.a.perform("getValue")` the method `getValue` is searched in `b.a`, respectively in its base classes, and then in `b`, where it is finally invoked.

The main difference between explicit and implicit acquisition is that one can decide for each method to invoke it normally or using environmental acquisition (using method `acquire`).

6.1.6 Remarks

Environmental acquisition is, for instance, used and promoted by Zope—a software system combining a Web-server and a content management system.

Following sources cover concepts of environmental acquisition: [GIL96], [ACQU00], [ACQU01], and [ACQU98]. Environmental acquisition is compared to a similar approach of PCoC in Section 5.4.

6.2 Aspect Oriented Programming

This section provides an rough overview of aspect oriented programming and illustrates this software-technical paradigm with a few examples.

6.2.1 Overview

Aspect-oriented programming (AOP) is a programming technique first defined by Xerox Palo Alto Research Center at the beginning of 1997. See [XEROX0297], [XEROX1297], and [XEROX0697]. Some sources say that the term already came up in OOPSLA circles in 1995.

Traditional software units of modularity in programming languages include modules, classes, and methods. However, some functionality cannot be encapsulated in single modules. Some units are multiply implemented across the class hierarchy.

Aspect-oriented programming (AOP) brings new units of modularity, called *aspects*. Aspects are constructs, respectively implementations as classes, representing factors of behavior (features) and can be involved in more than one object or component. AOP builds on traditional technologies, including procedural and object-oriented programming, which have already made significant improvements in software modularity.

Aspects eliminate object and component borders. Code relating to aspects is often expressed as small code fragments tangled and scattered throughout several components. Because of the way they cross module boundaries, it is said that aspects *crosscut* the program's hierarchical structure. Aspects encapsulate crosscutting concerns.

As mentioned, aspects can be invoked on different parts of a system, and they are not limited to public interfaces of objects and components.

Typical aspects are:

- tracing/logging/monitoring
- error/exception handling
- synchronization/thread-safety
- caching strategies
- persistency
- resource sharing
- distribution concerns
- optimizations
- etc.

Aspect compilers are used to generate code from actual source code and aspects. When using such language extensions, one does not actually see the existence of aspects in the source code where they are later invoked. The code produced by the aspect compiler can then be compiled by a normal compiler. The term "*code-weaver*" is often used for this kind of compiler to illustrate what it does. However, AOP is also possible without preprocessors. Aspects can, for instance, be invoked through templates, name spaces, and inheritance (see later). PCoC, for example, supports crosscutting concerns at runtime (cf. Sections 2.8 and 5.4).

An example for an aspect-oriented programming language is AspectJ (see [KICZAL00]). It is a general-purpose AOP extension to Java and product of years of research at Xerox PARC. The project is partially supported by the Defense Advanced Research Projects Agency (DARPA).

AspectJ enables the modularization of crosscutting concerns as described above. A corresponding aspect compiler is called AspectWeaver^{TM} (ajc). It runs right before the Java compiler.

The main design goals of AspectJ were to make it a compatible extension to Java and as simple as possible.

Compatibility includes:

- Upward compatibility: all regular Java programs must also be regular AspectJ programs.
- Platform compatibility: regular AspectJ programs must run on standard Java virtual machines.

- Tool compatibility: it must be possible to work with AspectJ using existing tools, including IDEs, documentation and design tools.
- Programmer compatibility: programming with AspectJ must feel like natural extension of programming with Java.

AOP is said to become a powerful means of coping with the rising complexity of modern software applications. Enabling highly modular software, it holds the potential for simplifying the development of complex software applications, for improving the reuse of code fragments, and finally for reducing maintenance efforts and costs.

A good introduction into aspect-oriented programming can be found in [CZARN01]—an article in the iX magazine of 2001—and in subsequent articles of the series. Other useful sources for this section were [LUTZ01] and [KICZAL00].

6.2.2 AspectJ

A critical element in the design of any aspect-oriented programming language is the join point model. The join point model provides the frame of reference that makes it possible for execution of a program's aspect, and non-aspect code to be coordinated properly.

AspectJ is based on a model in which so-called join points are nodes in a simple runtime object call graph. These nodes include points at which an object receives a method call and points at which a field of an object is referenced. The edges between nodes represent control flow relations. In this model each node is visited twice—once when walking down the call graph for subnodes, and once when coming back from the call of subnodes.

The following join points are possible (taken from [KICZAL00]):

join point	***pointcut designator syntax***	***explanation***
calling method and constructors	`calls(<signature>)`	a method or constructor of a class is called. Call join points are in the calling object or not related to an object if a static method is called
reception of method and constructor calls	`receptions(<signature>)`	an object receives a method or constructor call. Reception join points are before method or constructor dispatch, i.e., they happen inside the called object. Control flow has already been transferred to the called object, but no particular constructor and method has been called yet.
execution of methods and constructors	`executions(<signature>)`	an individual method or constructor is invoked

getting fields	`gets(<signature>)`
	`gets(<signature>) [<value>]`
		a field (member variable) of an object, class or interface is read

setting fields	`sets(<signature>)`
	`sets(<signature>) [<value>]`
	`sets(<signature>) [<oldvalue>] [<newvalue>]`
		a field of an object or class is set

exception handling	`handles(<throwable type name>)`	an exception handler is invoked

Note that pointcuts identify join points of specific types. Here are some more complex pointcut designators:

join point	***pointcut designator syntax***	***explanation***
any	`instanceof(<currently executing object type name>)`	matches join points if the currently executing object is of the given type
any	`within(<class name>)`	matches join points if the the currently executing code is contained within class <class name>
any	`withincode(<signature>)`	matches join points if the the currently executing code is contained within the member defined by the method or constructor signature
any	`cflow(<pointcut designator>)`	matches join points of any kind that occur within the dynamic extent of any join point matched by <pointcut designator>
method calls	`call(<pointcut designator>)`	matches method call join points that in one step lead to any reception or execution join points matched by <pointcut designator>

Note that a pointcut is a set of join points that optionally exposes some of the values in the execution context of these join points. Pointcut designators can be combined using `AND`, `OR`, and `NOT` operators (‘&&’, ‘`||`’, ‘`!`’).

Before we take a look at using pointcuts, we define a simple class:

Program 6.2-1 Sample class

```
// A regular class
class MyPoint {
   private int fx;
   private int fy;

   public void setX(int x) {
      fx = x;
   }
   public void setY(int y) {
      fy = y;
   }
   public int getX() {
      return fx;
   }
   public int getY() {
      return fy;
   }
}
```

Now we can define a simple pointcut:

Program 6.2-2 A simple pointcut

```
pointcut myPointMoves():
   receptions(void MyPoint.setX(int)) ||
   receptions(void MyPoint.setY(int));
```

Pointcut `myPointMoves` identifies whenever the position of an object of class `MyPoint` is changed.

Another facility of AspectJ is called *advice*. An advice is a method-like mechanism used to declare that certain code should execute at each of the join points in the pointcut. AspectJ supports `before`, `after`, and `around` advice.

`before` and `after` are quite self-explanatory. `around` is invoked even before the `before` section. There can be many `around` sections for a join point, these are invoked with the most specific piece first. The body of an `around` advice can call `runNext` which invokes the next most specific piece, and if no other `around` body is left, `before` is invoked.

Here is a simple example of an `after` advice:

Program 6.2-3 A simple advice

```
static after(): myPointMoves() {
   System.out.println("Point moved.");
}
```

Parameters can be passed to advices as well as to pointcuts. Read more about these constructs in [KICZAL00] and [XEROX02].

Aspects are defined by aspect declarations, which are similar to class declarations. They may include pointcut declarations, advice declarations, as well as all other kinds of declarations permitted in class declarations.

Program 6.2-4 Aspects in AspectJ

```
// Define a tracing aspect for set methods and copy
aspect MyTracer {
   pointcut tracedCalls():
      calls(void MyPoint.set*(int)) && calls(void MyPoint.get*());
   before(): tracedCalls() {
      System.out.println("Invoking: " + thisJoinPoint)
   }
}
```

In the example above we register all `set` and `get` methods of class `MyPoint` in the pointcut `tracedCalls`.

The idea behind all this is to extend existing classes and methods by specific behavior and to keep these extensions out of regular code. These units supplying the specific behavior are called aspects. An aspect can cover different classes and methods at the same time, thus combining their common functional behavior (aspect) at one location. This makes it not only easier to maintain common aspects such as tracing or preconditions—it keeps code more readable. A disadvantage is the need for additional compile runs using aspect parsing tools.

6.2.3 Other Mechanisms

Beside the most advanced and sophisticated tool AspectJ, there are some other mechanisms and development tools available for aspect-oriented programming. One is the programming language *D*, also developed by Xerox. Read more about it in [XEROX1297].

The idea behind all available approaches and tools is the same—to provide a mechanism for weaving code. Different aspects of code can be developed and maintained separately.

[CZARN01] describes a concept for aspect-oriented programming with C++. It uses name spaces to separate pure functional code from aspects.

The following example illustrates this approach:

Program 6.2-5 Defining aspects by using name spaces in C++ (1)

```
namespace original
{
   // A regular class
   class MyPoint {
   private:
      int fx;
      int fy;
   public:
      MyPoint()
         : fx(0), fy(0) { }
      void setX(int x) { fx = x; }
      void setY(int y) { fy = y; }
      int getX() { return fx; }
      int getY() { return fy; }
   };
}
```

Note that we define the original class in the name space `original`. Having defined it, we can go on and define an aspect for this class in another name space:

Program 6.2-6 Defining aspects by using name spaces in C++ (2)

```
namespace aspect
{
   typedef original::MyPoint MyPoint;
   // A regular class
   class MyPointWithTracing : public original::MyPoint {
   public:
      void setX(int x) {
         MyTracer::before("setX");
         MyPoint::setX(x);
      }
      void setY(int y) {
         MyTracer::before("setY");
         MyPoint::setY(y);
      }
      int getX() { return MyPoint::getX(); }
      int getY() { return MyPoint::getY(); }
   };

   class MyTracer {
      static void before(char* str) { cout << "before " << str endl; }
      static void after(char* str) { cout << "after " << str endl; }
   }
}
```

In the program above, we first define `MyPoint` as `original::MyPoint` for simplification of the rest of the code. We trace each call of `setX` and `setY` by invoking our tracer. Of course, we then have to call the actual method of the base class.

Now that we have defined the aspect, we can make use of the class in any user code:

Program 6.2-7 Aspects using name spaces in C++ (3)

```
namespace composed
{
   // typedef aspect::MyPoint MyPoint;         // original point class
   typedef aspect::MyPointWithTracing MyPoint; // point class with tracing
}
// use that point class configured in name space "composed"
using namespace composed;
MyPoint p;
p.setX(10);
```

Note that in this case the base class `original::MyPoint` need not declare its methods as virtual. The nice thing about this concept is that neither client code, nor code of the original class is affected by the aspect. Aspects can be switched on and off just by configuration of the composed name space (which itself can happen outside client code, maybe in a special header file). It may not be as convenient as AspectJ, and it is not possible to define pointcuts, but it is a way to introduce aspects in existing code without bothering with a language and tools for aspect-oriented programming.

6.2.4 Remarks

Aspect-oriented programming (AOP) brings new units of modularity and thus reusability, called *aspects*. Aspects are constructs, respectively implementations as classes, representing factors of behavior (features) and can be involved in more than one object or component.

AOP is said to become a powerful means of coping with the rising complexity of modern software applications. Enabling highly modular software, it holds the potential for simplifying the development of complex software applications, for improving the reuse of code fragments, and finally for reducing maintenance efforts and costs.

There are already real AOP solutions such as AspectJ. See [KICZAL00].

The PCoC framework supports crosscutting concerns at runtime, like AOP does at compile time. Different aspects can dynamically wrap Activities (see Section 2.8) or Activity Sets (acquisition in PCoC, see Section 5.4). Clients can also attach as listeners to Dispatchers and Activities, so they get notified before and after the Dispatchers or Activities are invoked or their state is changed (see Section 2.9). On notification, clients can take corresponding action.

Since operations (Activities) are objects in PCoC, they can be traced and extended by advice mechanisms as in AspectJ. For example, a component developer has to override the method `doPerform` of an Activity base class (cf. Program 4.3-15). Since `doPerform` is wrapped and invoked by the base-class method `perform`, the framework can do a great deal of administrative work before and after the invocation of actual user code. This approach is, for example, used for tracing invocations of Activities for later replay through scripting (macro recording and replay).

Currently, there is no sophisticated implementation of pointcut and join point registration and evaluation in PCoC (except tracing Activity invocations for the purpose of macro recording). However, it is possible to attach listeners to Dispatchers. They can provide execution state information (before execution, executing, after execution) and other reflective information (current Activity Set, etc.) The reflection facilities together with the listener support can be used to simulate pointcuts for the invocation of Activities.

Read more about aspect-oriented programming in [XEROX0297], [XEROX1297], [XEROX0697], and [KICZAL00].

6.3 Event Channeling

Events are an important communication mechanism for many domains. We distinguish between directed events where the receiver of an event is specified by its sender, and event channels.

An event channel is a channel at which interested receivers can register. The sender does not normally know the actual receivers. The advantage of this concept is that it can be used to decouple involved objects.

However, distribution and handling of events over event channels also leads to some problems. For example, it may be difficult to find out the sender of an event, or the reason why it was triggered. There is also an increased effort to handle different events that are sent through the same event channel. Generally, analyzing and dispatching an event via event-channeling may require more effort on the receiver-side than for directed events.

A solution can be a mediator instance within an event channel that selects and invokes special callbacks on receivers. Such a mediator can be a table that associates events with concrete methods. PCoC has a similar approach. A request is directly delegated to Activities on receiver side, which are associated with the request. More precisely the method `perform` of each associated Activity is called, when a Dispatcher delegates a corresponding request. An Activities Provider can register at a kind of event channel or even create it by adding an own Activity to an Activity Set that is in the acquisition branch of the Dispatcher (via proper dispatch directives). In order to make an Activity Set and a corresponding Activity reachable to a Dispatcher, either the Activity Set must be acquired by one that is already in the current acquisition branch of the Dispatcher, or the directives must be properly adapted. No analyzing and dispatching is necessary on the receiver side. Read more about the dispatch mechanism of PCoC in Section 5.6.2.

Events and especially event channeling are described in great detail in [GRIFF98] on Pages 254ff.

6.4 Java Reflection

This section describes Java reflection facilities such as dynamic method invocation, dynamic class lookup and instantiation, and proxy class. Concepts are compared to those of PCoC.

6.4.1 Overview

The Java programming language provides a facility called reflection. It allows an executing Java program to examine (introspect) itself, and to manipulate internal properties of the program. For example, it is possible for a Java class to obtain the names of all its members.

The ability to examine and even manipulate a class from within itself may not sound impressive, but in other programming languages this feature does not exist. For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program. Microsoft’s .NET introduces reflection facilities for its supported programming languages: Visual Basic, Managed C++, C#, and Microsoft’s version of Java. Currently, developers are restricted to Microsoft’s compiler and their operating systems to make use of .NET and its facilities. See also Microsoft’s .NET Readiness Kit, [MICROS01].

One tangible use of reflection is in JavaBeans where software components can be manipulated visually via a builder tool. The tool uses reflection to obtain the properties of Java components (classes) as they are dynamically loaded.

Compare Java reflection to .NET reflection facilities in Sections 6.7.6 and 6.7.5.

6.4.2 Reflecting Classes and Methods in Java

The class `Class` is a meta-class that holds information about other classes. It can be used to instantiate objects of a given class. The following two accesses are probably the ones used most often:

Program 6.4-1 Retrieving a class object

```
Class cl = Class.forName("MyClass");
Class cl = MyClass.class;
```

These are almost equivalent in meaning, however, the first statement throws an exception if the specified class does not exist (`ClassNotFoundException`). In either case `cl` gets a reference to a class, which can then be used to look up the methods of the class.

For primitive data types the corresponding class can be retrieved as follows:

Program 6.4-2 Retrieving a class object for primitive data types

```
Class c = int.class;
Class c = Integer.TYPE;
```

Both statements are equivalent. Primitive data types are not really relevant here; these approaches are mentioned for completeness.

The following non-static methods of `Class` are the most interesting for dynamic method invocation:

Program 6.4-3 Retrieving method objects

```
public Method getMethod(String name, Class[] parameterTypes);
public Method[] getMethods();
```

`getMethod` returns a Method object that reflects the specified public member method of the class or interface represented by this `Class` object.

`getMethods` returns an array containing `Method` objects reflecting all the public member methods of the class or interface represented by this `Class` object, including those declared by the class or interface and and those inherited from base classes and super-interfaces.

There are similar methods `getDeclaredMethod` and `getDeclaredMethods` that return only methods declared in the given class, but not in one of its base classes or base interfaces.

The JDK 1.2 API documentation [JDK12] provides more detailed description of these methods.

6.4.3 Java Type Checking

We may want to check whether an object is an instance of a specific class.

