JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Enhanced Heap Visualization with GCspy

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Peter Hofer, 0855349

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
o.Univ.-Prof. Dr. Dr.h.c. Hanspeter Mössenböck

Mitwirkung:
Dipl.-Ing. Thomas Schatzl

Linz, August 2011

Abstract

GCspy is a tool intended for the developers of memory management systems. It can be easily integrated into any existing system and helps in comprehending and verifying the system's behavior by visualizing the heap layout at runtime. [Printezis02]

While the original software was developed in 2002, the primary goal of this thesis was to enhance the possibilities of visualization, to improve the overall usability and to remove some long-standing shortcomings of GCspy.

The thesis analyzes GCspy's architecture, describes the introduced improvements and illustrates the GCspy client from a user perspective.

Kurzfassung

GCspy ist ein Werkzeug für Entwickler von Speicherverwaltungssystemen. Es kann einfach in bestehende Systeme integriert werden und unterstützt beim Verständnis und beim Überprüfen der Arbeitsweise des Systems durch grafische Darstellung der Heapbelegung zur Laufzeit. [Printezis02]

Während die ursprüngliche Software bereits 2002 entwickelt wurde, war das primäre Ziel dieser Arbeit die Erweiterung der Möglichkeiten der Visualisierung, die Verbesserung der Benutzerfreundlichkeit als Ganzes und die Beseitigung einiger länger bestehender Mängel von GCspy.

Die Arbeit analysiert die Architektur von GCspy, beschreibt die eingeführten Neuerungen und illustriert die Bedienung des GCspy-Clients.

Contents

Chapter 1

Introduction

Memory management systems have to meet high expectations. They need to provide high allocation performance while at the same time keeping resource usage and fragmentation low. Garbage collectors are expected to quickly dispose of unused objects, but should impose minimal delays on the running application doing so. To meet these expectations, many of today's memory management systems have turned into highly sophisticated and complex pieces of software that incorporate a vast amount of research. As a consequence, it becomes increasingly hard to comprehend and verify how they operate.

GCspy is intended to provide the developers of those systems with a tool to monitor and review the behavior of their software beyond simple trace messages, assertions and memory dumps. It provides ready to use components for easy integration into any existing memory management system. Because of the abstraction of the heap using the concept of *spaces, streams* and *blocks,* developers can define their own representation of managed memory. The *GCspy client* visualizes the heap layout and allows to assess the current state of the heap at a glance.

Goals and tasks

This thesis focuses on the GCspy client. Although it is a central component of the GCspy framework, the GCspy client provides only elementary visualization capabilities. Most notably, it offers only a single mode of visualization that is not suitable for some frequently used types of data, particularly enumerations. It also lacks the functionality to compare related values, even though this qualifies as a frequent use case. The GCspy client displays various information only in text where a graphical representation would be much more effective. Furthermore, the user interface is more complex than necessary.

GCspy was first released in 2002. Since then, there has been no substantial development activity. As a result, the GCspy client still contains several long-standing defects. Many of these defects result in spurious crashes during normal use, which constitute a major usability issue. Another sign for GCspy's unmaintained state is that it depends on

outdated libraries such as the Java Advanced Imaging API, which was last updated in 2006 [JAI]. The user interface is also not on par with modern user interfaces and does not allow the user to resize or rearrange its views, which is inconvenient when working with large amounts of data.

Therefore, the primary objective of this thesis is to extend the visualization capabilities of the GCspy client, to improve its general usability and to resolve the described long-standing issues.

The goals and tasks of this thesis can be summarized as follows:

- Analyze the architecture and functionality of the GCSpy framework.
- Compare GCspy with other software for monitoring the heap state.
- Inspect the GCspy client code base to identify areas for improvement.
- Resolve identified issues and apply design changes to simplify future changes.
- Extend GCspy's visualization capabilities and modernize the user interface.

Outline

Chapter 2 of this thesis analyzes the architecture of the GCspy framework, the communication between its components and the abstraction of the heap layout. It also compares GCspy with similar tools capable of collecting and analyzing runtime heap information. Chapter 3 illustrates the functionality and use of the GCspy client from a user perspective. Chapter 4 describes the implementation of GCspy and the purpose and relationships of the most relevant classes. Chapter 5 discusses the enhancements introduced in the scope of this thesis and their implementation.

Chapter 2

The GCspy framework

2.1 Overview

GCspy is an architectural framework for collection, transmission, storage and visualization of heap information. It is not specific to any particular application or runtime environment and can easily be adapted for use with any memory management system. [Printezis02]

This chapter analyzes the architecture of GCspy and how the individual components communicate with it each other. It then goes on to illustrate how the layout of heaps is abstracted using general, non-specific data structures in GCspy. Finally, GCspy is compared to other tools for visualizing runtime heap information.

2.2 GCspy architecture

GCspy is designed and implemented as a client-server architecture.

The server component runs within the observed application or its virtual machine. The GCspy client connects to the server via a TCP/IP socket connection. Triggered by certain events such as a garbage collector run, the server sends a snapshot of the heap to the client. The client then interprets and visualizes the received data. It also maintains an index with a history of received snapshots to allow viewing the state of the heap at previous points in time.

The GCspy framework provides the server infrastructure as libraries for C, C++ and Java. The client (also called visualizer) is a Java application with a Swing user interface. Developers only need to implement data collection in their memory management system and provide a driver that interprets and preprocesses the collected data for use with GCspy. Due to the representation of the heap layout in abstract structures, the client does not require any modifications for supporting new memory management systems.

Figure 2.1 depicts the described architecture for a virtual machine with garbage collection. The left-hand side shows the virtual machine with an integrated GCspy server component. On the right-hand side is the stand-alone client which communicates with the server within the virtual machine. Only the *data collection* and *driver* components on the server side are specific to the memory management system in question and must be implemented by users of GCspy.

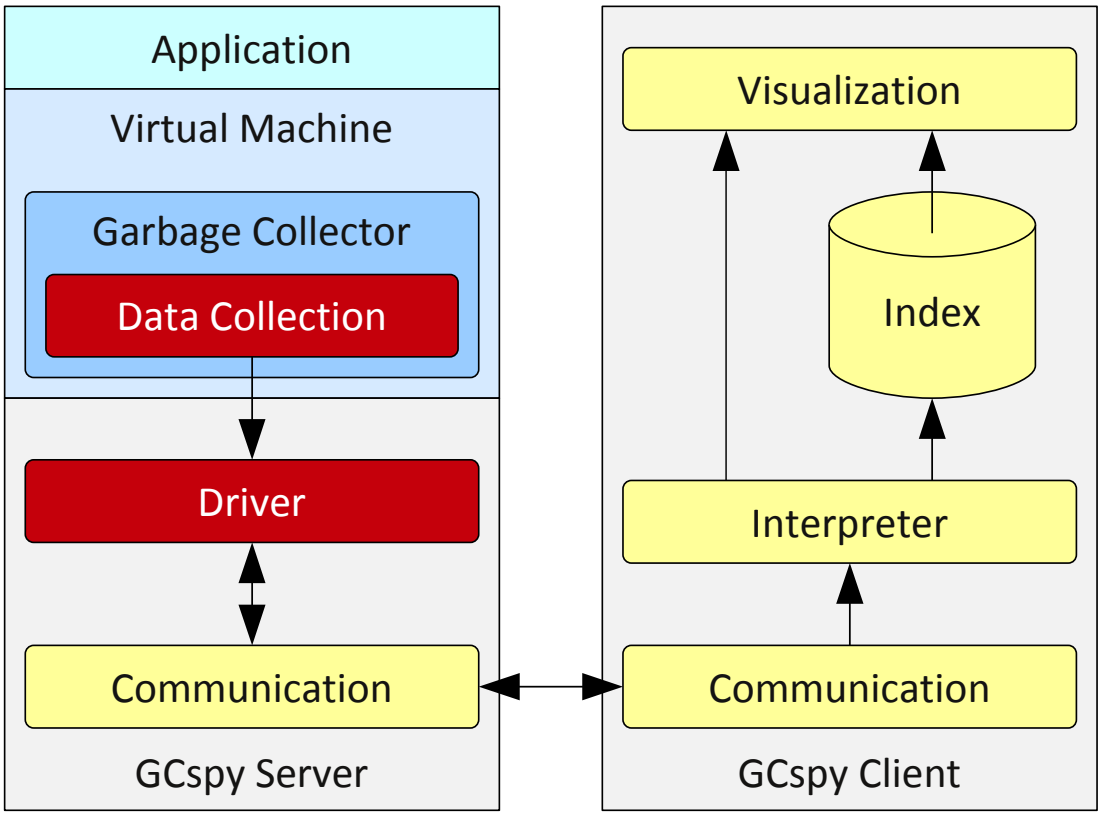

Figure 2.1: Architecture of GCspy with a garbage-collecting virtual machine

At runtime, the GCspy client establishes a TCP socket connection with the GCspy server. When a significant event occurs on the server, the *data collection* component collects and provides the relevant information about the heap. The *driver* then converts the collected data to a representation in GCspy's more general abstract data structures. This representation is then sent in the response to the client, which receives and interprets the data. The client visualizes the new heap snapshot and in addition, stores it in an index so it can be viewed again at a later point in time.

2.3 Abstraction of the heap

GCspy abstracts the specifics of the observed memory management system by introducing the concept of *events*, *spaces*, *blocks* and *streams*. This section describes these terms.

The Heap is the complete set of data that is observed by GCspy.

Events are significant points during the execution of the observed application at which the heap state may be collected and visualized on the client. An example for an event is the start or end of a garbage collection run.

Spaces are typically regions of managed memory, but can also represent lists or sets. A heap can have multiple spaces. For example, the young and old generations of a generational garbage collector would each qualify as separate spaces. Each space consists of a number of *blocks*.

Blocks represent the managed units of a space. They are displayed as equally sized, adjacent *tiles* in the GCspy client. However, their actual size, location and order in memory may differ. Other than chunks of a memory area, blocks can be used to represent entries of a free-list or remembered-set or other entities.

Streams are the attributes of the blocks of a space. GCspy has two types of streams:

Value streams have integer values that lie within a specified range. The client uses the value relative to the range boundaries to visualize these streams with vertical filled bars or fading colors. The preferred display color can be set on the server side. As an example, the number of objects per block would be specified in a value stream.

Enumeration streams specify a set of possible named values. The client represents the different values with distinguishable colors or patterns and displays the assigned descriptions. Enumeration streams are typically used to specify the type of a block or boolean values such as *marked* and *not marked*.

GCspy also has two visual elements to indicate relations between blocks:

Separators between two adjacent blocks denote boundaries between distinct areas.

Links between two adjacent blocks indicate a connection or common characteristics of these blocks.

It is the driver's task to interpret the data gathered by the data collection layer of the memory management system and provide the GCspy server infrastructure with a representation of this data as spaces, blocks and streams.

Figure 2.2 illustrates this abstraction with a Mark&Sweep garbage collector. The data collection functionality within the collector creates a "raw" representation of the heap by dividing it into equally large chunks of memory. The chunks have three attributes: the number of contained objects, how many of these objects have been marked and the occupancy in percent. The driver is provided with that data and creates a representation using GCspy's abstract structures. The heap is represented by a single space, chunks are turned into blocks and the three attributes become streams. This implementation-independent representation is then sent to the client for visualization.

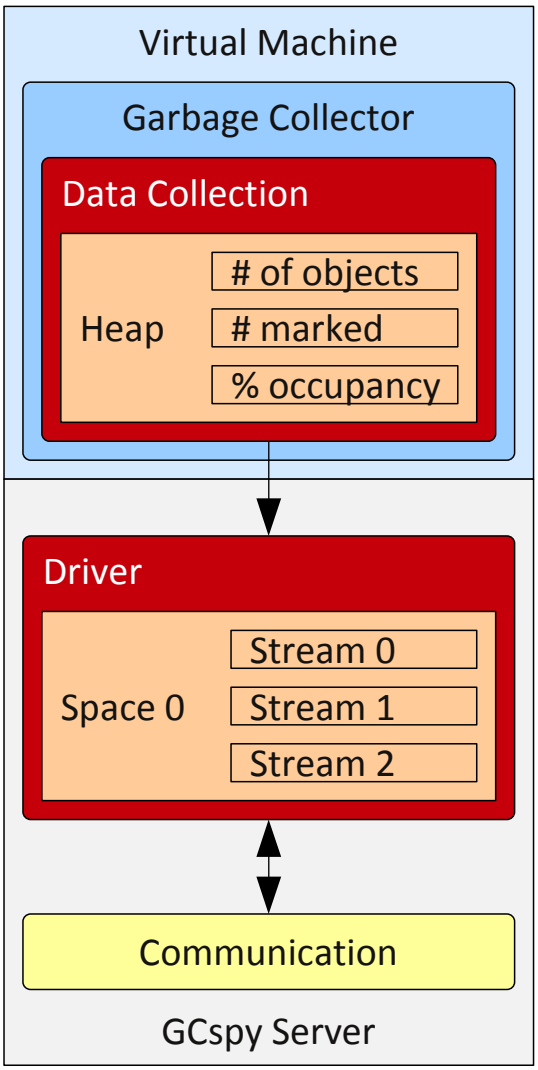

Figure 2.2: Abstraction of a Mark&Compact garbage collector in GCspy

2.4 Communication

The GCspy client and server communicate over a TCP/IP socket connection using a custom binary data protocol.

When establishing the connection, both client and server transmit a *magic string* and their *byte order* (endianness). When the magic strings differ or the byte order does not match, the connection fails and an error is raised on both sides. If successful, the server sends a string with its name. The client then sends a flag indicating if the server should pause execution as soon as possible until it receives a command to continue. The server proceeds to send the client all information on the number and layout of spaces and their streams and the memory management events that may occur.

2.4.1 Commands

After the connection has been fully established, client and server communicate via *commands*. Figure 2.3 shows the structure of such a command. Each transmitted command starts with a fixed 4-byte *start tag* for verification by the receiver, succeeded by a single byte with the *type* of the command. Then, *data* specific to this type of command follows. Finally, each command ends with a `0` byte and a 4-byte end tag.

···	start tag	command type	command-specific data	`0` byte	end tag	···

Figure 2.3: Composition of a GCspy command

GCspy distinguishes between server-to-client and client-to-server commands in communication. Figure 2.4 depicts the structure of a server-to-client *stream command*. The server sends these commands to the client to indicate that the values of a stream have changed.

Like all commands, a stream command begins with a start tag and the command type, which is `0x07` in this case. Next is one byte with the space identifier succeeded by another byte that identifies the particular stream of that space. After that, the command contains a 4-byte integer with the number of values in the stream, followed by the sequence of actual values. These values are encoded as single-byte, two-byte or four-byte integers depending on the type of the stream. The client knows the stream's type from earlier *space commands* from the server. The stream command ends with a zero byte and the end tag.

···	start tag	type `0x07`	space identifier	stream identifier	
	4-byte length	sequence of values	`0` byte	end tag	···

Figure 2.4: GCspy's stream command

Appendix A contains a complete list of commands of both types and their description.

2.5 Comparison to other tools

There are several other tools for monitoring heap state and usage of a process. However, most of these tools are designed to monitor the memory usage of the *application* instead of the behavior of the *memory management system*.

This section describes the differences between GCspy and two other tools, VisualVM and VisualGC. *VisualVM* is a free and powerful performance visualization tool originally

developed as part of the NetBeans IDE [VisualVM]. *VisualGC* is an unbundled Java SDK tool that graphically displays performance data of the garbage collector and the virtual machine [VisualGC].

2.5.1 VisualVM

VisualVM was originally developed as profiler for the NetBeans IDE, but has since been made available as a stand-alone application. It can be obtained for free from its project website on *java.net*. A stable distribution of VisualVM named *Java VisualVM* is included in Sun JDK releases starting from JDK 6 update 7 [VisualVM].

VisualVM is intended for application developers and system administrators to profile and monitor their applications both during development and in production use. It uses the Java Management Extensions [JMX] to obtain data from the observed application's virtual machine, which effectively limits VisualVM to monitoring Java applications. VisualVM is extensible with plugins that can be installed from a central repository on the Internet or a local file.

VisualVM only shows basic usage data for the whole heap and the generational garbage collector's permanent generation. It can be used to generate heap dumps and inspect the class instances in memory at the time of the dump, but does not provide any further information on how these instances are managed by the garbage collector. However, garbage collectors could make their performance data available via JMX and a plugin for VisualVM could be developed to visualize that data.

Figure 2.5 depicts VisualVM plotting heap usage over time in a virtual machine that is running a benchmark. The graphs in the center area of the figure show that VisualVM only distinguishes between used and free heap space and that it can additionally plot usage data for the permanent generation. The lower region of the screen capture contains VisualVM's plots of class loads and threads running in the virtual machine. Also, the tab panel in the upper area of the screen capture has a tab of the VisualGC plugin for VisualVM.

2.5.2 VisualGC

VisualGC is an unbundled tool for the Sun JDK available for download from the Sun Developer Network. It can also be installed as a plugin for VisualVM [VisualGC].

VisualGC was originally developed to show the effect of various tunable parameters of the Java virtual machine. Its audience are application developers and system administrators who wish to tweak the performance of their application and virtual machine instance and verify the results.

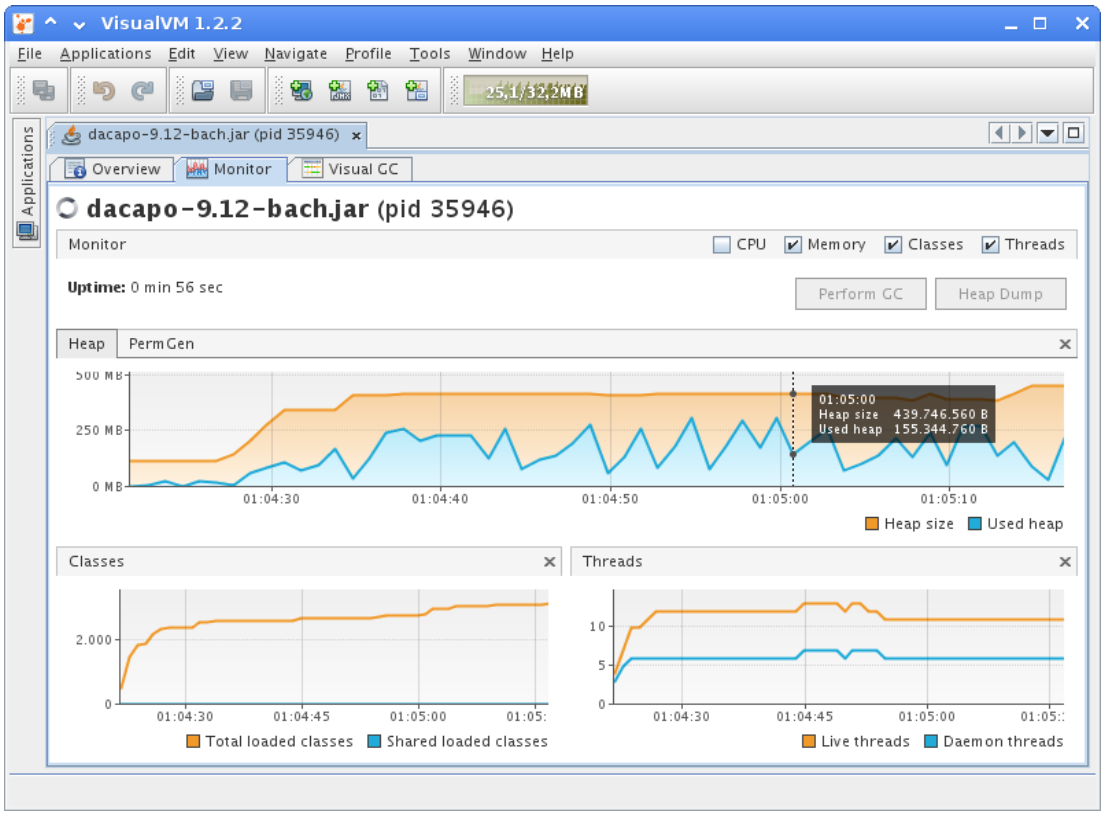

Figure 2.5: VisualVM showing heap utilization during a memory-intensive benchmark

For access to performance data, VisualGC uses the *jvmstat* lightweight performance instrumentation [jvmstat]. Only Sun's Hotspot virtual machine implements this instrumentation and its interfaces are considered private, uncommitted and are officially unsupported. Therefore, VisualGC is limited to Hotspot and not even guaranteed to work with future Sun JDK releases.

VisualGC plots the graphical "fill state" of the heap regions over time and in addition the time spent in garbage collection, compiling and loading classes. It does not show the location of objects in the heap or the steps performed in garbage collection.

Figure 2.6 shows VisualGC visualizing heap usage of a benchmark running in a Hotspot virtual machine. The graphs in the window on the left-hand side display the "fill state" of the particular heap regions (in this case *generations*). Coloring of the grid allows to distinguish between committed and uncommitted memory. The plots in the window on the right show memory usage of the generations over time and also time spent in garbage collection, compilation or loading classes.

2.5.3 GCspy

GCspy is designed specifically for the developers of memory management systems.

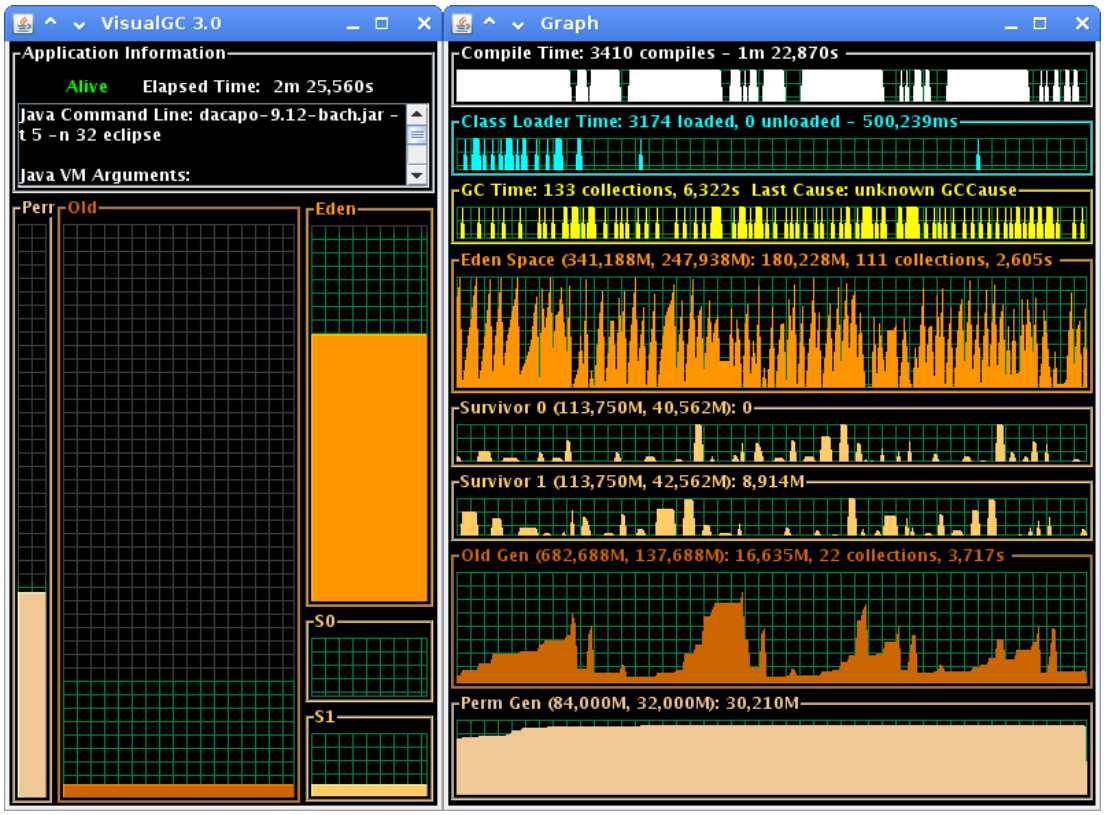

Figure 2.6: VisualGC showing heap utilization per space and the time spent in garbage collection during a memory-intensive benchmark

It provides a ready to use server component for C, C++ and Java to communicate with the client and only requires development of the data collection and driver modules specific to the memory management system that should be monitored. GCspy has its own client-server communication protocol and is independent of any custom or proprietary interfaces.