Program 6.4-4 Checking the class of an object

```
Class cl = Class.forName("MyClass");  // or cl = MyClass.class
boolean classOK = MyClass.class.isInstance(new Integer(10));
System.out.println(classOK);  // output is false
boolean classOK = MyClass.class.isInstance(new MyClass());
System.out.println(classOK);  // output is true
```

In this example, a `Class` object for `MyClass` is created, and then some objects are checked to see whether they are instances of `MyClass` or of any of its subclasses. `new Integer(10)` is not, but `new MyClass()` is. The method `isInstance` is the dynamic equivalent of the `instanceof` operator.

6.4.4 Dynamic Method Invocation in Java

In Java, methods can be invoked dynamically by using method `invoke` of class `Method`. This feature is illustrated in the the following example. Note that the `Method` class is in the Java reflection package `java.lang.reflect`.

Program 6.4-5 shows an application of Java reflection on a String object (`string`). We specify the kind of method we want, which is in this case the method `concat` with parameter types `parameterTypes`. `getMethod` returns a method reference to a corresponding method, or throws an exception if an error occurs. With `invoke` we specify the object where method `m` should be invoked, and the arguments that should be passed to the method—in this case the string “`def`”. Finally, we print the result.

Program 6.4-5 Dynamic method invocation

```
String string = "abc";
String result = null;
Class[] parameterTypes = new Class[] {String.class};
Object[] arguments = new Object[] {"def"};

try {
   Method m = String.class.getMethod("concat", parameterTypes);
   result = (String) m.invoke(string, arguments);
}
catch (NoSuchMethodException e) { System.out.println(e); }
catch (IllegalAccessException e) { System.out.println(e); }
catch (InvocationTargetException e) { System.out.println(e); }
System.out.println("Result: " + result);
```

Read more about dynamic method invocation on the .NET platform in Section 6.7.6.

6.4.5 Java Reflection: Invoking Constructors

Invocation of constructors is slightly different to that of methods. An example for constructor invocation is given below:

Program 6.4-6 Invoking constructors

```
import java.lang.reflect.*;

public class MyClass {
   public MyClass() {
   }

   public MyClass(int x) {
   }

   public static void main(String[] args) {
      try {
         Class cl = Class.forName("MyClass");
         Class[] paramaterTypes = new Class[1];
         paramaterTypes[0] = Integer.TYPE;
         Constructor ct = cl.getConstructor(paramaterTypes);
         Object[] arglist = new Object[1];
         arglist[0] = new Integer(10);
         Object retobj = ct.newInstance(arglist);
      }
      catch (Throwable e) {
         System.err.println(e);
      }
   }
}
```

First, we get the class reference for class `MyClass`. Then, we specify the parameter types of the constructor we want to get, which is in this case a constructor with a single `int` parameter. We retrieve the constructor, and use it to create a new instance of `MyClass`, and pass an `Integer` object with the value `10`.

6.4.6 Java Reflection: Fields

Another feature of the Java reflection API is the possibility to inspect fields and to modify their values at runtime. This is illustrated in the following example.

Program 6.4-7 Retrieving fields

```
import java.lang.reflect.*;

public class MyField {
   public int val;
   public MyField() { val = 10; }
   public static void main(String[] args) {
      try {
         Class cl = Class.forName("MyField");
         Field f = cl.getField("val");
         MyField fObj = new MyField();
         System.out.println("val = " + fObj.val);  // output is 10
         f.setInteger(fObj, 20);
         System.out.println("val = " + fObj.val);  // output is 20
      }
      catch (Throwable e) { System.err.println(e); }
   }
}
```

We retrieve the `int` field `val` and set its value from `10` to `20`.

This feature is needed for JavaBeans where it is possible to change the value of a field via introspection in the GUI of a JavaBean.

6.4.7 Dynamic Classes in Java: Class Proxy

All the software constructs of the reflection package presented so far have one thing in common—they cannot be assembled at runtime. They can only be inspected. In the case of fields, also the content can be modified. This may, however, not always be sufficient.

In Java 1.3, Sun introduced a new class called `Proxy`, which allows dynamically assembling classes, so-called *dynamic proxy classes*, or *proxy classes* for short. This section describes the class and its use.

Sources for information about Dynamic Proxies are [JDK13P], [JDK13D], and [BLOSS00]. The following sections contain large excerpts of these documents.

6.4.7.1 Overview

`Proxy` allows assembling a class at runtime, a so-called (dynamic) proxy class. A list of interfaces specifies which interfaces the proxy class implements. Method invocations on the proxy class are dispatched to an invocation handler.

The class `Proxy` is derived from `Object`:

Program 6.4-8 Class Proxy

```
public class Proxy
   extends Object
   implements Serializable { ... }
```

A proxy class can be created with the following static method of `Proxy`, given a list of interfaces and a class loader.

Program 6.4-9 Method for creating Proxy classes

```
public class Proxy
  extends Object
  implements Serializable {
   ...
   public static Class getProxyClass(ClassLoader loader, Class[] interfaces)
     throws IllegalArgumentException {
       ...
   }
}
```

The list of classes contains only interfaces, not classes or primitive types. The specified interfaces must be visible by name through the class loader.

All non-public interfaces must be in the same package. There must not be methods with the same name and parameter lists and different return values in the given interfaces.

Given the interface:

Program 6.4-10 Sample interface

```
public interface IPoint {
   void setLocation(int x, int y);
}
```

we can create a proxy class as follows:

Program 6.4-11 Creating a Proxy class

```
Class proxyClass = Proxy.getProxyClass(IPoint.class.getClassLoader(),
                                       new Class[] {IPoint.class});
```

This proxy class implements interface `IPoint`. Note that the methods of interfaces for a proxy class also support primitive data types as parameters. They are mapped to corresponding object types, e.g. `int` becomes `Integer`, `boolean` becomes `Boolean`, etc.

A proxy class itself cannot do much. For instantiation of the class we need an invocation handler to dispatch method calls invoked on the proxy class.

`InvocationHandler` is an interface that declares only the following method:

Program 6.4-12 InvocationHandler interface

```
public interface InvocationHandler {
   public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable;
}
```

`invoke` is called whenever a method is invoked on an associated proxy instance. The proxy instance passes itself, the invoked method, and the argument list of the invoked method as arguments.

Given the invocation handler:

Program 6.4-13 Sample invocation handler

```
public class PointProxyHandler implements InvocationHandler {
   public Object invoke(Object proxy, Method method, Object[] args)
     throws Throwable {
      try {
         System.out.println("method: " + method.getName());
         String str = "arguments:";
         for (int i = 0; i < args.length; i++) { str = str + " " + args[i]; }
         System.out.println(str);
      }
      catch (InvocationTargetException e) { // an invoked method threw an exception
         throw e.getTargetException();      // forward this exception
      }
      catch (Exception e) {
         throw new RuntimeException("Unexpected exception " + e.getMessage());
      }
      finally { System.out.println("after method: " + method.getName()); }
   }
}
```

and the proxy class of Program 6.4-11, we can create a proxy instance as follows:

Program 6.4-14 Creation of a Proxy instance (1)

```
Class proxyClass = Proxy.getProxyClass(IPoint.class.getClassLoader(),
                                       new Class[] {IPoint.class});
IPoint pointProxy = (IPoint) proxyClass.
   getConstructor(new Class[] { InvocationHandler.class }).
   newInstance(new Object[] { new PointProxyHandler() });
```

Mostly the proxy class object is not needed; one can use a simpler method of `Proxy`, combining proxy definition and instantiation:

Program 6.4-15 Creation of a Proxy instance(2)

```
IPoint pointProxy = (IPoint) Proxy.newProxyInstance(IPoint.class.getClassLoader(),
                                 new Class[] {IPoint.class},
                                 new PointProxyHandler() });
```

Now we can invoke methods on our proxy instance:

Program 6.4-16 Invocation of a Proxy instance

```
pointProxy.setLocation(10, 20);
if (pointProxy instanceof IPoint) { System.out.println("ok"); }
```

In this case, the invocation of all methods on `pointProxy` will be dispatched to `PointProxyHandler`. This includes, besides `setLocation`, the methods `hashCode`, `toString`, and `equals` which are declared in `java.lang.Object`.

The cast operation `(IPoint)` will succeed and not throw a `ClassCastException`. The proxy can be casted to all interfaces specified at proxy creation.

`instanceof` will also return `true` for all interfaces that have been specified for a proxy class.

Sometimes we may want the invocation handler, in our case `PointProxyHandler`, to dispatch method calls to a special object:

Program 6.4-17 Invocation handler forwarding method calls

```
public class PointProxyHandler implements InvocationHandler {
   Object point;
   public PointProxyHandler(Object point) {
      this.point = point;
   }

   public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {

      Object result;
      try {
          result = method.invoke(point, args);
      }
      catch (InvocationTargetException e) { // an invoked method threw an
         throw e.getTargetException();      // exception; forward this exception
      }
      catch (Exception e) {
         throw new RuntimeException("Unexpected exception " + e.getMessage());
      }
      finally {
      }
      return result;
   }
}
```

The object that should actually perform invoked methods is passed as an argument with the constructor. In any case, the invocation handler of a proxy class either dispatches any method calls to various objects or directly provides the implementation for the methods declared in the interfaces added to the associated proxy. With PCoC, a Dispatcher only delegates requests for the invocation of Activities with the same name (and signature) as the Dispatcher. Each request is delegated to Activities. An Activity implements a single operation.

If we call any method in the invocation handler of a proxy, we have to pass the proxy instance as receiver in order to use delegation rather than forwarding. For example, `otherMethod.invoke(proxy, args).`

Given the class implementing the `IPoint` interface

Program 6.4-18 Interface implementation for a Proxy instance

```
public class PointImpl implements IPoint {
   int x;
   int y;
   void setLocation(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public String toString() { ... }
   public boolean equals(Object obj) { ... }
   ...
}
```

the creation of a proxy will look as follows:

Program 6.4-19 Invocation of a method using a Proxy instance

```
IPoint pointProxy = (IPoint) Proxy.newProxyInstance(IPoint.class.getClassLoader(),
                                 new Class[] {IPoint.class},
                                 new PointProxyHandler(new PointImpl()) });
pointProxy.setLocation(10, 20);
if (pointProxy instanceof IPoint) { System.out.println("ok"); }
```

In this case, the invocation of all methods on pointProxy will be dispatched to PointProxyHandler, respectively to PointImpl. This includes, besides setLocation methods hashCode, toString, and equals which are declared in java.lang.Object.

The cast operation (IPoint) will succeed, and not throw a ClassCastException. The proxy can be casted to all interfaces specified at proxy creation.

instanceof will also return true for all interfaces specified for a proxy class.

6.4.7.2 Methods Duplicated in Multiple Proxy Interfaces

When two or more interfaces of a proxy class contain a method with the same name and parameter signature, the order of the proxy class's interfaces becomes significant. When such a duplicate method is invoked on a proxy instance, the Method object passed to the invocation handler will not necessarily be the one whose declaring class is assignable from the reference type of the interface that the proxy's method was invoked through. This limitation exists because the corresponding method implementation in the generated proxy class cannot determine which interface it was invoked through. Therefore, when a duplicate method is invoked on a proxy instance, the Method object for the method in the foremost interface that contains the method (either directly or inherited through a superinterface) in the proxy class's list of interfaces is passed to the invocation handler's invoke method, regardless of the reference type through which the method invocation occurred.

If a proxy interface contains a method with the same name and parameter signature as the hashCode, equals, or toString methods of java.lang.Object, when such a method is invoked on a proxy instance, the Method object passed to the invocation handler will have java.lang.Object as its declaring class. In other words, the public, non-final methods of java.lang.Object logically precede all of the proxy interfaces for the determination of which Method object to pass to the invocation handler.

PCoC has a similar model for dispatching requests to Activities with the same name and signature, acquired from different Activity Sets. Let us assume, an Activity Set A has acquired Dispatchers with the name copy from the Activity Set B and then from C (in this order). When the copy Dispatcher is invoked in A, it delegates requests with directive PCCDirectives.first to an Activity in (or acquired by) the first acquired Activity Set B. With directive PCCDirectives.broadcast, requests are delegated to B and C. The ranking of the acquired Activity Sets can be changed at any time, for example, upon focus changes in the GUI, etc. See Sections 5.4.3 and 5.6.2.

6.4.7.3 Remarks

Proxy allows the assembling of so-called proxy classes at runtime from a list of interfaces. By providing an invocation handler one can create a proxy instance from a proxy class. A proxy instance will behave like an instance of a real class. It is allowed to call all methods declared in the specified interfaces. These method calls will be dispatched to the specified invocation handler, as well as those of the methods hashCode, equals, or toString declared in java.lang.Object. The operator instanceof on a proxy instance returns true, if the corresponding proxy class implements the specified interface.

Although this facility makes Java more dynamic, it also has its limitations. For example, as already mentioned, a list of interfaces has to be specified. This means that proxy classes are not fully dynamic. So, although they are assembled at runtime, it is not possible to add classes and instances at runtime where interfaces do not exist at compile time.

PCoC has a slightly different approach; it allows adding operations, respectively Activities, to a component at runtime. Imagine a source-code editor Tool providing only the following basic operations: loadFile, saveFile, cut, copy, paste, getSelection, `setSelection`, `insert`. Say, we want to write an extension to this editor, but do not have the source code (or do not want to pay additional license fees). We may just want to reduce compile time, and the freedom to change the source code without forcing third-parties to rebuild everything. Anyway, let us assume, we want to add some code cleanup features to the editor, that allow formatting source code following some coding conventions. In this case the following new operations have to be introduced (they are self-explanatory): `indentLine`, `comment`, `reformat`. PCoC allows to add these Activities to the editor at runtime. Users of the editor would think these Activities originally were provided by the editor itself. The Activities belong to the same Activity Set as the editor, but they may be created by another, external component connected to the application via XMLRPC over TCP/IP (locally or remote).

Other advantages compared to proxy classes are the possibility to change the ranking of acquired Activity Sets, respectively their Dispatchers, at runtime, the support for multicasts, and no need to define interface classes. Java Proxies and PCoC Dispatchers both support delegation. An Activity can send requests to the original receiver (Activity Set) where the Activity has been requested. When an invocation handler of a proxy uses the proxy instance as receiver for further method calls, we also have delegation. Note that the proxy instance where a method has been invoked on is passed as first argument to an invocation handler. See Program 6.4-13.

The advantages of proxy classes are a lower memory consumption, better performance, and their simple use—methods of proxy class instances are invoked like instance methods of any other classes:

```
pointProxy.setLocation(10, 20)
```

In contrast, with PCoC, we have to use Dispatcher requests and to pack arguments into an argument container:

```
activitySet.perform("|MyPoint|setLocation",
                    (new PCCMaterial(10)).add(20))
```

6.5 Java Swing Actions

This section presents the Java Swing *action* pattern and some typical use cases, and discusses differences to PCoC. We will see how to use actions to share commands among different user interface components such as menu items and toolbar buttons. We will also see how to change status info when the mouse moves over action-based regions of a component. Another example will show us how action events can be forwarded from one class to another. Finally, this section discusses differences between Java actions and PCoC constructs such as Activities, Dispatchers, and Tasks.

This section refers mainly to the following articles: [DAVID00], and [TOEDT01] on Pages 32ff. For a more detailed description of Java Swing actions, see [WALRA99] on Pages 389ff, and 459ff.

6.5.1 *Overview*

A Java Swing *action* is an implementation of the command design pattern. It separates the controller logic of a command from a visual representation such as menu items or toolbar buttons. This is useful because the look of a menu or toolbar item can be changed just by changing its configuration, instead of making the changes in source code. Actions support property changes, for example, their short description, their state (“Enabled”, “Disabled”), etc.

So far, Swing actions are similar to PCoC Tasks together with the Dispatchers it uses and Activities—the classes that actually implement operations. Before getting too deep into detailed differences, take a more detailed look at Java Actions.

6.5.2 Using Java Swing Actions

The `javax.swing.Action` interface extends `ActionListener`. It inherits the method declaration of `actionPerformed` from `ActionListener`.

Program 6.5-1 ActionListener interface

```
public interface ActionListener extends EventListener {
   public void actionPerformed(ActionEvent e);
}
```

This method is invoked when an `ActionEvent` occurs, and an object of a class implementing this interface has been registered with a Java component, using the component's `addActionListener` method.

Program 6.5-2 Using ActionListeners

```
JMenuItem menuItem = new JMenuItem("Foo");
public class MyActionListener implements ActionListener {
   public MyActionListener(JMenuItem menuItem) {
      menuItem.addActionListener(this);
   }
   public void actionPerformed(ActionEvent e) {
      System.out.println("actionPerformed triggered by ", e.getSource);
   }
}
```

Later we will see that an `Action` implicitly becomes an `ActionListener` of menus, menu items, or toolbar buttons where the `Action` instance is added.

Now to the actual `Action` interface:

Program 6.5-3 Action interface

```
public interface Action extends ActionListener {
   public static final String DEFAULT = "Default";
   public static final String NAME = "Name";
   public static final String SHORT_DESCRIPTION = "ShortDescription";
   public static final String LONG_DESCRIPTION = "LongDescription";
   public static final String SMALL_ICON = "SmallIcon";
   public static final String ACTION_COMMAND_KEY = "ActionCommandKey";
   public static final String ACCELERATOR_KEY = "AcceleratorKey";
   public static final String MNEMONIC_KEY = "MnemonicKey";

   public Object getValue(String key);
   public void putValue(String key, Object value);

   public void setEnabled(boolean b);
   public boolean isEnabled();

   public void addPropertyChangeListener(PropertyChangeListener l);
   public void removePropertyChangeListener(PropertyChangeListener l);
}
```

In addition to `actionPerformed`, the `Action` interface adds some string definitions and method declarations.

It defines a set of keys, such as `Action.SHORT_DESCRIPTION`, where a key represents the name of a property of an action instance.

These keys together with the methods `putValue` and `getValue` are used to set and examine properties such as descriptions, icon, and associated keystrokes.

Here, the meanings of some keys:

- `DEFAULT` is a constant that can be used as a key for `getValue` or as a value for `putValue` calls for text properties and icons.
- `SMALL_ICON` is used for a small icon of an action in toolbar buttons.
- `ACTION_COMMAND_KEY` is used to determine the command string for the `ActionEvent` that will be created when an action is going to be notified as the result of residing in a `KeyMap` associated with a `JComponent`.

There are many others; most rather self-explanatory.

Now let us use the property methods of `Action`. Since we cannot instantiate an interface, we use `AbstractAction`. This class implements the `Action` interface and adds property change support and storage and retrieval for key values. All we have to do is to subclass `AbstractAction` to add specific key values (name, description, or icon) and implement the `actionPerformed` method.

For example, the property value of an action instance can be set as follows:

Program 6.5-4 Using actions (1)

```
Action copyAction = new AbstractAction() {
   public void actionPerformed(ActionEvent e) {
      System.out.println("copy action triggered by ", e.getSource);
   }
}
copyAction.putValue(Action.SHORT_DESCRIPTION, "Copy selection");
```

The following code fragment does the same as that in the example above:

Program 6.5-5 Using actions (2)

```
Action copyAction = new AbstractAction("Copy selection") {
   public void actionPerformed(ActionEvent e) {
      System.out.println("copy action triggered by ", e.getSource);
   }
}
```

Note that the domain where actions are used is the same as that of PCoC Tasks (see Sections 4.5 and 5.7). Like actions, Tasks also provide a listener mechanism (see Section 4.3.2). The sender of a message is passed with a `PCCSenderInfo` instance.

Now back to actions. Action listeners attached with `addPropertyChangeListener` are notified whenever a property is changed using `putValue` (cf. Program 6.5-4), or by setting the state property using `setEnabled`. Listeners can be detached using `removePropertyChangeListener`.

All objects that have been added as listeners to an action will have a property-change listener; this will response to the `PropertyChangeEvent` sent by the action and change the state of the component.

Program 6.5-6 PropertyChangeListener interface

```
public interface PropertyChangeListener extends EventListener {
   public void propertyChange(PropertyChangeEvent evt);
}
```

`isEnabled` and `setEnabled` are used to access the state of an action. Components, such as menu items or toolbar buttons, which have been attached to the same action will share some properties. Changes will be propagated to all listeners. For example, if the action becomes disabled by calling `setEnabled(false)`, then all components that use this action will also be disabled.

Say, we have an action attached to a menu item, and change the state from “Enabled” to “Disabled” using `setEnabled(false)`, the menu item will change its look appropriately upon a property change notification. For example, `AbstractButton`, a base class of `JMenuItem`, and `JMenuItem` provide a method for creating a property-change listener that changes the look of the control to enabled or disabled if the state of an attached action changes.

PCoC Tasks provide a generic state concept for Activities (see Section 4.3.7). There are some predefined states, such as “Disabled”, and “Enabled”. However, Activity states can be defined arbitrarily, since they are simple strings. Another useful state is, for example, “Active”, denoting a checkmarked state of a menu item.

Some components, such as `JButton` and `JMenuItem`, have constructors that take an action as argument and use its properties for their construction. Some Swing container components, such as `JToolBar` and `JMenu`, have `add` methods that take an action as an argument and create a component, such as `JMenuItem`, from that action.

6.5.3 Using Actions for Forwarding

For actions that extend `AbstractAction`, we only have to implement the `actionPerformed` method. As described earlier, an action may be attached as an `ActionListener` to other objects, such as menu items, toolbar buttons, etc. The event source calls `actionPerformed` for all attached listeners and passes an action event instance which holds information such as the source of the event. The distribution of an action event may be initiated by a user selecting a menu item of which the action is a listener.

In some cases we may want to forward an action event to an intermediate object instead if handling the event in the action itself. The delegates of the action would do the actual work, while the action would be used only as a dispatch instance. This concept makes sense if behavior and/or data must be shared among many actions, such as editing commands (`copy`, `paste`), or if an abstraction layer is needed for services, such as file system operations.

Program 6.5-7 Command dispatch using action events

```
public MyDispatchAction extends AbstractAction {
   public MyDispatchAction(String name) {
      super(name);
      putValue(Action.ACTION_COMMAND_KEY, name);
   }

   void addActionListener(ActionListener listener) {
      if (fListeners != null) {
         fListeners = new EventListenerList;
      }
      fListeners.add(ActionListener.class, listener);
   }

   void removeActionListener(ActionListener listener) {
      if (fListeners == null) {
         return;
      }
      fListeners.remove(ActionListener.class, listener);
      if (fListeners.length == 0) {
         fListeners = null;
      }
   }
...  // see below
```

The following method is called when an action is invoked.

Program 6.5-8 Dispatching an action event

```
...  // see above
   /** Forwards the ActionEvent to all registered listeners.
     *
   public void actionPerformed(ActionEvent evt) {
      if (fListeners != null) {
         Object[] listeners = fListeners.getListenerList();

         // Create a new ActionEvent with the original source and the
         // command property associated with this action
         ActionEvent e = new ActionEvent(evt.getSource(), evt.getID(),
                                  (String)getValue(Action.ACTION_COMMAND_KEY));
         // forward the new action event object to all listeners
         for (int i = 0; i <= listeners.length-2; i+=2) {
            ((ActionListener)listeners[i+1]).actionPerformed(e);
         }
      }
   }
   private EventListenerList fListeners;
}  // MyDispatchAction
```

In Program 6.5-7 and Program 6.5-8 we generate a new action event object and forward it to all listeners of this action.

The next example shows how such dispatcher actions can be used.

Program 6.5-9 Using dispatch actions

```
public class MyMenuManager {
   public MyMenuManager(MyActionProvider provider) {
      fMenuBar = new JMenuBar();
      JPopupMenu menu = new JPopupMenu();
      ArrayList actions = provider.getActions();
      for (int i=0; i<actions.size(); i++) {
         menu.add(actions.get(i));  // add action provider actions
      }
      fMenuBar.add(menu);
   }
   public JMenuBar getMenuBar() { return fMenuBar; }
   JMenuBar menuBar;
}

public class MyActionDelegate implements ActionListener {
   public void actionPerformed(ActionEvent evt) {
      String command = evt.getActionCommand().toLowerCase();
      if (command.equals("cut")) { ... }               // do the actual work
      else if (command.equals("copy")) { ... }
      else if (command.equals("paste")) { ... }
   }
}

public class MyActionProvider {
   public MyActionProvider() {
      fActions = new ArrayList();
      fActions.add(new MyDispatchAction("Cut"));  // create some dispatch actions
      fActions.add(new MyDispatchAction("Copy"));
      fActions.add(new MyDispatchAction("Paste"));

      MyActionDelegate mydelegate = new MyActionDelegate();
      for (int i=0; i<fActions.size(); i++) {          // register mydelegate as
         actions.get(i).addActionListener(mydelegate); // handler for all actions
      }
   }
   ArrayList getActions() { return fActions; }
   ArrayList fActions;
}
```

As expected, the menu manager just iterates over the list of actions and associates menu items with them. Of course, the GUI properties could be taken from a configuration so that the actions really only act as dispatch objects responsible for forwarding events to delegates which do the real work.

`MyActionProvider` creates some dispatch actions that are used by the menu manager. In this case we attach an instance of `MyActionDelegate` as listener and actual handler of the action events; there may also be other actions in other objects that use the same handler. This design allows sharing of behavior across different classes.

Now, let us deploy these classes in an application:

Program 6.5-10 Using action providers in an application

```
public class MyDelegateDriver {
   public MyDelegateDriver() { }
   public static void main(String[] args) {
      MyActionProvider actionProvider = new MyActionProvider();
      MyMenuManager menuManager = new MyMenuManager(actionProvider);
      JFrame myframe = new ...
      ...
      myframe.setJMenuBar(menuManager.getMenuBar());
      myframe.pack();
      myframe.setVisible(true);
   }
}
```

First, we create an action provider—an object delivering a list of actions. Then, we instantiate a menu manager, which creates menus and a menu bar and associates the menu items with the actions delivered by the action provider. Finally, we associate the menu bar with a new frame, and show the frame including the menu bar. This code fragment shows how to separate GUI from functional code. The action provider could equally be deployed completely without GUI.

With PCoC, the message dispatch concept is separated from GUI-specific data and behavior. Dispatchers are responsible for forwarding and delegation (the latter is not supported by Swing actions) and contain no GUI-specific data and code (see Sections 4.4 and 5.6). Tasks are used mainly for GUI-specific purposes, and therefore are associated with GUI-specific definitions from the configuration (see Sections 4.5 and 5.7).

6.5.4 Remarks

The concept of Java Swing `Actions`, respectively `ActionListener`s, is powerful and important. It allows to a certain level the separation of interactive (GUI) and functional part of a component.

Actions can be used for the following domains:

- Enable or disable a set of controls based on actions.
- Provide descriptions of the action for menus, toolbars, and for the status bar in response to mouse gestures.
- Forward action events of actions to other objects attached as listeners.
- Abstraction of dispatchers and delegates for forwarding method calls.

Though this concept is very versatile, it also has some limitations. Actions are, as opposed to PCoC Dispatchers and Tasks, not designed to pass arguments other than action events to their listeners. Of course, an action could assemble arguments from its environment and forward them together with a command name in an instance of a new action event class to its listeners.

One problem is also that actions, in contrast to PCoC Dispatchers and Activities, carry some information for GUI elements. Most literature praises the power of `Action` to decouple GUI elements from functional logic. This is only true to a certain level. Actions are designed to be attached to menus, or toolbars, otherwise it would not make sense for them to hold a description string and icon name. Although actions can be used for other things, their support for GUI-specific properties would be an overkill.

Although actions can be composed to groups, they miss some abstraction layers. There is no general concept for grouping actions.

States are limited to “Enabled” and “Disabled”, whereas the set of PCoC Activity states is not limited at all.

Actions have no built-in priority management (focus management) that changes the order of listeners with focus changes in the UI, and the action event may be sent selectively to either only one receiver or all receivers (broadcast).

Actions cannot be configured to take explicit arguments. Think of menu items and toolbar buttons for commands such as `cut` and `copy`, or one to show context help for a selection in a text view. They may take the current selection as argument and do something with it. In the case of actions, each component providing such an action would have to get the current text selection itself and copy it into the clipboard. In the case of PCoC, one would define the Activities `cut`, `copy`, and `showContextHelp` so to expect a selection as argument. Any component could provide selections for them. There would be only one configuration for each of these Activities, that says that they are coupled with Activity `getSelection`, no matter by which component(s) it is provided. Whenever a component is added that supports an Activity `getSelection` to the application, the existing Activities `cut`, `copy`, `showContextHelp` would automatically use this new Activity when the new component has focus (the highest priority). There would not be the need to add listeners to each type of action that operates on selections.

PCoC aims at more modularity. It is possible to define Activities that deliver objects such as selections, etc., and to define Activities that need one or many objects as arguments. It is also possible to combine Activities, respectively Dispatchers, to Tasks. For example, one can combine Dispatchers `Selection getSelection()` with `copy(Selection)` to a Task `copy(getSelection())`. In PCoC, such connections are made only by configuration. The framework automatically registers and unregisters listeners, etc.

A big advantage of Java Swing actions is the simple concept. The concept can be understood very quickly. It is more difficult to understand the PCoC concepts.

6.6 Method Pointers in C++

The C++ standard ([ISOCPP98], page 83) defines member pointers. This feature is rather unknown.

Program 6.6-1 Method pointers in C++

```
class MyClass {
   void foo(int i) {};
}
void (MyClass::* x)(int) = &MyClass::foo;
MyClass* o = new MyClass();
o->*x(10);
MyClass o2;
o2.*x(10);
```

In this case, `x` is a pointer to a method of `MyClass` with `void` as return value and `int` as argument. `foo` can be a static or non-static method. If it is a static method, `x` is a plain function pointer. Operators `->*` and `.*` provide access to the member associated with `x`. In the example, we pass the integer value `10` as argument to `foo`.

As opposed to method objects in Java or C#, which are used for dynamic method invocation, this construct is fully typed at compile time. With dynamic method invocation, argument types are checked at runtime (see Sections 6.7.4, 6.7.6, and 6.4.4).

An extended version of this method pointer concept is available in the form of delegates with the .NET platform (see Section 6.7.5).

For more information about dynamic type information of C++ in the ANSI C++ standard, see [ISOCPP98] on Pages 71ff, 128, and 341ff, and in [STROU97] on Pages 407ff.

6.7 Microsoft .NET

This section describes reflection and dynamic method dispatch mechanisms of Microsoft's .NET architecture, and shows some critical issues for its use in real applications.

6.7.1 Overview

Microsoft introduced the first release candidate of its new technology product .NET at the end of the year 2001. What Microsoft calls a framework is a combination of a so-called Common Language Runtime (CLR), class libraries, services, and a set of programming languages and utilities that support software development.

.NET introduces a new programming language called C# that is said to combine the best of C++ and Java. Other supported languages are Visual Basic, Managed C++ which is an adaptation of the C++ programming language, and a Java-clone called Visual J++, and a JavaScript-like programming language called JScript.

The core of a .NET environment is called Common Language Infrastructure (CLI), and Microsoft's concrete implementation Common Language Runtime (CLR).

> "At runtime, the common language runtime is responsible for managing memory, starting up and stopping threads and processes, and enforcing security policies, as well as satisfying any dependencies that one component may have on any other component." - [MSNET01] on Page 13.

Basis for this environment is a common standard for object oriented programming languages (Common Language Specification) and a common type system (Common Type System). Compilers of .NET do not create machine dependent binaries, but object-code in the Microsoft intermediate language (MSIL). This code is then translated and optimized by a just-in-time compiler to be executed natively on a specific target machine. A garbage collector automates memory management.

The common type system (CTS) provides a programming language-independent concept of data types, enabling many more programming languages to share data and code on the .NET platform. In this case, Microsoft learned from their mistakes in the past—for example the difficulties when sharing data among different programming languages when using COM, DCOM, COM+, etc. For more information about the CTS, see [MSCTS01] on Microsoft's .NET Readiness Kit CD.

For Web services, .NET supports protocols such as HTTP, XML, SOAP. SOAP is an XML-based messaging protocol, which, by the way, will also be supported by PCoC.

.NET also supports security mechanisms for communication between components.

More relevant for this thesis is the improved support for name spaces and name-space hierarchies. A name space uses a "dot syntax" notation to logically group related classes together. For example, `System.Web.Services` conveys that contained classes provide functionality that is somehow related to Web services. When organizing classes this way it is easy to understand what functionality they provide. Actually this is the same reason why the concept of Activity Set containment hierarchy was introduced to PCoC.

The following sections should provide an overview of reflection facilities supported by .NET and its programming languages. We will concentrate mostly on C# for simplicity.

Compare .NET reflection to Java reflection facilities in Section 6.4, and especially Section 6.4.4.

6.7.2 Critical issues of .NET

Software companies are often not able to change existing architectures, because of the effort and risk that have to be invested. Although they may need some of the concepts, they may still be content with their existing application design. They may only need parts of various frameworks. So, a crucial question is whether frameworks are modular in design, i.e., if parts of the frameworks can be deployed independently.

In the case of .NET, the framework facilities can hardly be deployed independently. Developers who want to use single features have to use the whole architecture which is huge. Specific programming languages or language extensions (Managed C++, C#, Visual Basic) are required to make use of the .NET features. We may have the same problem with other architectures or frameworks, e.g. Java and PCoC. However, PCoC is quite small, and can easily be deployed with existing (C++ or Java) applications. The framework can be used without GUI extensions (Tasks, configuration, etc.), in the case that only the delegation facilities (Dispatchers, Activities, Activity Sets) are required.

On the other hand, a big advantage of .NET is that it supports almost everything that is needed to create a modern application. It supports interoperability between components written in different programming languages, it provides a security concept, class libraries (collections, strings, etc.), and much more.