GCspy can visualize multiple spaces, which consist of an arbitrary number of blocks. The blocks of a space can have any number of different property values called streams. The user can choose the most fitting representation of the heap layout as spaces, blocks and streams. Therefore, an almost arbitrary level of detail is possible.

However, no main-stream virtual machine or memory management system comes with built-in support for GCspy. To use GCspy with one of these systems, the developer has to integrate it with GCspy first. Due to the generally higher level of detail when compared to other tools, integration of GCspy also demands a more sophisticated data collection layer and better understanding of the memory management's inner workings than other solutions.

Still, GCspy has been successfully integrated and used with many garbage collectors and virtual machines such as *Jikes RVM* [Printezis02] [Singh07], *Rotor* [Marion05], *dlmalloc*

[Cheadle06] or *JamVM* [Baldwin05]. It has also been used for visualizing specific approaches to memory management like the train garbage collector [Printezis02b].

Chapter 3 contains examples on using the GCspy client and several screen captures.

Chapter 3

GCspy client user manual

3.1 Overview

The GCspy client is the part of the GCspy framework responsible for the visualization of heap information. It is a stand-alone application that receives snapshots of the heap from a server or reads them from a trace file and creates a graphical representation of the heap layout. This graphical representation allows users to quickly analyze and assess the state of the heap.

The GCspy client is implemented in Java and uses the Swing user interface toolkit. It builds on the abstraction of the heap described in chapter 2 and therefore requires no changes to support new memory management systems.

This chapter describes the main use cases, functionality and usage of the GCspy client.

3.2 Main use cases and functionality

Simple integration: Developers can easily add GCspy support to their memory management system using the GCspy server framework provided for C, C++ and Java. Any changes are made exclusively on the server side, the client is unaffected because of the abstraction of the heap layout. Incorporation of a driver for GCspy is straightforward and possible in less than 300 lines of code. [Printezis02]

Coherent visualization: GCspy's visualization capabilities allow to quickly comprehend the state of the heap and spot deviations from expected values. Values from multiple streams can be visualized next to each other to allow a quick visual comparison of related values. Enumeration values are assigned easily distinguishable colors. The tile size can be changed for each space to get a better view of large spaces, while the intelligent magnification feature shows an enlarged view of the tiles near the mouse cursor. Additional settings exist for better results on different display devices such as projectors.

Timeline navigation: The GCspy client provides the ability to jump to specific events within the recorded timeline and examine the state of the heap at that time.

Keyboard shortcuts: Keyboard shortcuts and menu mnemonics in the GCspy client provide quick access to commonly used features.

Flexible user interface: All components of the user interface can be rearranged and resized. Particularly large space views can be detached from the main window and shown in their own window or moved to a separate screen.

3.3 Main window

The client's main window is the central work space for users of GCspy. It contains a view for each of the heap spaces where the blocks of the space are visualized as tiles. These tiles can be selected to show more detailed information about a block. Navigation controls allow to select earlier or later snapshots in the trace history.

Figure 3.1 depicts a typical layout of the GCspy main window while visualizing a trace.

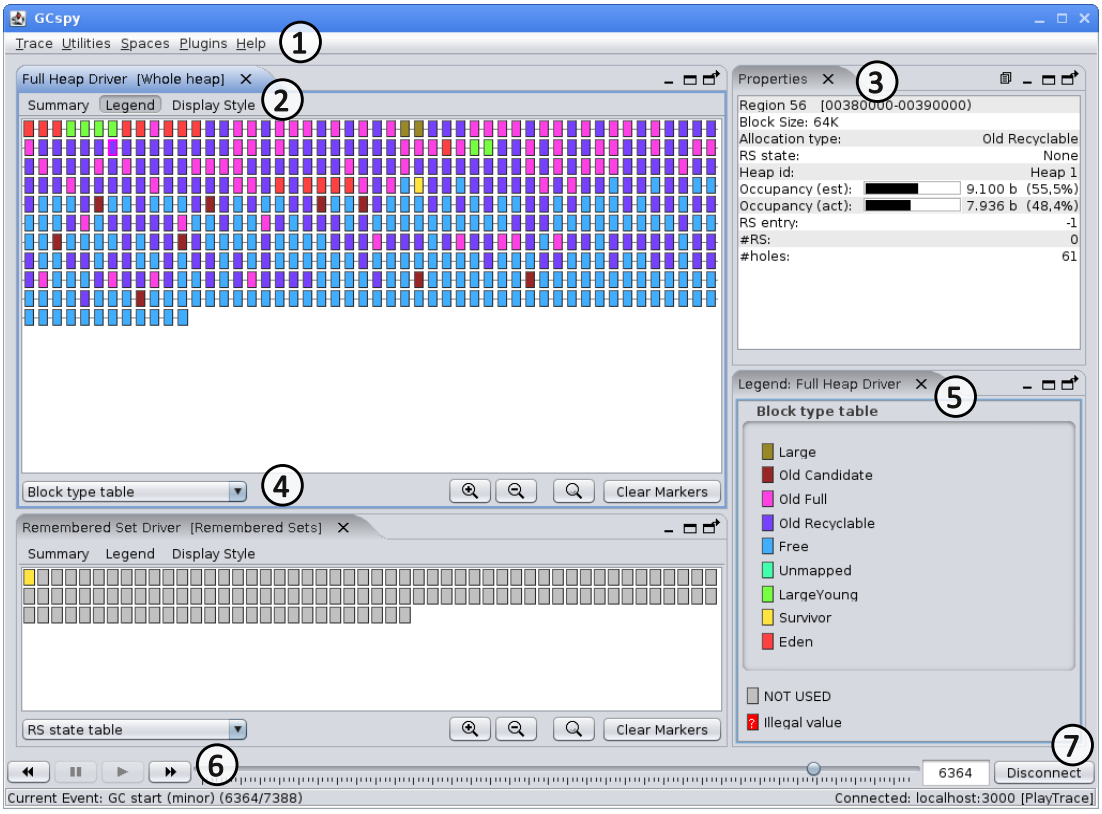

Figure 3.1: GCspy main window showing a trace

The main window consists of the following components, as marked in the figure:

① The menu bar at the top of the main window provides access to less commonly used functionality and plugins.

② The space views on the left visualize spaces as tiles based on its blocks and streams. The toolbar at the top of a space view allows to show views with a summary or legend for the space such as ⑤ or to change the display style of the tiles.

③ The property view on the right side of the window shows a description and the values of all streams for the currently selected tile.

④ The controls at the bottom of a space view allow selecting the displayed stream, changing the tile size, enabling magnification and clearing tile markings.

⑤ The legend depicts how different values of a stream are rendered in the tile view. This view is not shown by default, but can be activated from a space view's top toolbar.

⑥ The buttons, slider and input field at the bottom of the main window allow to navigate to other snapshots in the trace history or to pause and resume execution on the server.

⑦ The disconnect button closes the connection to the currently connected server. When the GCspy client is not connected, there is a connect button at this location to establish a server connection or replay a trace from a file.

The main window's layout is very flexible: except for the menu and navigation controls, all components are dockable. This means that users can change their position and size at will. Components can be rearranged, detached from the main window or grouped in tabbed containers. They can also be temporarily maximized to take up the entire main window area, minimized to save space, or hidden entirely. This is particularly helpful when dealing with large spaces.

3.4 Connecting to a GCspy server

When the GCspy client is launched, it automatically opens its *Connect* dialog. This dialog enables users to connect to a GCspy server or replay a trace stored in a file.

Figure 3.2 shows the connect dialog for both establishing a connection to a server and replaying a trace from a file.

For socket connections to a server, the user can specify the host to connect to and the port where the GCspy server is listening. In addition, execution on the server side can be suspended immediately after the connection has been established.

Alternatively, traces can be replayed from a file. In this case, a local *replay server* is run within the GCspy client. The replay server acts like an actual GCspy server and offers the same level of control such as pause and continue.

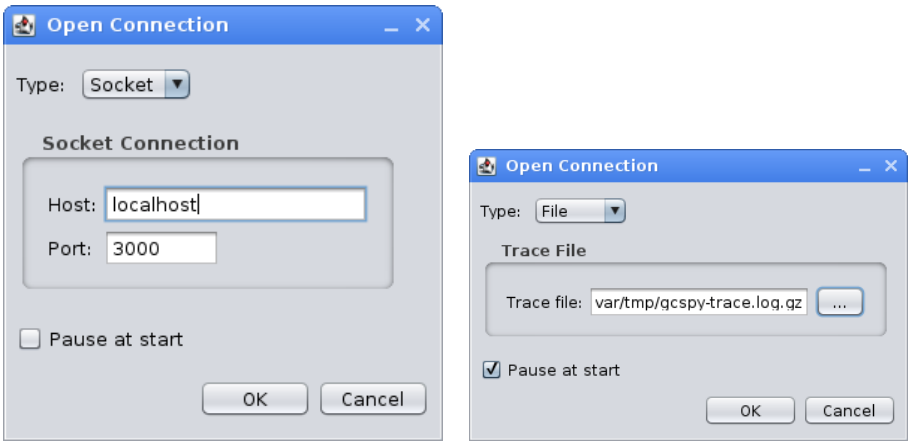

Figure 3.2: GCspy's connect dialog for remote connections as well as replaying files.

3.5 The space view

The space view visualizes the blocks and streams of a space as tiles. It consists of a toolbar at the top, a large area displaying the tiles of the space and another panel at the bottom with controls for selecting the stream to visualize or to control magnification and tile size.

Figure 3.3 depicts a space view visualizing a stream named *Block type table*. This stream has been declared in the server as enumeration stream, and the possible values (block types) and their display names are also provided by the server. The client assigns visually distinguishable colors to the enumerated values. Every tile is colored according to the value of the block it represents.

Figure 3.3: The space view showing a single stream.

The drop-down list at the bottom left allows selecting the stream used for visualization of the space. Additionally, a selection window can be opened that allows choosing multiple streams which will be visualized next to each other in each tile by dividing the tile vertically.

The magnifier buttons in the bottom panel allow to increase or decrease the tile size. This can be helpful when visualizing multiple streams or viewing very large spaces. Additionally, another feature called *intelligent magnification* can be enabled. When this feature is active, an area in one of the space view's corners shows a magnified view of the region near the mouse pointer. That area automatically moves to another corner when the mouse pointer is near.

Figure 3.4 shows a space view that visualizes two streams by dividing its tiles vertically. The *actual occupancy* stream specifies how much of a block's capacity is utilized by objects and is rendered in the right half of each tile. The *estimated occupancy* stream represents the memory management system's estimate of that value and is rendered in the left half. Rendering these two streams side-by-side for each block allows for a very quick assessment of how accurate the estimates are. The multi-stream visualization mode can be selected in the drop-down list at the bottom left ①. Stream values are rendered as vertical bars, so an empty cell means a value close to zero while an almost full bar represents a value close to the 100%.

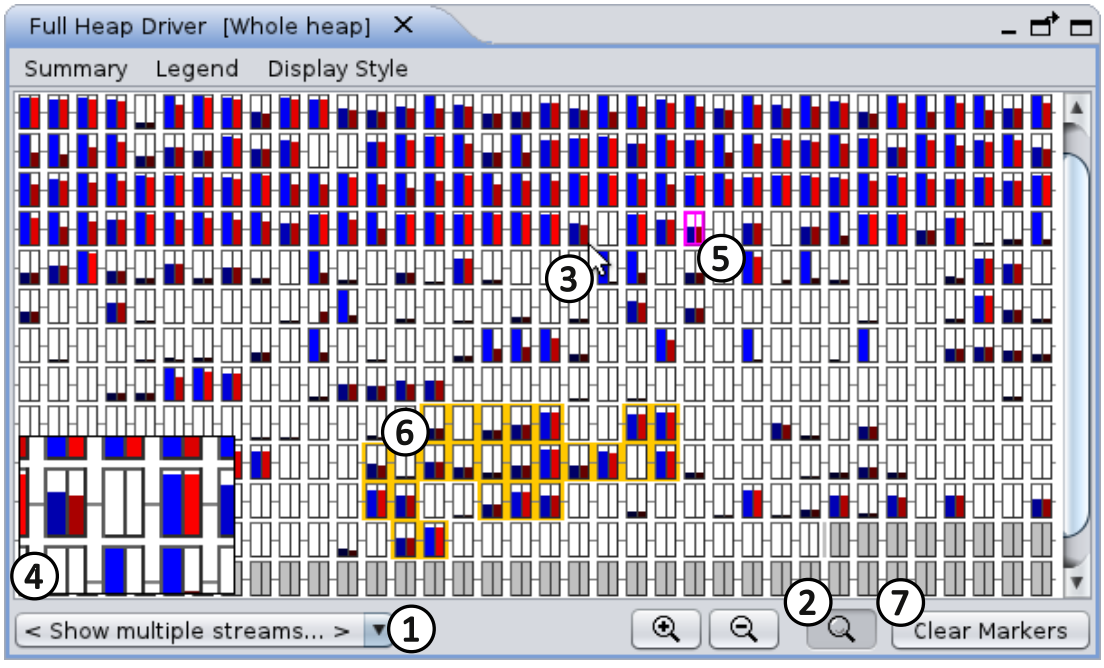

Figure 3.4: Space view showing multiple streams with magnification and markings

The figure also shows the intelligent magnification feature. It can be enabled with a button in the bottom right section ②. The area near the mouse pointer ③ is enlarged and displayed in a rectangle in one of the space view's corners ④.

For remembering a set of tiles, for example while stepping through execution on the server, tiles can marked by right-clicking them. Multiple adjacent tiles can be marked at once by dragging the mouse with the right mouse button held down. The marked tiles are highlighted with a colored background. Figure 3.4 also contains several of these marked tiles ⑥. All tile markings of a space view can be cleared with the *Clear markers* button at the bottom right ⑦.

The *Summary* button in the space view's top toolbar opens a summary view in the main window that shows data aggregated for all blocks of a space. The *Legend* action opens the legend view that shows how tiles with different values are visualized for that space. Clicking *Display Style* opens a dialog which allows changing the colors and visualization style for the space in question.

Figure 3.5 shows a summary view for a space. It lists all different enumeration values for enumeration streams such as *Allocation type*. For each enumeration value, it shows the number of blocks with that particular value on the right. The bars in the middle visualize the percentage of these blocks out of all blocks. For value streams such as *Occupancy (est)* or *#holes*, either the average value or a total of all blocks is listed and visualized.

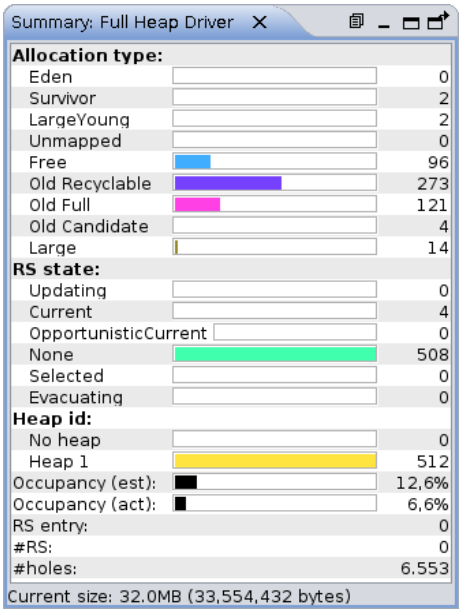

Figure 3.5: Summary for a space

Figure 3.6 depicts the *Display Style* settings dialog for a space view. This dialog allows to select colors other than the default colors provided by the server.

In the *Streams* group, the colors with which the different streams are visualized can be changed. Colors for enumeration streams cannot be set, since the enumerated values are automatically assigned different visually distinguishable colors.

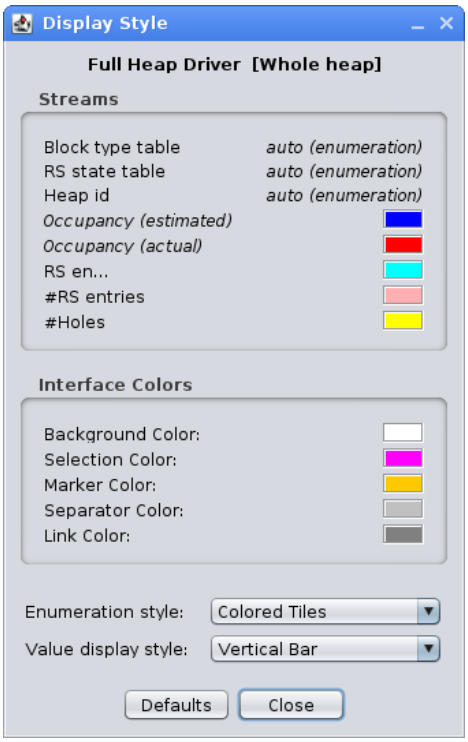

Figure 3.6: Display style settings for a space view

The controls in the *Interface Colors* group allow to change colors not directly related to the visualized heap data: the background color of the space view, the highlighting color of selected and marked tiles, and the color of separators and links between tiles.

At the bottom of the window, the visualization style for enumerations and value streams can be selected. By default, enumeration streams are rendered by filling the tile with the color associated with the block's enumeration value. However, some colors can be hard to distinguish for people with vision deficiencies or on certain display devices such as projectors. For this case, enumeration streams can also be visualized using colored or monochrome patterns of lines and dots.

Value streams are rendered with vertical bars by default, but can also be rendered with a solid fill color interpolated between black and the stream's color. In that case, black represents a value close to the lower bound of the stream's value range, while the stream color itself indicates a value near the upper bound of the value range.

3.6 Event filters

Event filters are used to enable or disable reporting of certain events and perform additional actions when they occur. Event filtering is performed by the server, but the filter settings can be changed in the client. The event filter settings dialog can be opened with the *Event Filters...* entry in the main window's *Utilities* menu.

Figure 3.7 depicts the event filter dialog. It contains a table with a row for each event that can occur in the server and its current filter settings. These settings in the table's cells can be changed in place. Using the *Reset* button at the bottom left of the dialog, all or only some of the settings in the dialog can be reset to their defaults. Clicking *OK* closes the dialog and passes the new filter settings to the server. The *Cancel* button closes the dialog, disregarding any changes.

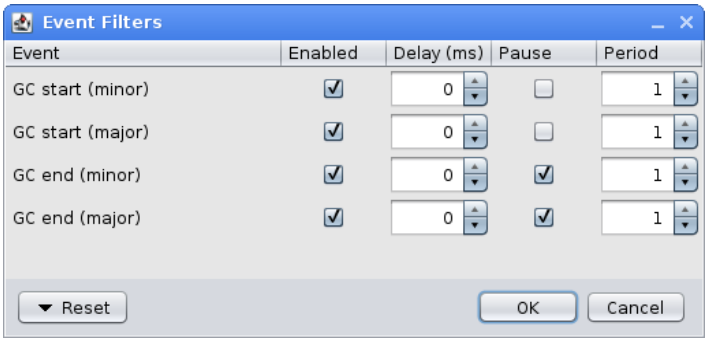

Figure 3.7: Event filter dialog

Using the checkboxes in the table's *Enabled* column, single events can be enabled or disabled. When a disabled event occurs in the server, the client is not notified of the event and no heap information is transmitted.

In the *Delay* column, a delay in milliseconds can be entered that should be imposed when such an event occurs.

When the checkbox in the *Pause* column is checked, execution on the server is suspended until the user resumes execution when this type of event occurs.

The value in the *Period* column specifies that only every *n-th* occurrence of the event should be reported to the client. For example, a value of 3 means that only the first, fourth, seventh, etc. occurrence of the event should be reported while the events in between are ignored.

Chapter 4

Technical documentation

4.1 Overview

This chapter documents the implementation of GCspy. It outlines the organization of the GCspy source code and describes the most important Java classes and the relationships between them.

4.2 Organization of the source tree

At the top level, GCspy's source code is separated into three subdirectories:

c contains a C language implementation of GCspy's server-side component as well as additional utility functions for working with GCspy's data structures.

cpp also provides an implementation of GCspy's server infrastructure complete with utility functions, but as C++ classes which are more suitable for integration into object-oriented C++ code. This implementation is stand-alone and does not reference the C code.

java contains the source code for the GCspy client as well as a Java implementation of the server-side component. It references neither of the other two other code bases.

The code for each language also includes implementations of drivers for testing and working examples. Directories `c` and `cpp` are split further into the subdirectories `src` and `include`, which separate the implementation and the header files for inclusion from other projects.

The top-level build script as well as the build infrastructure for the C and C++ code bases are realized as makefiles compatible with GNU make. Apache Ant is used for building the Java source code.

When building the C and C++ code bases, the directories `lib` and `obj` for build artifacts are created as subdirectories of `c` and `cpp`. The `obj` directory contains the object files

that the compiler creates. The `lib` directory holds the final build result, which is a library that can be linked against.

4.2.1 Java source tree

The `java` directory contains the `gcspy` directory at its root. It represents a Java package of the same name and holds all of GCspy's Java source code. The two other subdirectories `lib` and `icons` in `java` contain third-party libraries that GCspy depends on and graphics that are displayed in the GCspy client's user interface.

The Java source code is built using Apache Ant with a supplied `build.xml` file. During the build, the additional subdirectory `bin` for build artifacts is created. The build result is a Java archive file (JAR) in `bin` containing all compiled GCspy classes and the third-party dependencies for running the GCspy client.

The Java code base also contains several plugins for the GCspy client. The source code of these plugins is not separated from the main source tree. However, all plugins have their own subpackage in `gcspy.vis.plugins` and are built into distinct JAR files which are placed in the `plugins` directory.

After a build, the GCspy client can be started from the command line by running

```
java -jar gcspy-<version>.jar
```

from the `bin` directory. Alternatively, the GCspy client can be launched from a file browser, provided that it associates JAR files with a Java runtime environment.

Package hierarchy

The classes in GCspy's Java source code are organized in several packages. This section outlines the hierarchy and content of these packages.

gcspy The top-level package with the GCspy client's main class.

gcspy.comm Classes for client-server communication.

gcspy.utils General utility classes.

gcspy.tools Standalone command-line tools for dealing with trace files.

gcspy.interpreter General "interpreter" for processing data from communication, and classes representing GCspy's abstract data structures such as spaces and streams.

gcspy.interpreter.client Client-specific interpreter.

gcspy.interpreter.server Server-specific interpreter.

gcspy.vis User interface, visualization and most other classes of the GCspy client.

gcspy.vis.plugins Code of the plugin framework and parent package for all plugins.

gcspy.vis.plugins.histogram Histogram view plugin.

gcspy.vis.plugins.text Tabular text view plugin.

gcspy.vis.plugins.history History plugin.

gcspy.vis.utils Utility classes used by the GCspy client.

Because the primary scope of this thesis were improvements to the GCspy client, almost all of the changes described in this chapter were made in `gcspy.vis` and its subpackages.

4.3 Class overview

This section characterizes GCspy's most important Java classes and the relationships between them. The descriptions only intend to provide an overview over the code base and do not include all of the packages, classes, methods or attributes. Several details have been abstracted or left out to facilitate a better understanding.

4.3.1 Communication and data structures

This subsection describes relevant classes responsible for communication between client and server and classes that represent GCspy's abstract data structures. These classes are unrelated to the GCspy client's user interface or visualization. Many of the classes are shared between client and server implementations.