Another critical issue is the availability of a framework on different platforms. Companies providing their software for different platforms, cannot use proprietary programming languages. Reasons are the higher development costs caused by maintenance effort for different source code on different platforms, extensive developer education, higher effort for integration of a common architecture for all supported platforms, etc. There are already (open source) projects such as Mono for Linux to support .NET on other platforms than Microsoft's operating systems. See [MONO03].

6.7.3 Language Independent Object Model of .NET

COM and COM+ provides only a minimal set of properties of different programming languages. With COM, it is, for example, not possible to use C++ internal data structures in other programming languages. This causes a limited interoperability with other programming languages unless a custom serialization mechanism is put on top of these interface standards. See also [iX1201], Pages 123f, and 128.

With .NET, Microsoft introduces a language-independent object model. There are value types and reference types that are stored on the heap. The former are used for basic data types, enumeration types, etc., the latter for classes, delegates (type-safe function pointers), and pointers. See [iX1201] on Pages 123f, [WESTP01] on Pages 112f, and [MÖSS02].

6.7.4 Overview of C#

Microsoft introduced a new programming language called C# (C-sharp) with .NET. It combines some features of C++ and Java and adds some new ones. Microsoft states that C# is the first component-oriented programming language available. See [MSCSH01] on Microsoft’s .NET Readiness Kit CD, Page 7, for detailed information.

The most interesting features are, beside the garbage collection provided by the .NET Common Language Infrastructure for all .NET languages, enumeration types (currently not supported by Java; will be introduced in Java 1.5), properties (value assignment of member variables as setter and getter methods; see Section 6.7.8) and indexers (special case of properties; see Section 6.7.8), attributes (meta information that can be attached to program elements such as classes, methods, etc.; see Section), and delegates. A concept called *boxing* allows to keep primitive data types such as `int` compatible to `Object`.

The reflection mechanism supports retrieving and using types, methods, fields, properties, and others at runtime. See also [MICROS01] on Microsoft's .NET Readiness Kit.

Methods are no first class objects in C#, or in .NET in general. When method pointers and dynamic invocation is required, one can use delegates, or .NET reflection facilities for dynamic invocation (`GetMethod`).

The following sections offer an insight into some .NET and, mainly, C# capabilities.

There is a good comparison of C#, Java, and C++ in [BREYM02] (in German) on Pages 98ff; a comparison table is on Page 105.

6.7.5 .NET Delegates

6.7.5.1 Overview

With the .NET platform, Microsoft has introduced a powerful concept of type-safe method pointers called delegates. A .NET delegate represents a class derived from `MulticastDelegate` rather than a raw memory address. The main differences to C++ method pointers are:

- A delegate stores the target object (the receiver) if associated with a non-static method. However, delegates can also be associated with static methods. In this case, the target object is not set.
- A delegate can be associated with more than one method, for so-called multicasts.

It is quite common that objects engage in two-way conversation. To cope with threading issues, callback methods are frequently used. In object-oriented languages these must also be able to point to non-static methods (object methods). This is where delegates are useful. Although the C++ standard already supports method pointers to non-static methods (see Section 6.6), .NET delegates are easier to use and more extensible.

In contrast to method objects provided by the reflection mechanisms of Java and .NET, delegates are first class objects. The advantage of .NET delegates compared to dynamic method calls using reflection, and to PCoC, is their type-safety at compile-time. This helps to find wrong argument types earlier than with dynamic method invocation.

PCoC Dispatchers are basically similar to delegates (they also dispatch requests to methods and support multicasts) and to method objects provided by the Java and .NET reflection (they are invoked dynamically; argument types are checked at runtime). However, as opposed to delegates, Dispatchers support delegation, and not only forwarding.

Activities and their Dispatchers can be assembled at runtime, and can be added and removed at any time. So-called remote Activities can be attached to an Application using XMLRPC (and also RMI when using Java).

With .NET it is also possible to define (or better assemble) behavior at runtime. Assemblies, classes and methods can be defined completely at runtime using the `System.Reflection.Emit` name space. See [LIBERT01], Chapter 18. PCoC is not that powerful (the implementation of Activities can only be defined at runtime if they are attached via a remote connection), but this makes, on the other hand, the framework easier to use.

The capability for adding Activities or methods at runtime is useful if some functionality of a component should be loaded, or better extended via a remote connection or through configuration at runtime.

Delegates were first introduced in Microsoft's Visual J++, also designed by Anders Hejlsberg, and were a cause of much technical and legal dispute between Sun and Microsoft.

Sun states in [SUNDLG99] that bound method references such as delegates add complexity, the concept results in loss of object-orientation, delegates are limited in expressiveness—they are allegedly nothing more than function pointers (which is not true; see the description of delegates above), and they are no more convenient than adapter objects. Note that these comments do not reflect the opinion of the author. With these results in hand, the designers of the Java programming language decided to introduce inner classes, which can be used for purposes where usually delegates are used in .NET, for example, for call-backs, etc.

Sun also states in this article that any implementation of bound method references would either be inefficient or non-portable. A portable implementation of delegates that generates code capable of running on any compliant implementation of the Java platform would have to rely on the standard Java Core Reflection API.

Microsoft states in [MSDLG99] that delegates are simpler and more efficient and portable than inner classes. They also say that delegates are better for multicasting, and reduce the number of classes drastically, which leads to smaller binaries.

For more information, see Sun's original article [SUNDLG99], and Microsoft's response [MSDLG99].

6.7.5.2 Using Delegates

Delegates are declared using the `delegate` (C#, Visual Basic) or `__delegate` (Managed C++) keyword.

Let us compare delegates to Java action listeners in the following example.

Program 6.7-1 Using an ActionListener in Java

```
public class MyClass implements ActionListener {
   public MyClass() {
      button = new JButton("Test");
      button.addActionListener(this);
   }

   public void actionPerformed(ActionEvent e) {
      ...
   }
}
```

In the constructor of `MyClass`, we create a new `JButton` instance and add `this` as action listener. `JButton` holds a list of action listeners and notifies each, when the button is clicked. More precisely, it calls the method `actionPerformed`. As action listener, `MyClass` must implement the `ActionListener` interface, including the `actionPerformed` method.

Here a semantically equivalent implementation using delegates:

Program 6.7-2 Declaring a delegate in C#

```
public delegate void ActionListener(ActionEvent e);
public class Button {
   public ActionListener click;
   ...
}
public class MyClass {
   public MyClass() {
      button = new Button();
      button.click += new ActionListener(ActionPerformed);
   }
   public void ActionPerformed(ActionEvent e) { ... }
}
```

In this C# code fragment, we declare a delegate `ActionListener`. In the `Button` class, we use this delegate to hold and notify listeners of button clicks (in contrast to `JButton` in Program 6.7-1, where we use a list of action listeners). In the constructor of `MyClass`, we add a new `ActionListener` delegate which is associated with the method `ActionPerformed`. Note that the return type and parameter types of the delegate and the associated method must match.

With the Java action listener concept, we can add and remove listeners (listener objects) arbitrarily. All listeners must implement the `ActionListener` interface, including the method `actionPerformed`. With .NET, all methods which match a delegate's return type and parameter types can be added to and removed from the delegate.

Let us assume that we want to implement a logging facility that takes a message id and a string message as parameter:

Program 6.7-3 Declaring a delegate in C#

```
// This delegate actually represents a class encapsulating a function pointer to
// some method taking an integer and string argument and returning void.
public delegate void Logger(int id, string msg);
```

Such a delegate can be declared inside of a class or stand-alone. The given delegate actually represents a class encapsulating a function pointer plus a receiver to some method taking an integer and string argument and returning void.

Note that since beta 2 of .NET, all delegates are multicast delegates, therefore able to send events to one or many targets.

A compiler implicitly generates a delegate class for each delegate declaration. These classes are always derived from `System.MulticastDelegate` (C#, Visual Basic), respectively `System::MulticastDelegate` (Managed C++), and accordingly in other programming languages.

The C# compiler produces the following class from the declaration above:

Program 6.7-4 A delegate class in C#

```
public class Logger : System.MulticastDelegate {
   Logger(object target, int ptr);
   // The synchronous Invoke method
   public virtual void Invoke(int id, string msg);
   // The asynchronous version
   public virtual IAsyncResult BeginInvoke(int id, string msg,
                                           AsyncCallback cb, object o);
   public virtual void EndInvoke(IAsyncResult result);
}
```

This class is only created implicitly by the compiler; this is not source code.

Now that we have declared the delegate, we can make use of it. The following class, respectively its method, writes messages to the console:

Program 6.7-5 A console logger in C#

```
public class ConsoleLogger {
   public ConsoleLogger() { }
   void PrintMessage(int id, string msg) {
      Console.WriteLine("Message {0}: {1}", id, msg);
   }
}
```

Note that the `PrintMessage` method has the same parameter and result types as the delegate.

The following logger writes to a file:

Program 6.7-6 A file logger in C#

```
public class FileLogger {
   public FileLogger(string logfilepath) {
      FileInfo f = new FileInfo(logfilepath);
      f.Open(FileMode.OpenOrCreate,
             FileAccess.ReadWrite, FileShare.None);
      sw = f.AppendText();
   }
   ~FileLogger() { sw.Close(); }
   void LogMessage(int id, string msg) {
      sw.WriteLine("Message {0}: {1}", id, msg);
      sw.Flush();
   }
   private string logfilepath;
   private StreamWriter sw;
}
```

Note that we are intentionally using another method name (`LogMessage`) to demonstrate that only parameter and result types are relevant to delegates.

Now we implement a driver class, our application entry point:

Program 6.7-7 Our main class in C#

```
public class MyApp {
   public static int Main(string[] args) {
      ConsoleLogger cl = new ConsoleLogger();

      Logger logger = new Logger(cl.PrintMessage);  // logs to console
      logger(100, "Sample Message");
   }
}
```

In Program 6.7-7, we create a logger delegate from the `PrintMessage` method of a `ConsoleLogger` instance. When we invoke the logger, the call is actually forwarded to `cl.PrintMessage(100, "Sample Message")`.

For static methods, the target is not specified (cf. object `cl` is in the example above).

Program 6.7-8 A static method as callback

```
public static void StaticLog(int id, string msg) { ... }
Logger logger = new Logger(StaticLog);
```

Program 6.7-8 shows nothing but the use of a delegate as an ordinary function pointer. However, delegates can also be useful for other things such as multicasting to different methods, which is not possible with C(++) function pointers.

Compare delegates to C++ method pointers in Section 6.6.

Now back to our example. If we want to log to the console and to a file, we can simply do the following:

Program 6.7-9 Our main class in C#

```
public class MyApp {
   public static int Main(string[] args) {
      FileLogger fl = new FileLogger("c:/test.log");
      ConsoleLogger cl = new ConsoleLogger();
      Logger consoleLogger = new Logger(cl.PrintMessage); // logs to console
      Logger fileLogger = new Logger(fl.LogMessage);      // logs to a file
      Logger logger = (Logger)consoleLogger+fileLogger;
      logger(100, "Sample Message");           // logs to console AND a file
   }
}
```

In Program 6.7-9, respectively Program 6.7-30, the methods of the console logger and the file logger are called accordingly. To log to the console only, we can remove the file logger again:

Program 6.7-10 Removing a delegate

```
Logger logger = (Logger)consoleLogger+fileLogger;
logger(100, "Sample Message");                  // logs to console AND a file
logger -= fileLogger;
logger(100, "Sample Message2");                 // logs to console only
```

6.7.5.3 Remarks

The delegate concept was introduced as part of Microsoft's .NET platform. Delegates are method pointers that are more powerful than function pointers of C and C++. Each .NET programming language supports this concept.

One important feature of .NET delegates is the ability to use non-static methods as callbacks. The C++ standard already supports method pointers to non-static methods, but delegates are easier to use. Another important feature of delegates is the ability to combine methods to multicasts—if such a delegate is invoked, it forwards calls to all attached methods. The advantage of delegates as opposed to dynamic method calls using the Java or .NET reflection is their type safety at compile time.

In contrast to method objects provided by the reflection mechanisms of Java and .NET, delegates are first class objects. The advantage of .NET delegates compared to dynamic method calls using reflection, and to PCoC, is their type-safety at compile-time. This helps to find wrong argument types earlier than with dynamic method invocation.

PCoC Dispatchers are basically similar to delegates (they also dispatch requests to methods and support multicasts). However, as opposed to delegates, Dispatchers support delegation, and not only forwarding.

When invoking a Dispatcher one specifies the Activity Set, the name of the method and an array of arguments. Dispatchers automatically combine methods of the same name and signature, as opposed to delegates, which can be used to explicitly combine methods with a specific parameter and result types signature. Both can be used to make single or multicasts (broadcast to attached methods). Both allow attaching and detaching callback methods at runtime. In the case of PCoC the callbacks are again Dispatchers or Activities (operations that actually perform requests). See also Section 4.4.

For a good and detailed description of C# delegates, see [TROEL01] on Pages 250ff, and [LIBERT01] on Pages 277ff. The latter describes delegates also for Managed C++ and Visual Basic.

6.7.6 .NET Late Binding and Dynamic Method Invocation

6.7.6.1 Overview

Like Java, .NET supports reflection facilities such as dynamic exploration of types and dynamic method invocation. Late binding is a mechanism that allows resolution of the existence and name of a given type and its members at runtime rather than at compile time. Once the presence of a type has been determined, we can dynamically invoke methods, access properties, and manipulate the fields of a given entity.

Read about dynamic method invocation of Java in Section 6.4.4.

6.7.6.2 Dynamic Method Invocation

The following example illustrates dynamic method invocation in .NET. We use the `System.Activator` class to instantiate the class `MyClass`, which we have put into an assembly `MyAssembly`.

Program 6.7-11 Class instantiation in .NET

```
public class MyApp {
   public static int Main(string[] args) {
      // Load MyAssembly
      Assembly a = null;
      try { a = Assembly.Load("MyAssembly"); }
      catch(FileNotFoundException e) { Console.WriteLine(e.Message); }

      // Get MyClass
      Type cl = a.GetType("MyAssembly.MyClass");
      // Create an instance of MyClass
      object o = Activator.CreateInstance(cl);
      ...
   }
}
```

Now, `cl` points to a class `MyClass` loaded from assembly `MyAssembly`. `o` points to a new instance of this class.

Now, let us take a look at the implementation of `MyClass`:

Program 6.7-12 Class MyClass

```
public class MyClass {
   public void foo() { Console.WriteLine("Hello World!"); }
}
```

For simplicity of this example, this class contains only one method without any parameters.

Next we retrieve a `MethodInfo` object for method `foo`, which we can then invoke by using the `MethodInfo.Invoke` method.

Program 6.7-13 Dynamic method invocation in .NET

```
public class MyApp {
   public static int Main(string[] args) {
      ...
      // Get MethodInfo
      MethodInfo mi = cl.GetMethod("foo");
      // Invoke method without arguments
      mi.Invoke(o, null);
      return 0;
   }
}
```

Let us assume that `MyClass` also has a method that takes a string and an integer parameter.

Program 6.7-14 Class MyClass (2)

```
public class MyClass {
   public void foo() { Console.WriteLine("Hello World!"); }
   public void printMessage(int id, string msg) {
      Console.WriteLine("Message {0}: {1}", id, msg);
   }
}
```

The invocation procedure looks a little more complicated:

Program 6.7-15 Dynamic method invocation with arguments in .NET

```
// Get MethodInfo
MethodInfo mi = cl.GetMethod("printMessage");
object[] arguments = new object[2];
arguments[0] = 100;
arguments[1] = "A Message";
// Invoke method with arguments
mi.Invoke(o, arguments);
```

If the invoked method returns a value, it has to be casted to the correct type.

To clearly determine the right method, we may want to specify the full signature of the method to be invoked:

Program 6.7-16 Dynamic method lookup with parameter types

```
// Get MethodInfo
Type[] parameterTypes = new Type[2];
parameterTypes[0] = Type.GetType("System.Integer");
parameterTypes[1] = Type.GetType("System.string");
MethodInfo mi = cl.GetMethod("printMessage", parameterTypes);
```

The similarities to Java should be apparent (cf. Section 6.4.4 about dynamic method invocation in Java).

6.7.6.3 Remarks

Late binding and dynamic method invocation is mostly needed for component interoperability and dynamic (re-)use. Almost every modern programming language such as Java and the .NET platform languages support reflection, which is necessary for dynamic method invocation.

We remember that PCoC Activities are method objects and include references to their providers. Activities can be added to/removed from a component all at runtime. Dispatchers are similar to method pointers and delegates, and delegate requests to Activities. When invoking a Dispatcher, we specify the Activity Set, the name of the method and an array of arguments. See also Section 4.4. The PCoC constructs are not type safe at compile time, but at runtime.

6.7.7 Name Spaces on .NET

6.7.7.1 Overview

When developing applications, it may be useful to group semantically related types into custom name spaces. For this reason, each .NET platform language supports name spaces, and Java supports them with the package concept.

In C#, name spaces can be defined using the `namespace` keyword.

6.7.7.2 Using Name Spaces

The following example corresponds to the example in [TROEL01] on Pages 23-24.

Program 6.7-17 Using name spaces in .NET

```
// Hello world in C#
using System;
public class MyApp {
   public static void Main() { Console.WriteLine("Hello World!"); }
}
```

In this case, we use name space `System` where the `Console` class is defined. There are only slight syntactical differences between the various .NET programming languages, which is elegant for developers using .NET.

Program 6.7-18 A stack implementation

```
using System;
using System.Collections;
namespace OriginalNamespace {
   public class MyStack {
      private Stack stack;
      public MyStack() { stack = new Stack(); }
      public void Push(int i) {
         if (i >= 0) { stack.Push(i); }
      }
      public int Pop() {
         if (stack.Count > 0) { return stack.Pop(); }
         else { return -1; }
      }
      public int Peek() {
         if (stack.Count > 0) { return stack.Peek(); }
         else { return -1; }
      }
      public int Size() { return stack.Count(); }
   }
}
```

Name spaces are a way to group semantically related types such as classes, enumerations, interfaces, delegates, and structures. They are also useful to distinguish different implementations of syntactically equal types.

The stack implementation of Program 6.7-18 allows adding of positive numbers; we use the `Stack` class of the name space `System.Collections`.

Now we may want to add trace capabilities to this class, but only for debug reasons. In the release version of our application we may not want such time-consuming checks. In the following code fragment, we do not change the original implementation, but rather override it in another name space:

Program 6.7-19 Added tracing capability for the original stack implementation

```
namespace MyDebugNamespace {
   class MyStack : public OriginalNamespace.MyStack {
      private Stack stack;

      public void Push(int value) {
         int prevsize = Size();
         Console.WriteLine("MyStack.Push: {0}", value);
         base.Push(value);
         if (Size() != prevsize+1) {
            Console.WriteLine("Error in MyStack.Push: value not added");
         }
         else if (base.Peek() != value) {
            Console.WriteLine("Error in MyStack.Push: wrong last element");
         }
      }
      public int Pop() {
         int value = 0;
         if (Size() > 0) {
            int prevsize = Size();
            value = base.Pop();
            Console.WriteLine("MyStack.Pop: {0}", value);
            if (Size() != prevsize-1) { Console.WriteLine(
                       "Error in MyStack.Pop: value not removed correctly"); }
         }
         else {
            Console.WriteLine("Error in MyStack.Pop: stack is empty");
         }
         return value;
      }
   }
}
```

Now we have a stack class that overrides the original class and adds some tracing. Our client code might look like the following:

Program 6.7-20 Using name spaces

```
using OriginalNamespace;   // Original MyStack without tracing
// using MyDebugNamespace; // MyStack with tracing

MyStack stack = new MyStack();
stack.Push(10);  // push an integer value into the stack
```

Depending on the name space we use, either the original implementation of `MyStack` is taken, or that with tracing capabilities. The client code can stay the same in any case. During the implementation and test phase of source code we may use the name space with added tracing, and for the release version of the application, the original name space.

Note that Java supports a package concept. A Java package is a named scope and can be imported by using the `import` keyword. For example, `import java.lang.*`.

Name spaces are used often for aspect-oriented programming, where they can help to simplify the replacement of implementations, or the sharing of aspects among many classes. See also Section 6.2.

6.7.7.3 Name-Space Aliases

A cleaner and more convenient approach to resolve name space ambiguity are name-space aliases. For example, we can use them instead of the name-space type definitions of Program 6.7-20:

Program 6.7-21 Name-space aliases in C#

```
// A name-space alias
using MyStack = MyDebugNamespace.MyStack; // or: using MyDebugNamespace

MyStack stack = new MyStack();
stack->Push(10);  // push an integer value into the stack
```

The following code fragment illustrates the support of aliases in PCoC:

Program 6.7-22 Name-space aliases in C#

```
PCCActivitySet a = PCCRegistry.getOrCreateActivitySet("A");
PCCRegistry.setAlias("MyStack", "A");       // or: a.setAlias("MyStack")
a = PCCRegistry.getActivitySet("MyStack");  // returns Activity Set "A"
```

6.7.7.4 Nested Name Spaces

Like Java packages, .NET name spaces can be nested.

Program 6.7-23 Nested name spaces in C#

```
namespace MyApp {
   namespace MyServices {
      class Assert { ...
      }
      class Tracing { ...
      }
   }
   namespace MyWidgets {
      class TableView { ...
      }
      class GraphView { ...
      }
   }
}
```

Now, the name spaces can be imported selectively:

Program 6.7-24 Name-space aliases in C#

```
using MyApp.MyServices;
using MyApp.MyWidgets;
```

Almost every modern programming language supports nested name spaces. In PCoC, we use Activity Sets. They are named scopes like name spaces, but can be created at runtime. See also Sections 3.3 and 5.8.

6.7.7.5 Remarks

The concept of name spaces or packages, including nesting, is supported by almost all modern object-oriented programming languages. Name spaces allow grouping of different, mostly semantically related types. They can also be used to vary aspects of source code simply by using the one or other name space defining the same types, but with different implementations.

In short, name spaces can help to increase modularity, reusability, and to replace implementations of aspects by others without affecting client code.

We may encounter situations where these facilities are needed even at runtime. The name spaces supported by .NET are static. Once a .NET program is compiled and executed, it is no longer possible to reorganize its name spaces.

With PCoC Activity Sets, the features of name spaces can be achieved at runtime.

See also Sections 3.3, 5.3, 5.4.3, and 5.8.

6.7.8 .NET Properties

Properties are a new concept introduced with C#. A property is not a regular data field of a class, but a method pair `set/get` that just looks like a field (see Program 6.7-25). It usually hides and encapsulates data fields. Properties are said to be a natural extension of data fields, and are known as smart fields in the C# community.

In the following example, we define the properties `x` and `y`:

Program 6.7-25 A C# property

```
public struct Point {
   double fx;
   double fy;

   public double x {
      set {
         fx = value;
      }
      get {
         return fx;
      }
   }
   public double y {
      set {
         fy = value;
      }
      get {
         return fy;
      }
   }
}
```

We can then access the properties as follows:

Program 6.7-26 Accessing a C# property

```
Point q = new Point();
q.x = 20; // set-method is called implicitly
Console.WriteLine(q.x); // get-method is called implicitly
```

`q.x` can be accessed like a data field, but actually the corresponding `set` and `get` methods are called. Usually there may be real data fields behind properties, like `fx` in Program 6.7-25.

Indexers are a special case of properties. They allow indexed access to user defined classes.

Program 6.7-27 C# indexer

```
public enum CoordinateDimension : short { X = 1, Y = 2 }
public double this[CoordinateDimension which] {
   set {
      if (which == CoordinateDimension.X) fx = value;
      else fy = value;
   }
   get {
      if (which == CoordinateDimension.X) return fx;
      else return fy;
   }
}
```

Program 6.7-27 shows the implementation of an indexer in the class `Point` of Program 6.7-25. The indexer is specified by using the keyword *this*. The parameter and return types can be selected arbitrary.

This concept is not available in Java. In C++ the same effect can be achieved by overloading the index-operator `[]`.

The following example shows the use of an indexer:

Program 6.7-28 C# indexer usage

```
Point q = new Point();
q[CoordinateDimension.X] = 20;
q[CoordinateDimension.Y] = 30;
```

The members are accessed by using indexes. The `set` and `get` methods are called implicitly.

See also [TROEL01] on Pages 147-149.

6.7.9 Remarks

The reflection facilities described in this chapter are only a small excerpt of what .NET actually provides. However, all facilities relevant for this thesis were explained, or at least mentioned, to get a rough overview what is possible with .NET. Another rarely described feature supported by .NET is *Expando*, an interface that allows to add and remove members at runtime:

Program 6.7-29 Expando (excerpt)

```
// System.Runtime.InteropServices.Expando:
public FieldInfo AddField(String name);
public MethodInfo AddMethod(String name, Delegate method);
public PropertyInfo AddProperty(String name);
public void RemoveMember(MemberInfo m);
```

The `System.Reflection.Emit` allows more dynamic things such as defining assemblies, modules, types, and methods completely at runtime.

The following code fragment illustrates a possible use of the `System.Reflection.Emit` name space.

Program 6.7-30 Dynamically creating a class and a method (C#) (1)

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class TestFoo
{
   Type t;
   public TestFoo() {
      MakeFoo();
      UseFoo();
   }

   UseFoo()
   {
      object o = Activator.CreateInstance(t);
      t.GetMethod("Foo").Invoke(o, null);
   }
...
```

Program 6.7-31 Dynamically creating a class and a method (C#) (2)

```
...
   MakeFoo()
   {
      // Create a dynamic assembly in the current domain
      AssemblyName assemblyName = new AssemblyName();
      assemblyName.Name = "FooAssembly";
      AssemblyBuilder fooAssembly = Thread.GetDomain().DefineDynamicAssembly(
         fooAssembly, AssemblyBuilderAccess.Run);

      // Create a module in the assembly and a type in the module
      ModuleBuilder fooBuilder = fooAssembly.DefineDynamicModule("fooModule");
      TypeBuilder fooType = fooBuilder.DefineType("FooType",
                                             TypeAttributes.Public);
      // Add a Foo method to the type
      MethodBuilder fooMethod = fooType.DefineMethod("Foo",
                                 MethodAttributes.Public,
                                 null, null);
      // Generate the implementation for "Foo"
      ILGenerator ilg = fooMethod.GetILGenerator();
      ilg.EmitWriteLine("Hello World!");
      ilg.Emit(OpCodes.Ret);
      // Finalize the type so we can create it
      fooType.CreateType();
      // Create an instance of the new type
      t = Type.GetType("FooType");
   }
}
```

In the constructor, we first call the method `MakeFoo` which creates an assembly, a module, a type, and a method dynamically. We define the new method “`Foo`” to write “`Hello World!`” to the standard output. In `UseFoo`, we dynamically invoke the newly created method.

.NET provides powerful reflection facilities, equal or even surpassing those of Java. However, issues for its deployment may be the Microsoft licensing model, and whether it will be available on operating systems other than the Microsoft Windows family. In any case, the concepts are useful for modularity and reusability of software components.

More details about .NET attributes can be found in [TROEL01] on Pages 373ff, and [LIBERT01] on Pages 457ff.

As opposed to .NET delegates and name spaces, the PCoC approach is a purely dynamic one. As mentioned earlier in this thesis, the framework introduces dynamic facilities such as Activity Sets and the ability to add and remove operations at runtime. This allows to dynamically organize objects and components in groups and prioritized acquisition relationships, to apply different aspects to objects, to switch Activity Sets, and to modify objects (adding and removing members). These features may not always be needed, but they are useful for reusability and interoperability of objects or components.

PCoC does not provide support for fields, but only for operations, more precisely Activities. Fields must be accessed through operations. Note that even .NET properties encapsulate field access.

Compare .NET reflection to Java reflection facilities in Section 6.4, and especially Section 6.4.4.

6.8 Smalltalk

Smalltalk is a purely object-oriented programming language based on classes. It has affected or was even the basis for the design of many modern programming languages such as, for example, Java.

In Smalltalk, even primitive data types and methods are objects—so-called first class objects. Classes are also objects. All Smalltalk processing is accomplished by sending messages, like in Self (cf. Section 6.10).

Objects are instances of classes. The messages that an object can respond to are defined in the interface of its class. Methods define how messages are executed and represent a class behavior.

Sending a message to an object causes the method search to begin with the receiving object itself. If the method is not found in the class of the object, the method is searched for up the class hierarchy. The same is also valid for sending messages from within an object to itself using the identifier `self`.

See [FOOTE89], [GOLDB83], [IBMST95], and [GITTI00] for descriptions of the Smalltalk language and its concepts.

With PCoC, operations are exposed as Activities. They are treated as first class objects like methods in Smalltalk. With Dispatchers, PCoC offers a message dispatch mechanism that supports delegation. When a Dispatcher is invoked, it searches its acquisition parents for one or many Activities (like in Smalltalk where methods are searched in an object's class and then in its superclasses), gathers them in a distinct list and finally invokes them. As opposed to PCoC, Smalltalk does not support multicasts for the message dispatch. The structure of object sets can be modified at runtime by adding or removing Activities. In Smalltalk it is possible to add, remove and wrap methods at runtime. See Chapter 2 for a detailed comparison of the message dispatch mechanisms of Smalltalk and PCoC. Section 5.6 describes PCoC Dispatchers in more detail.

6.9 Oberon Message Objects

This section provides an overview of the message object concept proposed with Oberon-2, and how this relates to similar concepts in PCoC.

6.9.1 Overview

> “Oberon-2 is a general-purpose programming language in the tradition of Pascal and Modula-2 and was developed at the ETH Zurich. Its most important features are block structure, modularity, separate compilation, static typing with strong type checking (also across module boundaries), and type extension with type-bound procedures.
>
> Type extension makes Oberon-2 an object-oriented language. An object is a variable of an abstract data type consisting of private data (its state) and procedures that operate on this data. Abstract data types are declared as extensible records. Oberon-2 covers most terms of object-oriented languages by the established vocabulary of imperative languages in order to minimize the number of notions for similar concepts.” - [MÖSS91].

See Section 2.2.3 for an overview of Oberon message objects and interpreters.

6.9.2 Remarks

Message objects have some advantages over methods ([MÖSS98]):

- Messages are data packages; they can be stored and forwarded later on.
- A message object can be passed to a method (the message interpreter) that forwards it to different objects. This allows broadcasts, which are not or difficult to realize with methods.
- It is sometimes easier if the sender does not have to care about whether a receiver understands a message (implements a corresponding method) or not.

- It is also possible to access the message interpreter (handler) through a method reference, which enables to replace it by another at runtime.

Message objects also have disadvantages:

- The interface of a class does not reflect which messages can be handled by instances of the class. It is difficult to realize at compile time which objects are related through message forwarding at runtime.
- Messages are interpreted and forwarded at runtime, which is much slower than direct method calls. It depends on how fast dynamic type information is evaluated, or in the case of handling messages using their names, how fast strings are parsed.
- Sending message objects requires more code to be written. Arguments must be packed in the message object. A regular interface such as for methods is not available.
- Invalid messages are not recognized at compile time. Although this provides the flexibility to forward messages through objects that cannot handle them themselves, it can be troublesome to find errors.

Generally, one should use methods rather than message objects. However, in some cases it is really useful to use message objects.

PCoC also sends message objects, or kind of. Messages in PCoC consist of the following parts: the message path including the name of the Dispatcher that should be used, where the Dispatcher name corresponds to the name of the message that should be forwarded; some directives for delegating the message (for example, broadcast or single cast); an argument container (a Material instance); the method (`perform`, `getState`, etc.) which should be invoked on the corresponding Activities. Objects can register for specific messages. To do this, it is necessary to specify name and dynamic signature (parameter and return value types) of the message. For example, for a message with signature `void select(int from, int to)`, we would write:

Program 6.9-1 A PCoC Activity in Java