Figure 4.1 shows a diagram of the classes described hereafter. The descriptions explain the function and relationships of each class from top to bottom and left to right in the diagram, grouped by package.

Package **gcspy.comm**

SocketClient is a wrapper around Java's `Socket` class that implements communication functionality commonly used in GCspy.

CommandStream implements a data stream for sending and receiving *commands*, which are messages transmitted between server and client. Each instance of `CommandStream` keeps a collection of `Command` objects representing valid commands. When a `CommandStream` receives a certain type of command, it calls the `execute()` method of the corresponding `Command` object and passes the received data.

Command is the interface which actual commands (such as a *pause command*) have to implement. Classes that realize this interface perform an action specific to their type of command in their implementation of the `execute()` method.

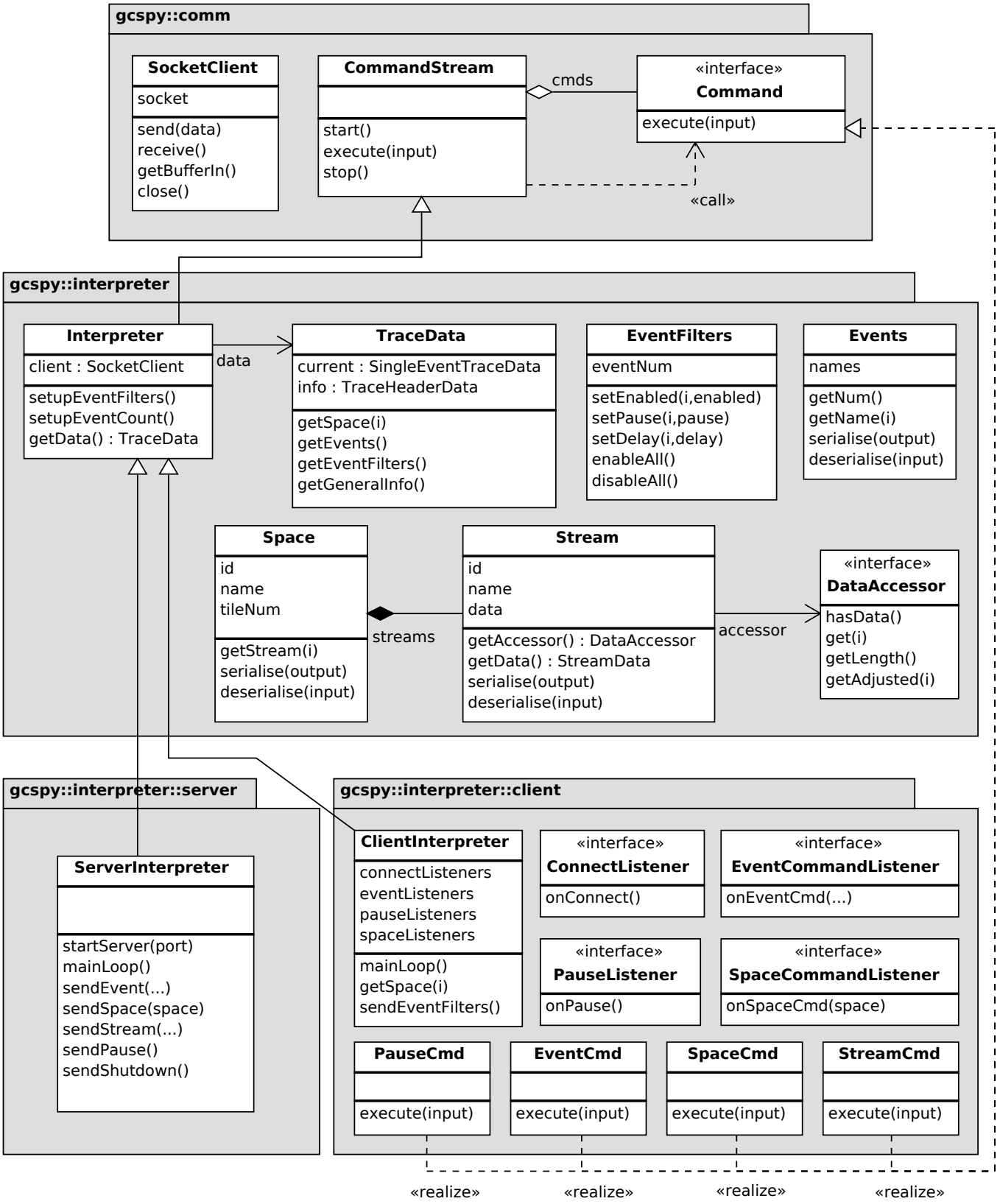

Figure 4.1: Diagram of GCspy's non-visualization Java classes

In addition to the classes described here, `gcspy.comm` also contains helper classes for reading and writing buffers.

Package **gcspy.interpreter**

Interpreter is a subclass of `CommandStream` and extends it with functionality to manage trace data using class `TraceData`. Trace data describes the state of the heap and is transmitted from the server to the client. `Interpreter` also provides public methods to obtain its trace data.

TraceData stores the state of the heap from the most recent event and additional meta-data about the trace itself. It provides public methods to access this information.

EventFilters holds the configuration of event filters, which are rules for pre-processing events on the server. It allows other objects to inspect or modify event filters.

Events holds information about the number and names of events that can occur on the server. The class also provides methods to serialize or deserialize this information for transmission.

Space represents a GCspy space as described in chapter 2, which is typically a region of the heap. It stores information about the space itself, such as its identifier, name and the number of blocks, as well as the streams of the space. The `Space` class has methods for serialization and deserialization, which also serialize or deserialize all the streams of the space.

Stream represents a GCspy stream, i.e. an attribute of the blocks of a space. The `Stream` class allows to access the stream's values in two ways. The method `getAccessor()` provides an accessor object that allows to obtain the stream's value as well as other information for specific blocks of the space. Calling method `getData()` instead returns the underlying instance of the `StreamData` class (not shown in the diagram) for direct access to the data. Like classes `Events` and `Space`, the `Stream` class also provides methods for serializing and deserializing its data for transmission.

DataAccessor is an interface that defines methods for reading a stream's value for a block by its index. The actual implementation class used for a stream depends primarily on the stream's data type.

Package **gcspy.interpreter.server**

ServerInterpreter is a subclass of `Interpreter` and implements server-side communication. This includes listening for client connections, providing the heap layout when the connection is first initiated, transmitting heap data when events occur as well as pre-processing events according to the event filters in place.

This package also contains implementations for all client-to-server commands as realizations of the `Command` interface. These classes are not shown in the diagram or described here, but the commands are characterized in appendix A.

Package **gcspy.interpreter.client**

ClientInterpreter derives from the `Interpreter` class like `ServerInterpreter`. The `ClientInterpreter` class handles all client-side communication. This involves connecting to a server, waiting for events, sending pause or resume commands, or submitting updated event filter settings to the server after they were changed in the client.

The `ClientInterpreter` class employs the observer pattern to notify other objects, particularly user interface components, when a certain type of action occurred. There are several listener interfaces in the same package, one for each type of action that can be subscribed to. Objects that want to receive notifications have to provide an implementation for one or more of these interfaces and register it with the `ClientInterpreter` instance. The `ClientInterpreter` keeps a list of subscribers for each type of listener and notifies all of them when an action of that type occurred.

ConnectListener is a listener interface for receiving notifications after successfully establishing a connecting to a server.

EventCommandListener is an interface for observing received *event* commands that the server sends after a memory management event has occurred.

PauseListener is an interface for subscribing to *pause* commands which the server sends when it has paused execution (typically as a result of a pause request from the client).

SpaceCommandListener is an interface for observing received *space* commands with which the server sends a new snapshot of a space and all its streams.

PauseCmd, EventCmd, SpaceCmd and StreamCmd are examples for realizations of the `Command` interface for server-to-client commands. The package contains several more classes for commands that are not described here. These commands and all other server-to-client commands are described in appendix A.

4.3.2 Client and visualization

This subsection describes the most important classes that are specific to the GCspy client. These classes make up the GCspy client's user interface, handle visualization of trace data, provide a framework for adding plug-ins and implement additional functionality as command-line utilities.

Figure 4.2 shows a class diagram comprising these classes. The following descriptions explain the function and relationships of each class in the diagram, grouped by package.

Package **gcspy**

Main contains the entry point for the graphical GCspy client, `main()`. The method creates GCspy's main window by instantiating class `MainFrame`.

Package **gcspy.tools**

These classes provide additional stand-alone command-line tools and do not belong to the graphical GCspy client (visualizer).

TerminalClient provides a separate command-line client that connects to a GCspy server and writes the received data to the terminal as formatted text. Like the full client, `TerminalClient` uses a `ClientInterpreter` instance for handling client-side communication.

TerminalStoreTrace is another command-line client that also connects to a GCspy server, but stores all received data in a trace file using `ClientInterpreter` and `StreamTrace`.

TerminalPlayTrace is a command-line tool that acts as a GCspy server and replays recorded trace data from a trace file to a client, also using `StreamTrace`.

StreamTrace implements reading trace data from as well as writing trace data to a file (or any stream of bytes).

Package **gcspy.vis**

MainFrame implements the GCspy client's main window. It provides access to all frequently used functionality. The largest part of the main window is a docking area that contains views for the spaces of the heap as well as information about the trace.

ConnectionDialog provides a dialog window where users can specify a remote host to connect to or a trace file to open. Instances are created by `MainFrame` on demand.

EventFilterDialog is another dialog window. It allows users to modify the event filters of the ongoing trace. `EventFilterDialog` is also instantiated by `MainFrame` as needed.

Indexer stores the offsets of events and heap snapshots within a trace. This class is used by `MainFrame` to be able to navigate through past events.