```
addActivity(new PCCSingleActivity("select", "SelectionCategory", "", "int,int") {
   ...
}
```

Note that this statement registers the receiver for messages with the name `select`. It does not represent an actual message object. The receiver itself is an operation implemented as an anonymous class derived from `PCCSingleActivity`. Whenever a message with name `select` is delegated to the Activity Sets of the object that registered the Activity, the Activity will be invoked, which itself can delegate the message or simply call a method of the surrounding object.

The interface definition ensures that the argument lists of all `select` messages sent to the current object will have the given interface; the invocation of Activities is type-safe at runtime.

PCoC Dispatchers act as what we called “handler” or “message interpreter” earlier in this section.

Receiver operations are also objects, because this provides the flexibility not only to handle messages at runtime, but also to extend receiver objects with operations at runtime. The state of receivers can also be handled with PCoC. Possible states are “Enabled” or “Disabled”, and any other strings. Implementations such as license management for operations can be encapsulated or at least simplified by this approach. With the PCoC architecture, it is possible to let objects dynamically acquire operations, or better, corresponding Dispatchers, from other objects. This improves code and object reuse and sharing.

Generally, all the advantages and disadvantages mentioned here for message objects are also valid for PCoC.

An introduction into message objects is given in Section 2.2.3. Read more about PCoC in Chapters 4 and 5, especially Sections 5.4 and 5.5. Information and references to Oberon-2 are available in [MÖSS91], [MÖSS98], [WIRTH92].

6.10 Self

One of the most closely related sources of this thesis is a former feature of the programming language Self: prioritized multiple inheritance. The feature was removed in version 3.0. This section gives an overview of this programming language, and how it is related to this thesis.

6.10.1 Overview

> “Self is an object-oriented language for exploratory programming based on a small number of simple and concrete ideas: prototypes, slots, and behavior. Prototypes combine inheritance and instantiation to provide a framework that is simpler and more flexible than most object-oriented languages. Slots unite variables and procedures into a single construct. This permits the inheritance hierarchy to take over the function of lexical scoping in conventional languages. Finally, because Self does not distinguish state from behavior, it narrows the gaps between ordinary objects, procedures, and closures. Self’s simplicity and expressiveness offer new insights into object-oriented computation.” - [UNGAR87].

Unlike other object-oriented programming languages, Self supports neither classes, nor variables. Instead, Self has adopted a prototype metaphor for object creation. Furthermore, while other object-oriented programming languages support variable access as well as message passing, Self objects access their state information solely by sending messages to “self”. The programming language is named after this messaging concept. Due to this approach, message passing is more fundamental in Self than in other programming languages. Methods and fields are stored in so-called slots. Methods are objects associated with code, as opposed to fields which contain data. Normally slots are readable, no matter if accessing data or methods. If there is a corresponding assignment slot (for example, “`x:`”) for a data slot (“`x`”), the member variable (“`x`”) is considered read-write. Microsoft lately introduced a similar concept called properties with its new platform .NET (about 15 years after Self showed up the first time). Read more about .NET properties in Section 6.7.8.

6.10.2 Prototyping vs. Class Inheritance

[UNGAR87] describes on Page 3 the differences between class-based programming languages and Self as follows:

facility	***class–based systems***	***Self: no classes***
inheritance relationships	instance of subclass of	inherits from
creation metaphor	build according to plan	clone an object
initialization	executing a “plan”	cloning an example
one-of-a-kind	need extra object for the class	no extra object needed
infinite regress	class of class of class of ...	none required

Creating new objects from prototypes is accomplished by a simple operation, copying. This approach has a simple biological metaphor, cloning. Creating new objects from classes as in class-based programming languages is accomplished by instantiation, which includes the interpretation of format information in a class. Instantiation is similar to building a house from a plan. Copying is often perceived to be a simpler metaphor than instantiation, since it is known from processes in nature.

Self allows multiple inheritance, which requires a strategy for dealing with multiple parents. It has block closure objects, and threads.

In prototype languages the structure of objects is not defined separately in classes. Existing objects can be copied and and extended and/or modified.

Let us take a look at the following example which illustrates differences to class-based approaches:

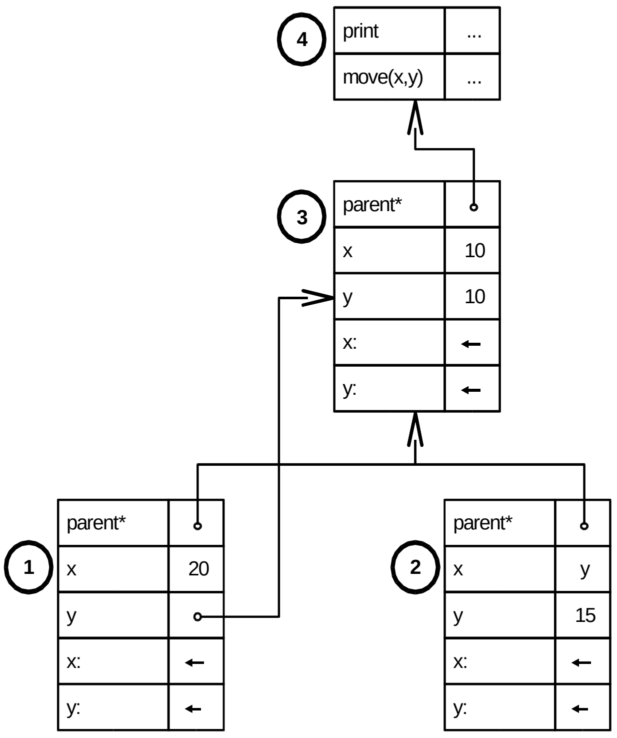


Figure 6.10-1 Self objects

Here we have some prototypical objects. These Self objects describe point structures. Each of them describes its own format, but some refer to members of other objects.

Object `1` is a copy of parent `3` and redefines slot (field) `x`. Slot `y` stays the same as in object `3`. Object `2` is also a copy of `3` and redefines both slots. Object `3` is a copy of `4`, thus it also has methods `move` and `print`, so do objects `1` and `2`. Each object has a pointer to its parent, except object `4`, which fully describes its own format.

Note that objects can have more than one parent object—Self supports multiple inheritance (through delegation).

Slots of Self objects can either be readable, or assignment slots. For example, `x`, `y`, `print`, and `move` of the example above are considered to be readable slots. `x:` and `y:` are assignment slots for assigning values to `x` and `y`. The ← represents the assignment primitive operation, which is invoked to modify the contents of corresponding data slots. Note that values `x` and `y` are accessed through messages only, as well as `print` and `move`, which are considered as methods.

Methods are Self objects that include code. Note that in other programming languages, such as Smalltalk and Python, methods are also first-class objects like in Self (cf. Section 6.8 and [ROSSUM90]).

The object in the `print` slot above includes code and thus serves as a method. However in Self, any object can be regarded as a method. A “data” object contains code that merely returns itself. See also [UNGAR87] and [SMITH95].

If accessing slots in a prototypical object, and the object itself does not define these slots, then the parent object is searched, and then slots in the parent’s parent, and so on.

Generally, it is much easier to use templates such as prototypes and modify them, instead of creating something new from a plan. Think of planning and building a house—doing everything yourself starting with a basic plan is a lot of work and needs a lot of understanding. Selecting an existing and appropriate plan, letting someone build a house according to that, and finally making own modifications is easier and faster.

PCoC provides proper base classes and configurations which should simplify component development. They can be considered as templates for concrete components.

6.10.3 Prioritized Multiple Inheritance

6.10.3.1 Overview

Prototypical programming languages do not include classes, but instead allow individual objects to delegate to (dynamically inherit from) other objects. [CHAMB91] describes the inheritance and encapsulation mechanisms designed and implemented in early versions of Self. The inheritance system of Self supports a unique sender-path tie-breaker rule that resolves many ambiguities between unrelated unordered (not ranked) parents in multiple inheritance relationships, and dynamic inheritance which allows an object to change its parents at runtime to effect significant behavioral changes due to changes in its state.

Earlier versions of Self (before version 3.0) supported prioritized multiple inheritance. Randall B. Smith explains in [SMITH95] on Page 6 that it was a mistake introducing prioritized multiple inheritance into Self:

> “It is always tempting to add a new feature that handles some example better. Although the feature had made it possible to directly handle some examples, the burden it imposed in all reasoning about programs was just too much. We abandoned it for Self 3.0. Although adding features seems good, every new concept burdens every programmer who comes into contact with the language”.

Note that Self still supports multiple inheritance, but parents are not prioritized any more.

The reason for introducing this feature in an early version is explained as follows:

> “We prioritized multiple parent slots in order to support a mix-in style of programming. The sender-path tie-breaker rule allows two disjoint objects to be used as parents, for example a rectangle parent and a tree node parent for a VLSI cell object. The method-holder-based privacy semantics allowed objects with the same parents to be part of the same encapsulation domain, thereby supporting binary operations in a way that Smalltalk cannot”.

The statement continues:

> “But each feature also caused us no end of confusion. The prioritization of multiple parents implied that Self’s resend (call-next-method) lookup had to be prepared to backup down parent links in order to follow lower-priority paths. The resultant semantics took five pages to write down, but we persevered. After a year’s experience with the features, we found that each of the members of the Self group had wasted no small amount of time chasing compiler bugs that were merely unforeseen consequences of these features. It became clear that the language had

> strayed from its original path.
>
> We now believe that when features, rules, or elaborations are motivated by particular examples, it is a good bet that their addition will be a mistake. The second author (David Ungar) once coined the term “architect’s trap” for something similar in the field of computer architecture; this phenomenon might be called “the language designer’s trap”.
>
> If examples cannot be trusted, what do we think should motivate the language designer? Consistency and malleability. When there is only one way of doing things, it is easier to modify and reuse code”.

We fully agree with this statement. The prioritized dynamic inheritance model of PCoC also causes some problems. It requires that developers learn this different model. On the other hand, the PCoC approach is simpler. See later.

The prioritized multiple inheritance concept of Self (before version 3.0) combines unordered and ordered multiple inheritance. That is, on one hand, the parents of an object are ranked by their priorities. Slots of higher priority parents take precedence over that of parents with a lower priority. On the other hand, parents at the same priority level are unordered with respect to each other, and accesses to any clashing slot definitions will generate an ambiguous message error.

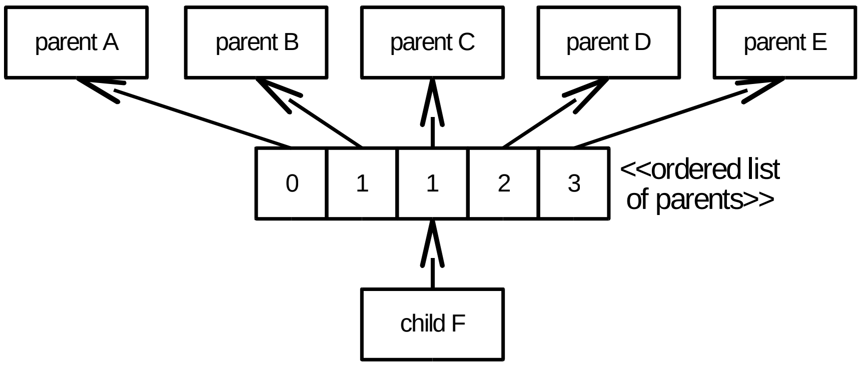


Figure 6.10-2 Prioritized multiple inheritance in Self

In this case, the parent object `A` has the highest priority (`0`), the highest ranking. `B` and `C` have the same priority (`1`). A so-called *sender-path tiebreaker rule* resolves ambiguities when the same slots are inherited from different parents with the same priority. Parents can direct subsequent messages (resends) back to the original receiver, or via other inheritance paths of the original receiver (in our case `child`), or to one of their own parents. This makes the concept even more complex. Read a more detailed description of this concept in [CHAMB91].

Some older programming languages such as New Flavors, CommonLoops, and CLOS, support ordered multiple inheritance. These languages linearize the inheritance graph, in addition to the ordering of parents. That is, they construct a total ordering of all classes that is consistent with each class’ local ordering, defined as the class followed by its direct parents in order. [CHAMB91] mentions two drawbacks of this linearization: ambiguities between otherwise unordered parents are ignored, and the concept fails if the local ordering of a class' parents is inconsistent with the global class ordering.

PCoC neither provides a global ordering of objects or classes (a linearization) like the mentioned programming languages, nor a mix of ordered and unordered multiple inheritance like the prioritized multiple inheritance feature of former Self versions (before version 3.0). It only supports a ranking of acquired parents.

The mechanism can be used to invoke Activities synchronously or asynchronously. Activities can be reused for various purposes. For example, they can be associated with menu entries and toolbar buttons, accessed via scripting and remote interfaces, etc.

PCoC users have a few things to learn:

1. an object can acquire (dynamically inherit) a set of operations (so-called Activities) from another object—the so-called parent. Acquired parents are ranked (prioritized). The ranking can be changed at any time.
2. how to send a message in order to perform such an operation. A message is always sent via the highest priority path which provides the specified operation. Multicasts are also possible.
3. resends to the original receiver are supported. When an Activity is invoked, it can invoke other Activities using Dispatchers of the original receiver of the current Activity request. The receiver (the Activity Set where a Dispatcher has been invoked) is passed as explicit parameter to the requested Activity. However, requests to other Activity Sets are also supported.
4. how to register and expose Activities (`addActivity`) and Activity Sets (specified name in the constructor of an Activities Provider). See Chapter 4.

The prioritized-acquisition mechanism of PCoC is mainly used for focus management of PCoC Tools and Service Providers, but is also useful for simple task scheduling, or simply for hiding the fact that some components may define the same operations. Many components provide semantically and syntactically equal operations such as `getSelection`, `copy`, `checkout`, and others. A `getSelection` message is automatically sent to the component with the highest priority. This is convenient for developers, since they need not bother about focus management, and the code stays the same, no matter if operations are invoked via script, GUI, or even if there is no GUI at all.

Unlike in Self, there cannot be multiple parents with the same priority. This makes the mechanism easier to understand.

PCoC has some introspection built in to help in cases of confusion. Of course, prioritized multiple inheritance and dynamic method invocation can be a real burden. However, with an introspection facility that gives a lot of feedback, developers can get and retain an overview of which instances of Activity Sets, Dispatchers, Activities, etc. exist at runtime and how they are related to each other. This helps to understand the behavior of the system and supports us to find errors. We found that developers are able to use this programming model efficiently and without getting lost in searching for bugs. For operations that should not be exposed through a user interface, scripting or remote interfaces, using normal methods is recommended.

See Sections 5.4.3 and 5.6 for details about the acquisition mechanism of PCoC.

6.10.3.2 Design Principles

One way to achieve malleability and reusability is sharing. So, one important concept of Self is that parents of an object are treated as shared subparts of the object. Message lookup in parents is done by treating the parents as part of the receiver.

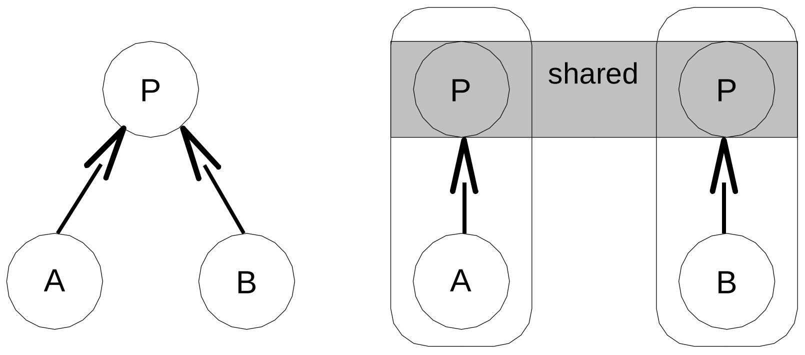


Figure 6.10-3 Parents as Shared Objects

The parent `P` of object `A` and `B` is treated as part of `A` and `B`. `self` always refers to the receiver of a message, regardless of inheritance. Inherited methods are considered to be shared parts of the same receiver object. For example, a method of `P` being invoked on `A` refers to the same receiver object `A`. `A` and `P` together form one receiver object, as well as `B` and `P`. If `A` cannot handle the message, it is searched for in parent `P`, and so on.

One problem of multiple inheritance is to resolve ambiguities between multiply-inherited conflicting behavior and states (methods and fields). Two or more parents may define the same slots. Schemes for automatically resolving such conflicts may be confusing to users.

Some older programming languages such as New Flavors, CommonLoops, and CLOS support ordered (ranked) multiple inheritance. This is useful if the object is more like one parent. This one parent would have a higher rank than the others.

The opposite approach is to leave the resolving of ambiguities to the programmer. Programming languages such as Common Objects, Trellis/Owl, Eiffel, C++ treat parents as equals without ordering. Ambiguities must be resolved explicitly be the developer. This unordered inheritance works best with relatively unrelated parents.

Both approaches have advantages and disadvantages. Earlier versions of Self combined these approaches by supporting prioritized inheritance. Parents with the same priority were unordered, and such with a higher priority had precedence over parents with lower priority. This means that slots of the parents with a higher priority had precedence over that of lower-priority parents. Slots of the object itself took precedence over inherited ones. If the same slot were inherited from two or more parents of the same priority, then the system generated a `messageAmbiguous` error.

In Self, parent slots are assignable like other slots, so it is possible to change the inheritance structure at runtime. This feature is called *dynamic inheritance.*

PCoC uses a similar approach. Parents can be assigned to components at runtime, and also removed. The corresponding operations are called “acquire” and “discard”. Actually PCoC supports more a kind of ordered inheritance rather than different priorities and unordered parents per priority level. The ranking can be changed at runtime. No clashes of slots, i.e., no ambiguous slots, can occur.

The PCoC inheritance mechanism can produce quite the same annoying effect as the prioritized multiple inheritance feature of earlier Self versions. The following drawbacks were reasons to abandon this feature in new versions of Self (borrowed from [CHAMB91], Pages 28-29):

> “One consequence of using higher-priority parents to implement mixins in Self is that these mixin objects should almost always be defined with no parents. Otherwise, the slots of the mixin’s ancestors, no matter how general, would override any slots in lower-priority parents, no matter how specific. This is almost never what the programmer intends.