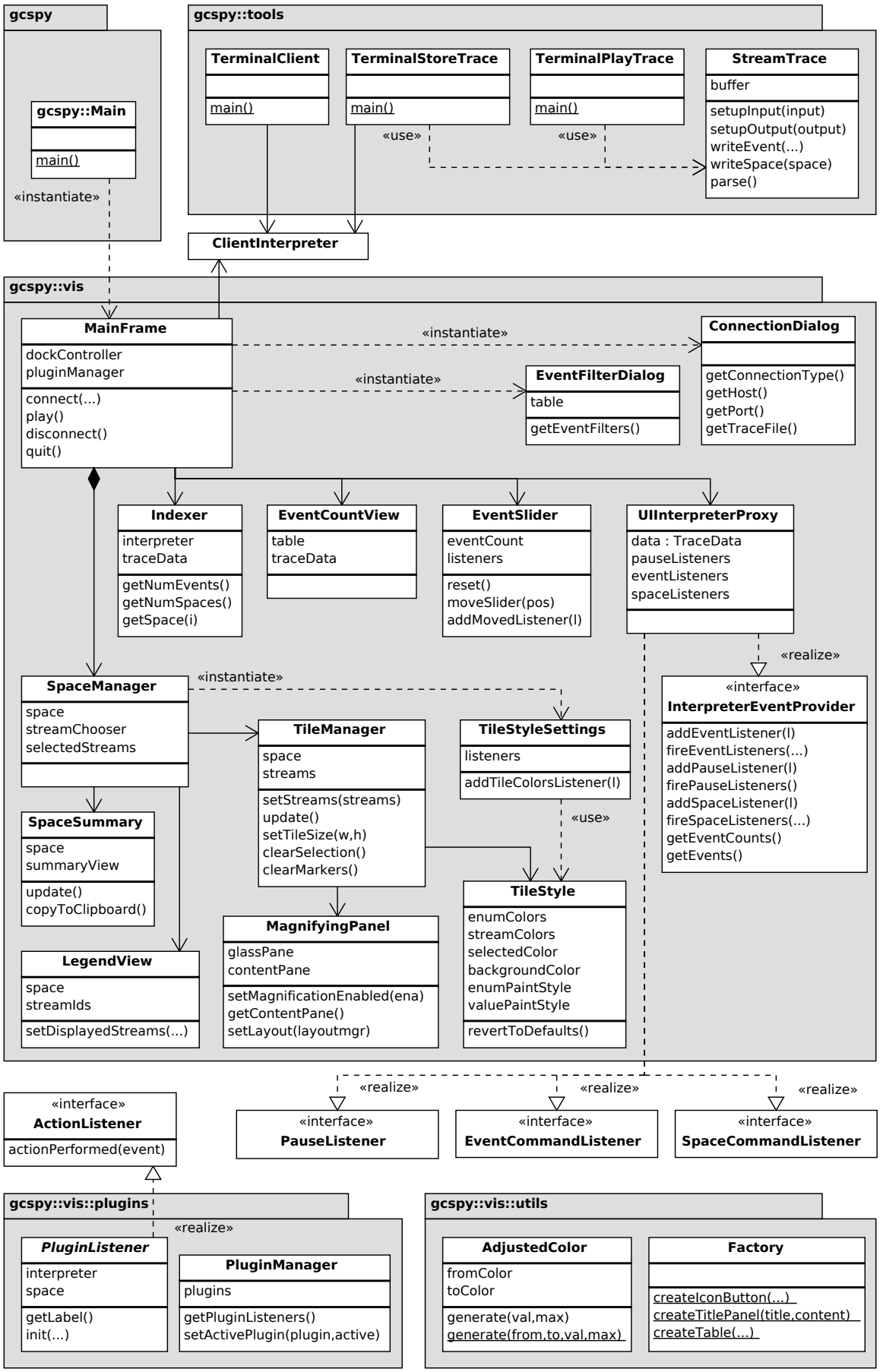

Figure 4.2: Diagram of GCspy's visualization and user interface classes

EventCountView is a user interface component that displays a table of all event types and how often they occurred in the current trace. The component is part of `MainFrame`, but not visible by default.

EventSlider provides a slider component for the user interface and a text input field with an event number. Users can navigate to events by clicking and dragging the slider or by entering a particular event number. `MainFrame` contains a single instance of `EventSlider` in its bottom panel.

UIInterpreterProxy forwards events from a `ClientInterpreter` to subscribed user interface components. For receiving events, `UIInterpreterProxy` itself realizes several of the `ClientInterpreter` listener interfaces. The `UIInterpreterProxy` class was introduced in the scope of this thesis and is discussed in more detail later in this chapter.

InterpreterEventProvider is an interface that defines methods for subscribing to events from `ClientInterpreter`, but also for triggering events which originate from within the user interface. This interface is implemented by class `UIInterpreterProxy`. `MainFrame` provides an instance of `UIInterpreterProxy` to user interface components so they can subscribe for the types of event they are interested in.

SpaceManager is a user interface component that visualizes a single space using `TileManager`. It allows choosing the visualized stream(s) and viewing additional information with controls in panels at its top and bottom.

SpaceSummary implements a graphical summary view of the values of all streams of that space. This view is not enabled by default.

LegendView implements a legend view for the currently visualized stream(s). It shows the appearance of tiles with various values.

TileManager visualizes the tiles of a space, showing one or more streams. It uses `MagnifyingPane` to enlarge the area near the mouse pointer when magnification is enabled.

MagnifyingPanel is a type of panel that can display an overlay with a magnified subsection of the panel's content.

TileStyle holds the colors and visualization options that are used by `TileManager` when rendering the space.

TileStyleSettings implements a dialog window that allows to change the used colors and visualization options. It is instantiated by `SpaceManager` and manipulates the `TileStyle` object which `TileManager` uses.

Package **gcspy.vis.plugins**

PluginListener is an abstract class central to GCspy's plugin framework. Each plugin for GCspy must provide a concrete plugin listener derived from this class. When the user activates a plugin in the user interface, GCspy calls the plugin listener's `actionPerformed()` method defined in superinterface `ActionListener`. In its implementation of this method, the plugin can react accordingly, for example by opening a dialog window.

PluginManager determines which plugins are available by scanning GCspy's plugins subdirectory for JAR files. When a server connection is established, `MainFrame` calls `PluginManager` to load all enabled plugins and return objects for their `PluginListener` implementations for integration with the user interface. Single plugins can be disabled and will then not be loaded.

Package **gcspy.vis.utils**

AdjustedColor allows to interpolate between two colors, given a start and end color and positive integers for value and maximum value (the minimum value is always considered to be 0).

Factory provides static convenience methods to construct commonly used user interface components.

Chapter 5

Enhancements

5.1 Overview

This chapter characterizes the enhancements to GCspy that were introduced in the scope of this thesis. Each of the following sections discusses a change or set of changes. A section starts with the motivation or requirements that led to the change. Then it describes the design of the solution as well as other approaches that were considered. Finally, the section characterizes the aspects of the final implementation.

5.2 Prior improvements

The source code of this thesis is based on GCspy release 1.0.12 [GCspy]. Before I started working on the implementation, my supervisor had already introduced several improvements to make GCspy more suitable for using it in a different project. These changes are not described in detail here and only listed in short:

Replaying and saving trace files in the user interface

Prior to this change, replaying trace files required manually starting a local replay server from the command line and connecting to it with the GCspy client. Saving trace data to a file was also done with a command-line utility that connects to the GCspy server and writes received trace data directly into the file.

With this change in place, trace files can be replayed from within the GCspy client's user interface and the data from an ongoing trace session can be saved to a file at any time.

Navigation within traces

Originally, GCspy could only visualize the heap state from the most recent event. Viewing the heap from events that occurred before that was not supported and the navigation controls only allowed pausing and continuing execution.

This change introduced the `Indexer` class that stores information about all past events up to the current event. Using `EventSlider` and other new navigation controls, the user can now seek and view any past event of the current trace.

Build process

The Java source code used to be built with `make`, which is unusual and rather unflexible for Java code bases. The makefile was replaced with a `build.xml` file for Apache Ant. The whole GCspy client is now packaged as a single JAR file complete with required libraries while previously, only plugins were packaged. The use of Apache Ant also allows easier import of the source code in most Java integrated development environments (IDEs).

5.3 Space view usability

GCspy's original space visualization had a number of limitations that can affect a user's efficiency in using GCspy. Hence, the motivation for this change was to improve usability for GCspy by addressing those limitations. Figure 5.1 is a screen capture of the original GCspy client with only the enhancements described earlier in Section 5.2. In the following, this figure will be used to point out some of these issues.

A major limitation was that users were required to manually switch between spaces by clicking the *Activate* button below the respective space's view ①, ②. This was necessary to view information for a tile within that space or to change the visualized stream with the view chooser ③. The desired improvement here was to make spaces more independent of each other and to show tile details after clicking on a space's tile without having to activate the space first.

Large spaces raised another problem: GCspy's space views did not have any scrolling capability. Instead, the space view would display only as many tiles from the start of the space as could fit in the available area. An arrow in the bottom right corner of the view would indicate when there are more tiles in the space than currently visible ④, ⑤. The intended improvement for this issue was that users should be able to scroll through all tiles of a space as well as change the tile size to see more tiles at once.

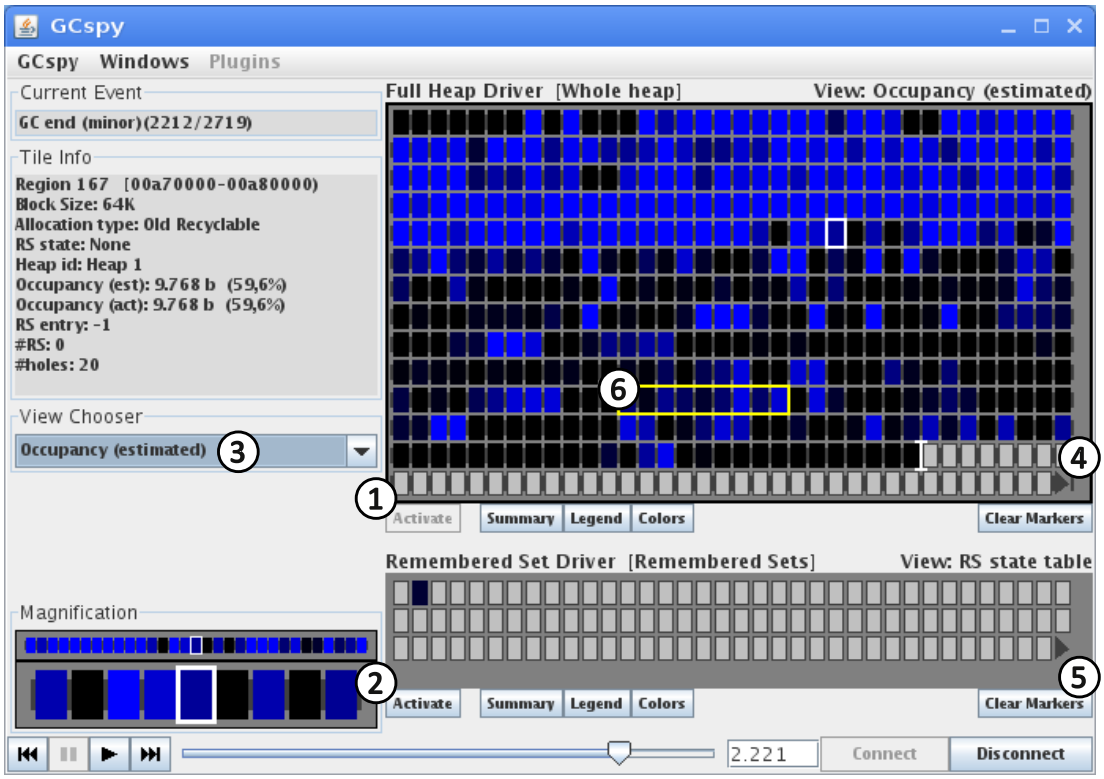

Figure 5.1: GCspy 1.0.12 with enhancements showing a trace

Another issue was the lack of keyboard navigation. After selecting a tile, the user could not use the arrow keys to navigate to neighboring tiles. Using the mouse for selecting tiles was mandatory.

Finally, marking of tiles was performed with the middle mouse button and was only possible for a single group of neighboring tiles ⑥. The objective here was to allow marking of tiles in separate locations and to avoid relying on the availability of three mouse buttons.

The first considered approach to address the described limitations was to revise and extend the existing components. This would have involved using component focus to determine the currently active space, reacting to keyboard events for navigation and adding a vertical scrollbar to reach all tiles of large spaces.

However, an inspection of the design and implementation of the space visualization components revealed that their behavior and structures were very rigid. The implementation barely used standard components and had its own implementation for rendering tiles and handling mouse events. Essentially, the class `TileManager` acted as canvas for the `SpaceManager` class to render the tiles of the space. Tests with `JScrollPane` to add scrolling capability to `TileManager` confirmed that extensive changes to the existing implementation would be necessary.

For these reasons and because other changes to tile rendering were already planned at that time, a different approach was chosen. The class `TileManager` was rewritten to use `JList` [JList], which is a standard Swing component designed to lay out objects in rows and columns and render them. The tremendous advantage of this approach is that `JList` already performs layouting of the tiles and implements keyboard navigation. Only slight changes were made to keyboard behavior so that the *Home* and *End* keys jump to the first or last tile of the current row instead of the whole space and *Ctrl-Left* and *Ctrl-Right* can be used to skip several cells when navigating.

Another important characteristic of `JList` is that it follows the *Model-View-Controller* architectural pattern (MVC) in which the collection of items (model), their visual representation (view) and user interaction (controller) are separated. This allowed for a much cleaner separation of data and visualization than before and proved very helpful for the subsequent enhancements to tile rendering which are described in later sections.

The new implementation of `TileManager` uses its `JList` in combination with Swing's `JScrollPane` class. When a space is large enough so that it does not fit the available area, `JScrollPane` automatically adds a vertical scrollbar. In addition, two new buttons were added to allow the user to increase or decrease the size of the individual tiles to fit more tiles on the screen or to get a better look at the displayed tiles. This was also easy to accomplish because `JList` allows to set a uniform size for all displayed items.