> Another potentially surprising effect of ordering an object’s parents is that a chain of resends may eventually “backtrack” and call a method defined in a lower-priority parent of a descendant of the sending method holder, if no more ancestors of the descendant’s higher priority parent contain matching slots. This is a desirable feature of ordered multiple inheritance and resends, since it allows mixins to invoke the methods that they override, which are defined in lower-priority “cousins.” However, backtracking to lower-priority branches may surprise the novice programmer in other situations, especially if the lower-priority branch is a child of the resending method.
>
> Ordered multiple inheritance, resends, and dynamic inheritance have complex interactions. Dynamic inheritance does not normally affect message lookup, since assignable parents cannot change while a message send is being handled. However, they may change between resends within a chain of resends. This would not be a problem in a system with single inheritance or unordered multiple inheritance, since the message lookup could always begin with the resending method holder’s parents and proceed upwards. But with the introduction of ordered multiple inheritance, a resend might have to backtrack to a lower-priority parent of a descendant of the resending method holder to find the next matching slot. If a parent between the receiver and the resending method holder were changed between resends, it would be difficult to determine what the next matching slot should be, especially if the resending method holder were no longer an ancestor of the receiver at all.”

Inheriting objects through different paths, and thus also through different priorities, can indeed be troublesome. But, PCoC does not offer a complex mechanism to continue the search for a method or better an Activity, when a matching Activity has already been found. Either an Activity, or a set of Activities in the case of multicasts, is found or not, but the search cannot be continued. This may be a limitation, but it is still possible to invoke Activities explicitly in a parent.

Resends to the original receiver (Activity Set) of an Activity request are possible, but the resends are treated the same way as any other Activity request. More precisely, a resend is the use of a Dispatcher in the receiver Activity Set which is passed as explicit parameter to the current Activity. Ambiguities between parents, or better their Dispatchers, does not have to be resolved. Because of the ranking of parents, there cannot be ambiguities. For a few cases, some Activities may be acquired through different acquisition branches (an indirect parent can be reached through different priorities), but we did not encounter problems because of this. Operations that are inherited from the same component through different paths are only taken from the higher priority occurrence.

In short, the complexity is quite low. Resends are treated like any other Activity requests, no ambiguities between parents have to be resolved, and there is no support for continuing the search for an Activity. There is only one dispatch mechanism, and a request can be sent by explicitly using a Dispatcher of an Activity Set (the receiver). Read more about the dispatch algorithm of PCoC in Section 5.6.2.

[CHAMB91] explains the prioritized multiple inheritance feature of earlier Self versions and its drawbacks in detail.

6.10.4 Remarks

Self is an object-oriented programming language based on prototypes. Objects are created by cloning objects instead of class instantiation. No class objects are needed. The state information of objects is accessed solely through messages to “self”, which gave the programming language its name. Methods are objects associated with code, as opposed to variables. For data there is the concept of assignment slots. Normally slots are readable, no matter if accessing data or methods.

The developers of Self abandoned prioritized multiple inheritance. They say it was a mistake to introduce it in an earlier version. Too many ways to do things made it hard to get used of this programming language. They wanted to keep it minimal, so they removed this feature in version 3.0. Note that Self still supports multiple inheritance, but without prioritization of the parents of objects.

Self’s problem with managing prioritized inherited operations with the same name from different parent objects is actually a feature in PCoC. The approach is not as powerful as that of Self, but therefore easier to understand. The idea of PCoC is to manage cases where an object acquires from others which may define the same operations, by ranking the acquired parents. Although PCoC introduces another programming model for invoking methods, it is kept minimal. Experience has shown that developers understand the basic concepts of PCoC quickly. Problems during debugging are reduced by supplying some introspection facilities for PCoC. To get detailed information about PCoC’s priority management, see Sections 5.4.3 and 5.6.

PCoC has also quite the same drawbacks as the prioritized multiple inheritance feature of earlier Self versions. Inheriting a component through two different branches with different priorities can be troublesome, although we have not encountered such troubles with PCoC yet (due to its dispatch algorithm; cf. Section 5.6.2). Changing parent ordering during the process of delegating messages (and invoking operations) can have strange effects. We try to reduce these effects by simply letting the framework determine the current receiver paths for a set of Dispatchers before actually invoking them. If the actual receivers are determined before Dispatchers are invoked, then priority changes during the execution do not have any impact on the selection of the Activities to be invoked. This behavior is described for macro-Tasks in Section 4.5.3. In Self, a priority change during the execution of a method with subsequent resends may lead to surprising results. Depending on the priority change, the method lookup for the resends may continue with different methods.

For more information about Self, see [UNGAR87], [CHAMB89], [CHAMB91], [AGESE95], and [SMITH95]. [LIEBER86] discusses delegation in prototypical systems and may be interesting in this context.

6.11 System Object Model (SOM, by IBM)

For the sake of completeness, IBM’s System Object Model should also be mentioned here. SOM is a technology that was designed to overcome several major issues of using object class libraries. So-called *system objects* can be distributed and subclassed in binary form. Subclassing is supported across different programming languages; users may use class libraries in their preferred programming language.

SOM provides support for the fragile base class problem (when a class and its subclasses can evolve independently) and subsequent updating. Changing interfaces in base classes or components affects existing client code. SOM supports upward binary compatibility, which ensures that client code will still run after updating components.

Another interesting feature of SOM is its naming service. Instances of naming services can be organized in hierarchies—a feature that is supported by almost any new programming language (through name spaces, packages) and also by PCoC (through Activity Sets).

Resolution of multiple inheritance is done as follows: when the same method is inherited from multiple ancestors, the leftmost ancestor specified in the base-class list is used. This mechanism is called left path precedence rule.

SOM fully supports meta-programming. A program based on SOM can define new interfaces, synthesize new classes matching them, and create new instances of the new classes. SOM even allows intercepting the execution of operations. Distributed SOM makes use of this and integrates distribution support on top of SOM.

Read more about SOM in [IBMSOM94]. You can register at *http://www-3.ibm.com/software/ad/som/som30tk.html* for more SOM documentation and reference guides.

6.12 Design Patterns

While developing PCoC the question arose whether or not Activities and Dispatchers are similar to known patterns of software engineering literature. This section answers this question by comparing PCoC patterns to existing ones. Detailed descriptions of common design patterns mentioned here can be found in [GAMMA95].

6.12.1 Command Pattern

A *command* is an operation encapsulated as an object. Although an Activity also encapsulates an operation as an object, there are not many other similarities between these concepts. Command objects are more basic. They usually allow `do`, `undo`, and `redo`-operations. In order to execute a command, a command object must be created. For example, on the selection of a menu entry (e.g., “undo”), such a command object may be created. In contrast, Activities are usually created only once per component. They must already be present in order to perform them. An Activity may create a command object on its execution. However, itself is no command object.

6.12.2 Strategy Pattern

The *strategy pattern* provides a concept to define sets of algorithms, to encapsulate them in classes, and to vary algorithms independently of clients that use it. For example, we may define various classes with the same interface, each implementing a different graph layout algorithm. Such a class is called *strategy*. In PCoC, the strategy pattern is used for Dispatchers to gather acquired Activities (Activities from acquired Activity Sets) using the specified directives, and subsequently to invoke different methods of the resulting Activities, for example `getState`, `perform`, `getContextData`. The search algorithm is implemented just once. See Sections 4.4 and 5.6.2. This is useful, since such algorithms are often complex, and it is reasonable to maintain it only once in the source code. The strategy objects do the actual work on the objects delivered by the algorithms.

Although, Dispatchers make use of the strategy pattern, neither Dispatchers, nor Activities are strategy objects.

6.12.3 Activities: First Class Objects

Some of the patterns used in PCoC are conventional ones, such as bridge, singleton, observer, strategy pattern or combinations of these. Some patterns may be, in part, similar to existing patterns.

Basically, Activities are method objects. On framework-level, they are treated as first-class objects. They can be created, associated with a concrete method, added, and removed at any time.

Activities are also similar to the command pattern, or Java Swing Actions (see Section *How to Use Actions* in [WALRA99]), but actually they are quite different (see the previous sections). Activities are similar to method descriptors of Java Beans, but additionally make it possible to add and remove them at runtime, to specify and change state, for example, “Enabled” or “Disabled”, and provide additional context data, for example, data additional to return value of an Activity delivering the current selection in a text editor Tool.

See [ENGLA97] on Pages 205ff, for an explanation of method descriptors. See also the Sections *Using Java Reflection* and *Finding the method*, and Pages 41f for the basics of the Java reflection.

6.12.4 Dispatchers

There is no pattern for gathering possible receivers of requests ranked by priority (depending on the focus history of their components) and subsequently sending the requests. Dispatchers might be used without Activities, for example, with method references as in Java reflection, or as function pointers, etc. These actually do not require Activities.

However, Dispatchers can be described as smart method references holding direct or indirect references to Activities (which, in fact, are method objects); they provide forwarding and delegation capabilities and can be deployed independent of other Dispatchers; Dispatchers are bound to a specific Activity name and interface (similar to method signature).

7 Conclusion

This chapter summarizes features, advantages, and disadvantages of the framework presented in this thesis. It describes the impact of PCoC on real applications, and impressions and feedback provided by current users of PCoC.

7.1 Overview

Through its support for dynamic component modification and delegation, PCoC can be used to introduce some flexibility into applications. Component functionality, as well as its look and feel, can be customized at runtime, separating the implementation process from the customization. Operations, or more precisely Activities, can be reused for various application interfaces such as the user interface, scripting, remote control, etc., thus they need only be implemented once. The framework automatically exposes Activities, or better their Dispatchers, for their use with these interfaces.

The use of thin base classes provided by PCoC as templates for components can help to further reduce the implementation effort. The method dispatch mechanism including its support for different component interfaces is hidden in the base classes, as well as the processing of component startup, initialization, and termination. This approach allows developers to concentrate on value-added tasks such as implementing the core logic (the purpose) of each component.

7.2 Which Mechanisms When?

This section is the result of many team discussions of framework architects at Takefive Software, respectively WindRiver Systems, including the author of this thesis.

7.2.1 Static Interfaces / Methods

Static interfaces, or more precisely ordinary methods or functions, should be used in cases where tight binding is necessary or desired, for example, for high performance, more convenience, syntax checking at compile time, simple static source code browsing, etc.

Advantages:

- Compile-time type checking for method arguments and return values helps to find errors earlier than with dynamic method calls or other reflection-based dispatch mechanisms, where types can only be checked at runtime.
- Static method invocation keeps client code small and readable, as opposed to reflection-based mechanisms where arguments and return values mostly must be packed into special container objects, and type-casts are necessary (e.g., in Java, from `Object` to concrete types), and where the method call itself is more complicated.
- As opposed to reflection-based mechanisms, static method invocation is a widely understood and preferred model for synchronous calls.

Disadvantages:

- Methods should only be used for non-blocking and short-term operations. For long-term operations which block the current thread, asynchronous concepts such as events should be used.
- Methods are not suitable for asynchronous calls. For deferred execution, we need an object that represents an operation including its arguments. For such purposes, command objects or events should be used. PCoC supports Tasks for this purpose.

- Methods are not suitable for connecting user interface (menu, toolbar) code to components. For example, there is no support for state-change notifications (“Enabled”, “Disabled”, etc.), customization, visual representation, etc. For this purpose, for example, command objects, Java Swing actions, or PCoC Tasks, are better suited.

7.2.2 Events

Events are the most general form of component communication and can also be used to model dynamic dispatch.

Advantages:

- The model and usage is commonly accepted and understood.
- Events can be used for one-to-one, one-to-many, and two-way communication.
- An event system can be structured to prevent deadlocks and race conditions. Events in a queue can be rearranged in order to prevent deadlocks depending on the resources they require, whereas synchronous method calls cannot be rearranged in a call stack.

Disadvantages:

- Events are less comfortable to use than methods. For example, arguments must be packed into the event object, and thus cannot be passed as easy as with methods. Debugging is more difficult, because we lose information about when an event has been sent (there is no useful backtrace).
- Events are objects which must be assembled and put into a queue. This causes a performance penalty compared to static interfaces.

7.2.3 PCoC / Dynamic Dispatch

Activities and Dispatchers provide something that is missing or at least less satisfying in Java and other programming languages: flexible method dispatch, including delegation and the possibility to add operations to components or objects at runtime, or to modify them.

Java provides a similar concept with Swing Actions, but they cannot take arguments, do not have return values, and there is also no delegation mechanism.

.NET delegates are similar to Dispatchers, but they are less dynamic. They cannot be declared and bound to methods at runtime, but they are nevertheless very useful for various purposes such as multicasts, as strategy objects in algorithms, etc.

.NET provides with the name space `System.Reflection.Emit` a facility that allows to define and create assemblies, modules, types and methods at runtime. With PCoC, Activities and Activity Sets can be defined and created at runtime. For the invocation of Activities, Dispatchers are used.

As opposed to the Dispatcher dictionaries of PCoC, v-tables are static and cannot be modified at runtime.

As opposed to PCoC Activities and Dispatchers, the Smalltalk method dispatch does not offer support for method states, attaching of listeners to method references, and multicasts.

Advantages of the flexible method dispatch mechanism, respectively PCoC, are:

- PCoC supports single- and multicasts through Dispatchers. Specified directives (e.g., `first` or `broadcast`) determine how requests are delegated.
- Activities can be defined, added, and removed at runtime. This introduces some flexibility into a system.

- PCoC Tasks can be used for synchronous and asynchronous calls. For asynchronous calls, Tasks are put into a queue (`task.performLater()`).
- Tasks, Dispatchers, and Activities provide support for states and state notifications ("Enabled", "Disabled", etc.) This feature is needed for menu entries, toolbar buttons, scripting (for checking the availability and state of an operation), etc.
- Activity Sets can be organized in hierarchies in order to group semantically related sets, and therefore providing dynamic scopes.
- Activity Sets can acquire others, respectively their Dispatchers. This concept is useful to share behavior between components.
- License management is handled generically. It is only necessary to specify a license string when creating an Activity or Activities Provider. The license management is done by the framework (using a license management library such as FlexLM).
- PCoC supports reusability of code. Activities are generically exposed through various application interfaces, including user interfaces (for menu and toolbar entries), scripting, remote control interfaces (e.g. XMLRPC), and for the use within the source code of the components providing them.

Disadvantages:

- The PCoC approach leads to an increased memory footprint compared to method pointers.
- Type-checks are only possible at runtime, since Activities are invoked dynamically.
- It is difficult to get an overview of the relationships between Activity Sets, Activities, Dispatchers, and Tasks through source-code browsing. Although there are sophisticated introspection facilities available to inspect the relationships at runtime, and detailed error logs for type- and dispatch-errors, etc., the search for errors is not as efficient as with more static approaches like classes and methods. Some annoying drawbacks of a prioritized multiple inheritance mechanism such as that of PCoC and in the programming language Self are discussed in Section 6.10.3.
- The PCoC approach is initially unfamiliar to users. Using Activities instead of only methods and sending requests through Dispatchers is less comfortable than ordinary method calls. For example, arguments must be packed into argument containers (instances of Material), and there are different directives for single- and multicasts, etc.