In order to remove the need for users to switch between spaces manually, spaces were made independent of each other. Each space view now has a separate drop-down list for choosing the stream to visualize. Tiles of a space can be selected without prior activation of the space, and tile details are always shown for the most recently selected tile regardless of its space. As a result, the *Activate* buttons were no longer necessary and removed.

One feature that `JList` does not provide is marking list items. This functionality was implemented in the new class `MarkerManager` that handles mouse click events and sets marked tiles accordingly. Tiles are marked by right-clicking a single tile or by holding the right mouse button and dragging the mouse pointer over several tiles. Marking new tiles no longer clears the previous marking, but instead marks new tiles or unmarks tiles. The entire marking can still be cleared with the *Clear Markers* button.

The screen capture from Figure 3.4 on page 16 shows most of the described improvements.

Figure 5.2 shows a simplified diagram of the classes involved in GCspy's new space view implementation. The following description characterizes their interaction and some implementation details.

SpaceManager uses `TileManager` to render the tiles of the space it represents, as described earlier in Section 4.3.2.

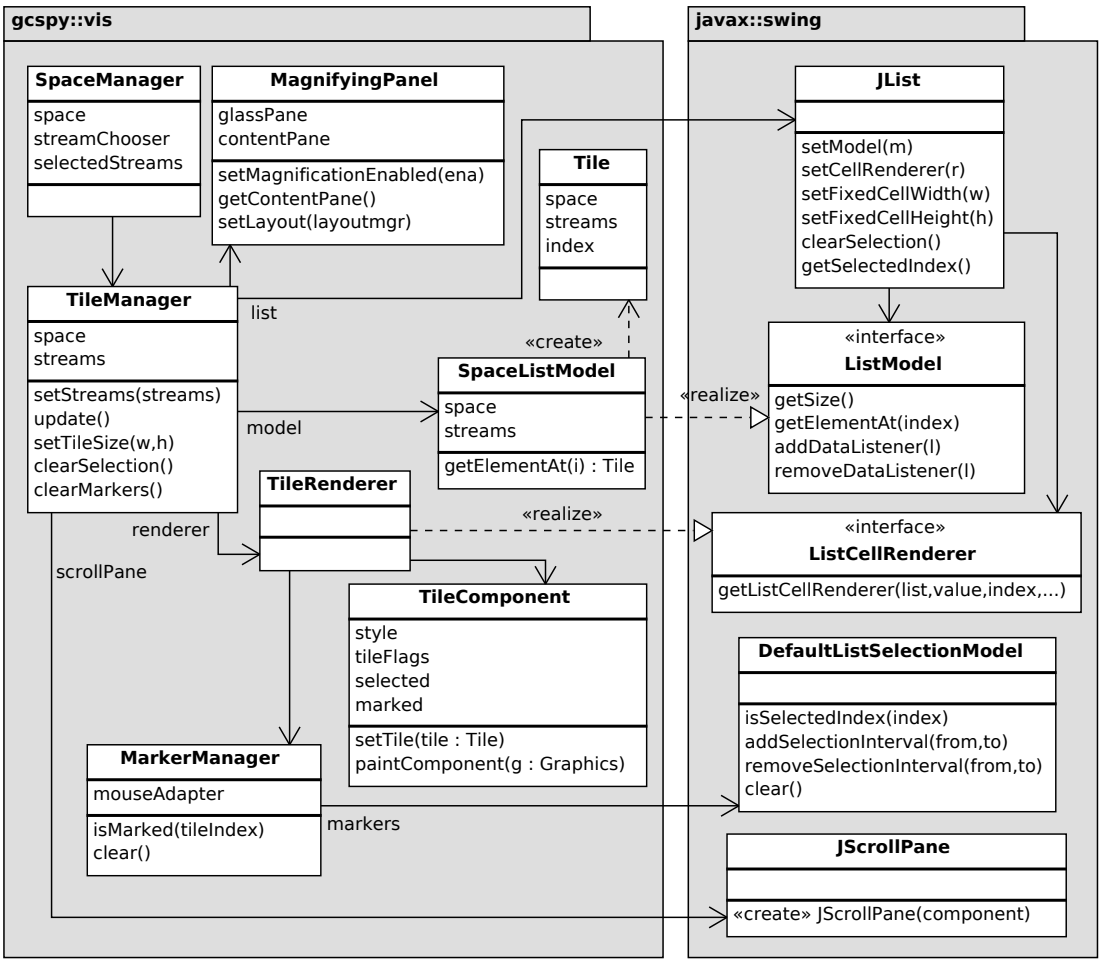

Figure 5.2: Diagram of GCspy's space view classes

JList is a standard Swing component designed to lay out and render items of a list in rows and columns. A class implementing the `ListModel` interface acts as the *model* which provides `JList` with the collection of list items to display. Another class that realizes the `ListCellRenderer` interface acts as *view* and provides `JList` with components that it can use to draw the list items on the screen.

TileManager is the class that assembles the tile view from single components. It instantiates a `JList` for rendering tiles and provides it with a `SpaceListModel` object as backing store and with a `TileRenderer` for drawing the tiles. The `JList` is placed inside a `JScrollPane` to add scrolling capabilities. The `JScrollPane` in turn is placed inside a `MagnifyingPanel` that can magnify its content.

Tile represents a single tile in a space by its index.

SpaceListModel is a list model implementation that provides `Tile` objects to `JList` as list items.

TileComponent is a visual component that renders a single tile of a space. In addition to the tile's values and special properties (such as separators or links), it also displays whether the tile is selected or marked.

TileRenderer is called from `JList` to obtain a component for rendering a specific list item. `JList` passes that list item's value, which is a `Tile` object created earlier by `SpaceListModel`. The `TileRenderer` has a private `TileComponent` instance which it then updates to match the requested tile (list item) and provides it to `JList` for drawing.

MarkerManager manages the set of marked tiles in the tile view. It handles mouse click events inside the `JList` and uses the `DefaultListSelectionModel` class to efficiently handle ranges of marked tiles.

5.4 Tile rendering

The GCspy client originally supported only a single "color-interpolating" mode of rendering tiles. The driver on the server side has to initially specify a color and upper and lower boundaries for each stream. When visualizing that stream, the GCspy client takes each block's value relative to the stream's boundaries and interpolates a color between black and the predefined color. The tile representing that block is then filled with that color. For example, if a stream's predefined color is red, a tile in dark red indicates a value near the lower boundary while a bright and saturated red tile stands for values close to the stream's upper boundary.

Figure 5.3 shows an example of how the GCspy client visualizes tiles of a stream. The values of the stream in question are between 0 and 65535, therefore the stream's lower bound is chosen to be 0 and the upper bound is set to 65535. The stream's predefined color is white, which is associated with the stream's upper bound in visualization. Black is associated with the stream's lower bound. Tiles with a value matching either bound are rendered in the respective bound's color. The color of tiles for other values within the bounds are interpolated between black and white, resulting in different shades of gray.

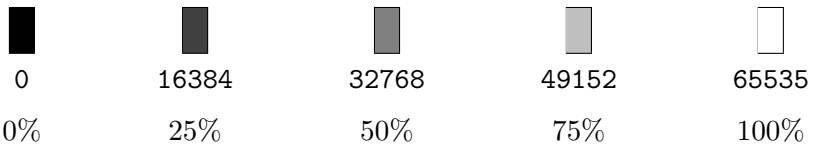

Figure 5.3: Tile visualization by interpolating colors

This mode of visualization allows users of GCspy to quickly assess which tiles of a space have low, medium, or particularly high values. Still, it is not very easy to estimate a

tile's *actual* value from its tile's color or to recognize a difference between tiles with somewhat close values such as 40% and 60%. This becomes a problem especially when using stream colors such as blue or violet where different shades become even harder to distinguish. Another issue is that this visualization relies entirely on correct color perception and therefore excludes people with a vision deficiency.

Enumeration streams were also rendered with this visualization mode. The major drawback of this approach is that with four or more enumeration values, it becomes very hard to tell the difference between the different shades and to match them to enumeration values.

Finally, only one stream of a space could be viewed at the same time. In practice however, there can be two or more streams that are related to each other, such as estimated occupancy and actual occupancy. A user can only compare such streams by viewing the details of a single tile or by repeatedly changing the displayed stream while focusing on a specific area in the space.

Resolving these limitations proved to be rather straight-forward after the changes to the space view described in section 5.3. Most of the enhancements described in the following were made in the `TileComponent` class.

First, a second visualization mode was conceived that addresses the problems with color-interpolating visualization. Instead of filling the entire tile with a color, it renders a vertical bar inside the tile. The bar's height represents the value of the tile relative to its bounds. The major advantage of this mode is that is becomes much easier to estimate a tile's value from the height of its bar. Therefore, users can also better tell the difference between tiles with close values. The effect even increases with larger tile sizes. This visualization mode also relies much less on color perception.

Figure 5.4 shows how tiles are rendered with the vertical bar visualization mode. The color of the bar is computed with the same interpolation that the original visualization method uses. This provides users with an additional visual clue for spotting tiles with particularly low or high values.

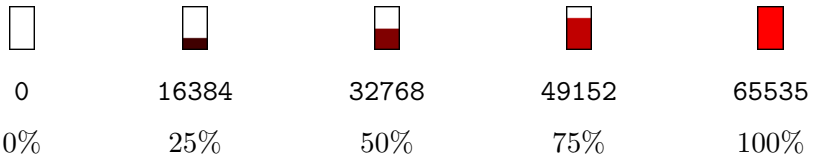

Figure 5.4: Visualization of tiles with vertical bars

Because vertical bar visualization proved superior in many cases, it was made the default for value streams. However, enumeration values were still not very distinguishable with this visualization mode. Hence, another mode was created specifically for enumeration streams.

This new visualization mode assigns each enumeration value a color from a pool of 14 distinct colors. Each tile is filled with the color assigned to its value. The first seven colors of the pool are selected by choosing bright colors with equivalent hue distances in the *Hue, Saturation, Brightness* (HSB, also called *HSV*) color model. The second set of colors are darker versions of the first seven colors.

This visualization proves to be very effective for streams with up to ten enumeration values. Users can quickly distinguish tiles with different enumeration values by their color. Therefore, GCspy automatically applies this new mode when visualizing enumeration streams. Still, this method relies on correct color perception. Users with a color vision deficiency could have difficulties distinguishing the colors. Also, some display devices such as projectors often replicate colors inaccurately.

For this reason, an additional variant of this visualization mode was added. With this variant, GCspy assigns each value of an enumeration stream one of eight different patterns composed of visual elements such as dots, a grid, horizontal, vertical, or diagonal lines. Tiles are then painted with these patterns instead of only filled with their respective colors. The pattern can be rendered in the same distinguishable colors as in the filling mode (the default) or in only black and white (monochromatic mode).

Figure 5.5 shows the legend view for an enumeration stream with monochromatic pattern visualization. GCspy assigns each block type a different, distinguishable pattern.

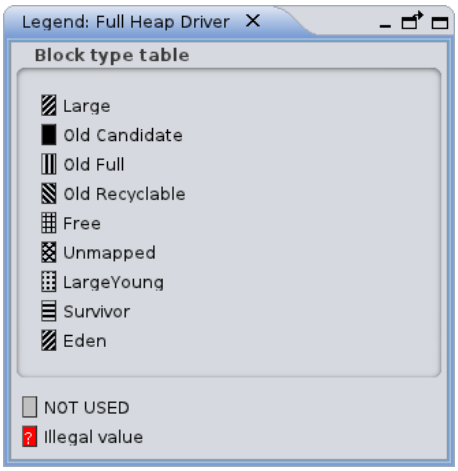

Figure 5.5: Legend of an enumeration stream with pattern visualization

The implementation of the patterns is centered around a new interface named `FillPattern` which declares a single method `fill()`. Each type of pattern like dots or horizontal lines is represented by a class realizing the `FillPattern` interface. These classes implement the `fill()` method so that it draws their kind of pattern into a rectangular area passed by the caller.