- PCoC leads to a performance penalty compared to static interfaces (methods, functions), since Activities, Dispatchers, Activity Sets, etc., must be created and managed. However, this approach has not been designed for performance relevant purposes, but rather for medium- to long-term operations.

7.3 Remarks

Each of the mentioned dispatch mechanisms solves a specific, somewhat orthogonal problem quite well. Even though Activities and Dispatchers can be considered as a specialization of methods, initial user acceptance is hesitant. Nevertheless, experience has shown that developers begin to like the concept and understand its strength after having tried it for a few components.

PCoC is not a replacement for interface standards such as DDE, COM, CORBA, .NET, XMLRPC, etc. It just provides features missing in most programming languages, interface standards, and frameworks: dynamic scopes, delegation, dynamic component modification, and priority management for components and operations.

Some developers require PCoC to support static binding, for example, in order to invoke Activities like normal methods rather than using dynamic invocation with Dispatchers; they require Activities that are, like methods, statically bound to classes, and maybe not even separate objects; argument and result types should be checked at compile-time, etc. This is not what PCoC has been designed for. In such cases, static interfaces (functions, methods) should be used. However, if an application must support control via scripting, or there is a need for a general concept for events and dynamic method invocation, PCoC can save development effort. It can be used for operations that are generally user or script driven. Although the performance is quite good, it is not good enough to use Dispatchers, for instance, in performance-critical program loops.

Disadvantages and limitations of PCoC are the increased memory consumption and lower performance compared to language supported facilities, since each Activity, Dispatcher, Activity Set, etc., is a separate object.

When using PCoC, we have to worry about the number of Activities to be created, in order to keep the memory consumption low, and for security reasons. It would not be acceptable to expose each method as an Activity. Whenever ordinary methods can be used, they should be used.

Note that some memory and performance is consumed through reflection mechanisms already supported by platforms such as, for example, .NET and Java. Besides that, current applications generally consume a lot of memory, so that the portion of PCoC is rather irrelevant. For example, a concrete application has an average consumption of about 2.1 MB for the framework (100 loaded components and Activity Sets × approx. 200 bytes, 5000 Activities × approx. 160 bytes, 10000 Dispatchers × approx. 40 bytes, 2000 Tasks × approx. 240 bytes), whereas the whole application needs over 200 MB. We cannot give more precise numbers, since the memory consumption depends on many factors. Beside the number of objects, it depends on the number of acquisition relationships between Activity Sets, the complexity of Tasks, the number of different Activity interfaces (equal interfaces are shared), the string lengths of names, paths, etc.

In any case, development and maintenance costs are much more crucial for software development than memory consumption (the latter is also very important to be considered, though).

7.4 Discussion

The following questions arose during the development of PCoC.

Why do we choose such a design?

It has proved to support rapid application development and needs low effort to train developers. It does not claim to be suitable for every kind of component, but for a large area its use makes sense. The modular architecture and the dynamic binding between components enables modifying existing components without the need to recompile components that use them. Components can be loaded on demand, or unloaded, or replaced by other implementations dynamically.

Is it not better to provide tool developers with a maximum of freedom instead of providing component templates?

First of all, for intra-component communication it makes no or not much sense to use PCoC. Within a component, developers can use programming languages, language constructs, patterns, etc., as they like and as it makes sense.

However, for operations that must be accessible via several application interfaces, or used for inter-component communication, it makes sense to use PCoC concepts. This includes interfaces such as menus, toolbars, scripting, remote invocation, and other interfaces. PCoC guides us through component development by providing component templates. We can concentrate on value-added tasks such as implementing the core functionality (the concrete purpose) of a component, rather than having to care about component interoperation, different component interfaces, existence of other components, startup and termination issues, etc.

The Eclipse platform of OTI, an IBM research institution, is another good example of how a framework can provide some simple rules to create larger applications. Their plugin architecture is in some ways similar to that of PCoC. It has built-in support for component startup, termination, configuration, interoperation with other components, etc. See [OTI1001] and [OTI1101].

But what is it good for? Where is the gain?

PCoC offers a uniform message dispatch mechanism that can be used for various purposes. This includes synchronous and asynchronous (deferred) invocation of operations, delegation, license-management for operations (Activities can be associated with license strings), focus management, state support for operations, including change notifications (“Enable”, “Disable”, etc.), wrapping operations in order to add aspects, and dynamic component modification. The support for various component standards (COM, CORBA, etc.) is encapsulated in the PCoC dispatch mechanism. This makes client code smaller, more readable, and easier to maintain. Because of the uniform way to solve different problems, this approach also reduces training time and costs for new developers.

How long does it take to educate a new developer to make him understand an architecture and able to implement a new component?

With PCoC, we saw that developers can learn the necessary things to start implementing their first component within a few hours. They can immediately concentrate on implementing the actual functionality of their components, instead of thinking about the architecture and component interfaces of an application, and of re-inventing solutions for various issues.

7.5 Future Work

We are currently introducing a threading model to PCoC; this should help component developers to reduce the effort of managing common threading issues such as deadlocks by automatically rearranging Tasks in a queue, depending on the required resources.

Beside possible deadlocks, there are other issues such as blocking the AWT event dispatch thread in Java programs, and the missing read/write capabilities of semaphores. Java semaphores do not support simultaneous read-only locks, while blocking write locks, and vice versa. For more information about current threading issues in Java, see [HOLUB02] on Pages 275ff.

Work is in progress to forward SOAP calls to invocations of PCoC Tasks, which would standardize the interprocess communication with other applications. Since the structures of these two facilities are similar, they can be easily mapped.

Another idea is to use the framework for task scheduling, but we have not started on this, yet.

PCoC could also be used for managing administrative processes in firms, and especially in factories. For example, in a firm a manager requires tasks to be executed. He or she delegates tasks to workers on different tools and devices, or delegate them to services. Each task can again consist of subtasks. Materials are needed in order to execute tasks, and tools and workers must be available. If a tool or worker is no further available, or the used material is not appropriate, the manager will be notified. PCoC provides facilities for this kind of administration. The concepts and processes are quite similar. There are managers called Dispatchers, responsible for delegating the execution of specific activities. A task is a combination of activities performed on different devices or tools and/or by different services.

Bibliography

[ACQU00] Simpson E., *Untangling Acquisition with Trees*, http://www.zope.org/Members/4am/aq_trees, 2000

[ACQU01] Udell J., *Tangled in the Threads: Nature vs. Nurture*, BYTE Magazine, http://www.byte.com/documents/s=705/BYT20010614S0001/, 6, 2001

[ACQU98] Fulton J., Zope Corporation, *Acquisition*, http://www.zope.org/Members/michel/Projects/Interfaces/ReleaseDocumentation, http://www.ccs.neu.edu/home/lorenz/research/acquisition/, http://www.zope.org/Members/jim/Info/IPCB/AcquisitionAlgebra/siframes.htm, 1998

[AGESE95] Agesen O., Hölzle U., *Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages*, OOPSLA '95 Conference Proceedings, http://www.cs.ucsb.edu/oocsb/papers/oopsla95-tf.pdf under http://www.cs.ucsb.edu/oocsb/papers/tf-vs-ti.html, ACM SIG-PLAN, 1995

[BLOSS00] Blosser J., *Explore the Dynamic Proxy API*, Java World, http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-proxy_p.html, 11, 2000

[BREYM02] Breymann U., Loviscach J., *Die neue C-Klasse: C# im Vergleich mit C++ und Java*, c't: Magazin für Computertechnik, http://www.heise.de/ct, Heise, 2, 2002

[CHAMB89] Chambers C., Ungar D., Lee E., *An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes*, OOPSLA '89 Conference Proceedings, http://research.sun.com/self/papers/oopsla89.ps.Z under http://research.sun.com/self/papers/implementation.html, ACM SIG-PLAN, 1989

[CHAMB91] Chambers C., Ungar D., Hölzle U., Chang B.-W., *Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF*, LISP and Symbolic Computation, http://research.sun.com/self/papers/papers.html, Computer Systems Laboratory, Stanford University, 1991

[COPLI91] Coplien J. O., *Advanced C++: Programming Styles and Idioms*, ISBN0-201-54855-0, Addison-Wesley, 1991

[CZARN01] Czarnecki K.,Dominick L., Eisenecker U. W., *Aspektorientierte Programmierung in C++: Teil 1, Erste Aussichten*, 2001

[DAVID00] Davidson M., *Using the Swing Action Architecture*, http://java.sun.com/products/jfc/tsc/articles/actions/index.html, 2000

[ENGLA97] Englander R., *Developing JAVA Beans*, ISBN1-565-92289-1, O'Reilly, 1997

[FOOTE89] Foote B., Johnson R. E., *Reflective Facilities in Smalltalk-80*, OOPSLA '89 Conference Proceedings, ftp://www.laputan.org/pub/foote/Reflection.rtf, ACM SIG-PLAN, 1989

[GAMMA95] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN0-201-63361-2, Addison-Wesley, 1995

[GIL96] Gil J., Lorenz D., *Environmental Acquisition: A New Inheritance-Like Abstraction Mechanism*, OOPSLA 1996 Conference Proceedings, http://www.bell-labs.com/people/cope/oopsla/Oopsla96TechnicalProgramAbstracts.html#GilLorenz, http://www.cs.technion.ac.il/Labs/Lpcr/publications/lpcr9507.html, http://www.ccs.neu.edu/home/lorenz/research/acquisition/, 1996

[GITTI00] Gittinger C., *Die Unified Smalltalk/Java Virtual Machine in Smalltalk/X*, OOP 2000 Conference Proceedings, 101communications LLC, 2000

[GOLDB83] Goldberg A., Robson D., *Smalltalk-80: The Language and its Implementation*, ISBN0-201-11371-6, Addison-Wesley, 1983

[GRIFF98] Griffel F., *Componentware: Konzepte und Techniken eines Softwareparadigmas*, ISBN3-932-58802-9, dpunkt, 1998

[HARRIS97] Harrison W., Ossher H., Tarr P., *Using Delegation for Object and Subject Composition*, Research Report RC 20946 (922722), http://www.research.ibm.com/sop/abstracts/delegation.htm, 1997

[HOLUB02] Holub A., *Taming Java Threads*, ISBN1-893-11510-0, Apress, 2000

[IBMSOM94] Object Technology Products Group, *The System Object Model (SOM) and the Component Object Model (COM)*, http://www-4.ibm.com/software/ad/som/library/somvscom.html, 1994

[IBMST95] Khor K. K., Chavis N. L., Lovett S. M., White D. C., *IBM Smalltalk Tutorial*, http://media.dml.cs.ucf.edu/COP4331/Tutorials/Smalltalk/, 1995

[ISOCPP98] American National Standards Institute, *Programming Languages--C++, ISO/IEC 14882*, 1998

[iX1201] Stal M., *C#- und .NET-Tutorial: Teil 1*, iX: Magazin für professionelle Informationstechnik, http://www.ix.de/ix/artikel/2001/12/122/, Heise, 12, 2001

[JDK12] Sun Microsystems, *Java 2 SDK, Standard Edition Documentation, Version 1.2.2*, http://java.sun.com/products/jdk/1.2/docs/api/index.html, 1999

[JDK13D] Sun Microsystems, *Java 2 SDK, Standard Edition Documentation: Dynamic Proxy Classes*, http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html, 2001

[JDK13P] Sun Microsystems, *Java 2 SDK, Standard Edition Documentation: Class Proxy*, http://java.sun.com/j2se/1.4/docs/api/java/lang/reflect/Proxy.html, 2001

[KICZAL00] Kiczales G., et al, *An Overview of AspectJ*, technical report, http://aspectj.org/documentation/overview/aspectj-overview.pdf, 2000

[KNIES98] Kniesel G., *Delegation for Java: API or Language Extension?*, Technical Report IAI-TR-98-4, ISSN 0944-8535, http://javalab.cs.uni-bonn.de/research/darwin/papers.html; http://javalab.cs.uni-bonn.de/data2/papers/darwin/patterns.IAI-TR-98-5.pdf, 1998

[KNIES99] Kniesel G., *Type-Safe Delegation for Run-Time Component Adaptation*, Proceedings of ECOOP99, http://javalab.cs.uni-bonn.de/research/darwin/papers.html, 1999

[KRASN84] Krasner G., *Smalltalk 80: Bits of History, Words of Advice*, ISBN0-201-11669-3, Addison-Wesley, 1984

[LIBERT01] Liberty J., *Programming C#*, ISBN0-596-00117-7, O'Reilly, 2001

[LIEBER86] Lieberman H., *Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems*, OOPSLA '86 Conference Proceedings, http://lcs.www.media.mit.edu/people/lieber/Lieberary/OOP/Delegation/Delegation.html, http://lcs.www.media.mit.edu/people/lieber/Lieberary/OOP/OOP.html, ACM SIG-PLAN, 1986

[LIU96] Liu Ch., *Smalltalk, Objects, and Design*, ISBN0-132-68335-0, Manning Publications, 1996

[LUTZ01] Lutz D., *Aspekt-Orientierte Programmierung*, OOP 2001 Conference Proceedings, 101communications LLC, 2001

[MAETZ97] Mätzel K. U., Schnorf P., *Dynamic Component Adaption*, Technical Report 97-6-1, http://www.goeast.ch/maetzel/Publicationlist.html, 1997

[MICROS01] Microsoft, *Reflection: Leveraging the Power of Metadata*, Microsoft .NET Developer Tools Readiness Kit, NETRK CD:/modules/Reflection.ppt, 2001

[MONO03] Ximian, Inc., *Mono*, http://go-mono.com/, 2003

[MÖSS02] Beer W., Birngruber D., Mössenböck H., Wöß A., *Die .NET- Technologie: Grundlagen und Anwendungsprogrammierung*, ISBN3-898-64174-0, dpunkt, 2002

[MÖSS91] Mössenböck H., Wirth N., *The Programming Language Oberon-2*, technical report, http://www.oberon.ethz.ch/books.html, 1991

[MÖSS95] Mössenböck H., *Objektorientierte Programmierung in Oberon-2*, ISBN3-540-60062-0, Springer, 1995

[MÖSS98] Mössenböck H., *Objektorientierte Programmierung in Oberon-2*, ISBN3-540-57789-0, Springer, 1998

[MSCSH01] Microsoft, *C#*, Microsoft .NET Developer Tools Readiness Kit, NETRK CD:/modules/CSHARP.DOC, 2001

[MSCTS01] Microsoft, *Common Type System*, Microsoft .NET Developer Tools Readiness Kit, NETRK CD:/modules/Common_Type_System.DOC, 2001

[MSDLG99] Microsoft Corporation, *The Truth about Delegates*, http://msdn.microsoft.com/visualj/technical/articles/delegates/truth.asp, 1999

[MSNET01] Microsoft, *.NET Framework Overview*, Microsoft .NET Developer Tools Readiness Kit, NETRK CD:/modules/NET_Framework_Overview.doc, 2001

[OTI1001] Arsenault S., *Contributing Actions to the Eclipse Platform*, Technical Documentation, http://www.eclipse.org/articles/index.html; http://www.eclipsecorner.org/articles/index.html, 2001

[OTI1101] Springgay D., *Creating an Eclipse View*, Technical Documentation, http://www.eclipse.org/articles/index.html; http://www.eclipsecorner.org/articles/index.html, 2001

[ROSSUM90] Rossum G. v., *What is Python?*, http://www.python.org/doc/essays/blurb.html, 1990

[SMITH95] Smith R. B., Ungar D., *Programming as an Experience: The Inspiration for SELF*, ECOOP '95 Conference Proceedings, http://research.sun.com/self/papers/programming-as-experience.ps.Z under http://research.sun.com/self/papers/programming-as-experience.html, ACM SIG-PLAN, 1995

[STAL01] Stal M., *Reich der Mitte: Die Komponententechnologien COM+, EJB und "CORBA Components"*, OOP 2001 Conference Proceedings, 101communications LLC, 2001

[STROU97] Stroustrup B., *The C++ Programming Language*, ISBN0-201-88954-4, Addison-Wesley, 1997

[SUNDLG99] The Java Language Team, JavaSoft, Sun Microsystems, *About Microsoft's Delegates*, http://www.javasoft.com/docs/white/delegates.html, 1999

[SZYPE98] Szyperski C., *Component Software: Beyond Object-Oriented Programming*, ISBN0-201-17888-5, Addison-Wesley, 1998

[TOEDT01] Tödter K., *Let's Swing: JFC für Fortgeschrittene*, OOP 2001 Conference Handout, http://www.toedter.com, 2001

[TROEL01] Troelsen A., *C# and the .NET Platform*, ISBN1-893-11559-3, Springer, 2001

[UNGAR87] Ungar D., and Smith R. B., *SELF: The Power of Simplicity*, OOPSLA '89 Conference Proceedings, ACM SIG-PLAN, 1987

[WALRA99] Walrath K., Campione M., *The JFC Swing Tutorial: A Guide to Constructing GUIs*, ISBN0-201-43321-4, Addison-Wesley, 1999

[WESTP01] Westphal R., *Einführung in die Microsoft .NET Plattform,* OOP 2001 Conference Proceedings, 101communications LLC, 2001

[WIRTH92] Wirth N., Gutknecht J., *Project Oberon: The Design of an Operating System and Compiler*, ISBN0-201-54428-8, Addison-Wesley, 1992

[XEROX02] AspectJ Team, *The AspectJ Programming Guide*, technical article, http://aspectj.org/doc/dist/progguide/, 2002

[XEROX0297] Mendhekar A., Kiczales G., Lamping J., *RG: A Case-Study for Aspect-Oriented Programming*, Technical report SPL97-009, P9710044, http://www.parc.xerox.com/csl/groups/sda/publications/papers/PARC-AOP-RG97/, Xerox Palo Alto Research Center, 1997

[XEROX0697] Kiczales G., Mendhekar A., Maeda C., et al, *Aspect-Oriented Programming*, http://www.parc.xerox.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/, Xerox Palo Alto Research Center, 1997

[XEROX1297] Kiczales G., Lopes C. V., *D: A Language Framework for Distributed Programming*, Technical report SPL97-010, P9710047, http://www.parc.xerox.com/csl/groups/sda/publications/papers/PARC-AOP-D97/, Xerox Palo Alto Research Center, 1997

[ZULLIG98] Züllighoven H. et al., *Das objektorientierte Konstruktionshandbuch*, ISBN3-932-58805-3, dpunkt, 1998

Index

Lebenslauf

9. Feb. 1972	Geboren in Braunau, Oberösterreich
1978 - 1982	Volksschule Altheim
1982 - 1986	Hauptschule Altheim
1986 - 1991	HTBLA Braunau / Zweig Informatik
Mai 1991	Matura mit gutem Erfolg
Juli 1991 - Feb. 1993	Militärmusik Linz / Ebelsberg
März 1992	Beitritt Shotokan Karate International / Österreich
Okt. 1992 – Mai 1997	Informatikstudium an der Johannes Kepler Universität Linz
	Während des Studiums tätig bei:
	Technodat GmbH, Salzburg (Programmierer)
	TakeFive Software GmbH, Salzburg (Programmierer)
Seit 1995	Karatetrainer
Mai 1997	Studienabschluss und Sponsion zum Dipl.-Ing.
Juli 1997	Anstellung als Softwarearchitekt bei WindRiver GmbH, Salzburg (vormals TakeFive Software GmbH)
Seit 1998	Obmann Shotokan Karate International / Landesverband Oberösterreich
Seit 1999	Mitglied des Nationalteams bei Shotokan Karate International / Österreich
Okt. 2000	Inskription zum Doktoratstudium der technischen Wissenschaften
Seit Nov. 2002	Anstellung als Softwareentwickler bei eurofunk Kappacher, St. Johann / Pg.