To generalize further, all visualization modes were implemented as patterns:

The *opaque pattern* fills a tile with a single color. Color-interpolating visualization and all variants of enumeration visualization use this "pattern".

The *vertical bar pattern* draws a vertical bar with a specific relative height inside the tile. GCspy uses this mode exclusively for the vertical bar visualization mode.

The *invalid pattern* draws a question mark on a red background inside the tile. This pattern is used to indicate tiles with invalid values, for example because the values is outside the stream's boundaries.

Figure 5.6 depicts the classes responsible for rendering tiles, including the `FillPattern` interface and three of its implementations. The interaction between these classes is characterized in the following.

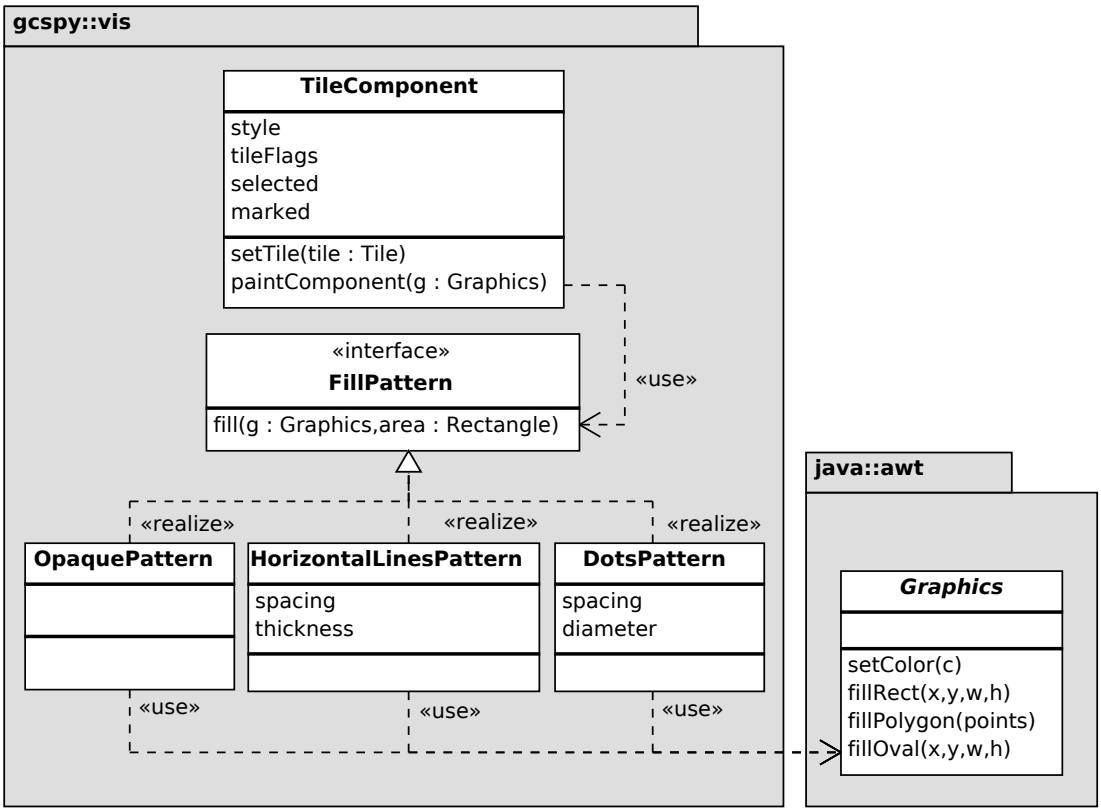

Figure 5.6: Diagram of classes responsible for rendering tiles

As described earlier in Section 5.3, `JList` asks `TileRenderer` to provide a component for drawing a specific tile on the screen. `TileRenderer` returns a `TileComponent` instance that is configured to draw the tile in question. `JList` calls the `paintComponent()` method of `TileComponent` and passes a `Graphics` object that provides methods to perform the actual drawing. `TileComponent` in turn calls the `fill()` method of the appropriate `FillPattern` implementation and passes the `Graphics` object as well as the rectangular area that should be painted. `TileComponent` itself also draws additional

visual elements such as highlighting when the tile is selected or marked, links or separators between tiles, borders, and the background.

The last remaining issue after the the modifications described so far was that it was hard to compare related streams of a space. This limitation was approached by allowing users to choose multiple streams for visualization. GCspy then vertically divides each tile into segments of equal size and visualizes each selected stream in a different segment of the tile. This feature is implemented in `TileComponent` by calling the `fill()` method on the appropriate pattern for each stream and passing the area of that stream's segment.

Figure 5.7 shows three different streams that are visualized at once in the tiles of a space. Each tile is divided into three segments. The left segment contains the *block type* enumeration stream which is rendered with patterns. The center segment visualizes the garbage collector's estimate of the occupancy of each block as a vertical bar. The right segment contains a block's actual occupancy and is rendered as vertical bar as well.

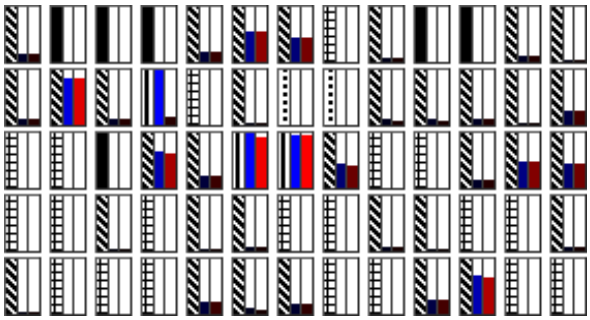

Figure 5.7: Visualization of three streams at once

Figure 5.8 depicts the dialog for selecting multiple streams for visualization. Users can open it by choosing <*Show multiple streams...*> in the stream selection drop-down list. A user can choose at most three streams to display at the same time.

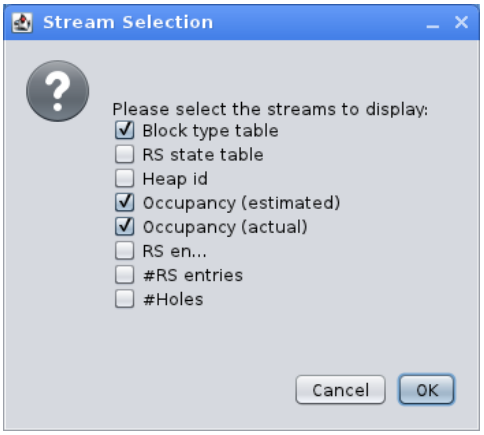

Figure 5.8: Dialog for selecting multiple streams for visualization

5.5 Magnification

GCspy 1.0.12 already included a magnification feature for tile views that was located in the bottom left corner of the main window. This feature can be seen in Figure 5.1 on page 32. The magnification area shows the neighboring tiles of the currently selected tile. The upper row of tiles uses a small tile size and provides an overview of the tile's neighborhood. The lower row uses a large tile size and intends to make the visual differences between the tiles more noticeable.

This approach to magnification has several disadvantages. Firstly, a tile must be selected to be shown in the magnification area. The previous selection is lost in the process. In earlier GCspy releases, selecting a tile could also require manually activating that tile's space first. Secondly, when inspecting a space view and using the magnification feature, the user has to repeatedly shift focus between the space view and the magnification area because they are in entirely different parts of the main window. Finally, the magnification area takes up space even when it is not being used.

The magnification feature might appear less important now that GCspy allows to change the tile size in space views. However, it still proves very useful for large spaces where small tiles are helpful for instantly spotting many particularly high or low values and the magnification feature allows to quickly get a more detailed view. Hence, a more sophisticated replacement of this feature was conceived.

The new *intelligent magnification* area is placed in a corner of the space view itself instead of a separate dedicated region of the main window. It magnifies the area near the mouse pointer rather than neighbors of the selected tile. When the user moves the mouse pointer near the magnification area, the area automatically moves to a different corner so it doesn't obstruct the user's view. Figure 3.4 on page 16 shows a space view with the magnification area in the bottom left corner ④.

This magnification mechanism is implemented in a single class named `MagnifyingPanel` which is depicted in the class diagrams shown in Figure 4.2 on page 27 and Figure 5.2 on page 34. `MagnifyingPanel` acts as container for other user-interface elements just like a normal panel. However, when magnification is enabled and the mouse pointer is inside the boundaries of the `MagnifyingPanel`, it draws a magnified view of the region near the mouse pointer into a rectangle in one of its two bottom corners.

Figure 5.9 shows how the functionality of `MagnifyingPanel` is implemented using layered panes. The `MagnifyingPanel` places the actual user-interface components it contains in the *content pane*, which is the bottom-most layer in terms of depth. Above the content pane is a *glass pane*. As the name suggests, the glass pane is fully transparent, but it intercepts any mouse events intended for the elements in the content pane. These events are necessary for tracking the position of the mouse and are forwarded to their actual recipients after processing.

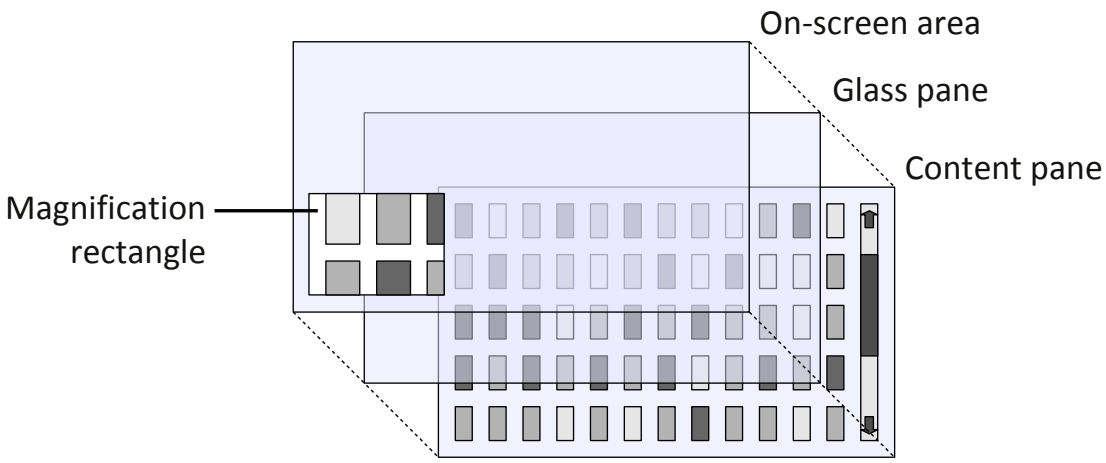

Figure 5.9: Layers of the magnifying panel

The `MagnifyingPanel` lets the panels draw into a buffer instead of directly on the screen. From this buffer, it can extract the region near the mouse pointer and enlarge it. `MagnifyingPanel` then draws the content of the buffer on the screen and adds the magnification rectangle with the enlarged region as overlay.

5.6 Space summary and tile property views

GCspy originally used plain text areas to display information for tiles and spaces. Figure 5.10 shows two examples for these text areas. The tile property view is located in the left pane of the main window. It shows information about the location of the tile within the space in the first line and then lists all stream values for the tile. Depending on the type of stream, a stream's value can also be shown as a percentage, which is the case with the *Occupancy (est)* and *Occupancy (act)* streams.

The space summary is in a separate window on the bottom right. It shows a summary for all streams over an entire space, in this case the *Remembered Sets* space. The blocks of this space represent remembered sets of references between heaps. The *RS state* stream specifies the current state of such a remembered set, while the *Heap Id* stream specifies the heap where the references originate. The summary values that the summary view displays are provided by the server. Enumeration streams such as *RS state* and *Heap Id* typically give the number of tiles with a particular enumeration value in their summary values.

Other streams represent a count, such as the *#holes* stream in the *Whole heap* space. This stream specifies the number of gaps ("holes") between the objects in a block. A meaningful summary value for this stream would be the sum over the values of all blocks, i.e. the total number of holes in the space. For streams specifying relative values or percentages, such as estimated and actual occupancy, the average value over all blocks is an informative summary value.

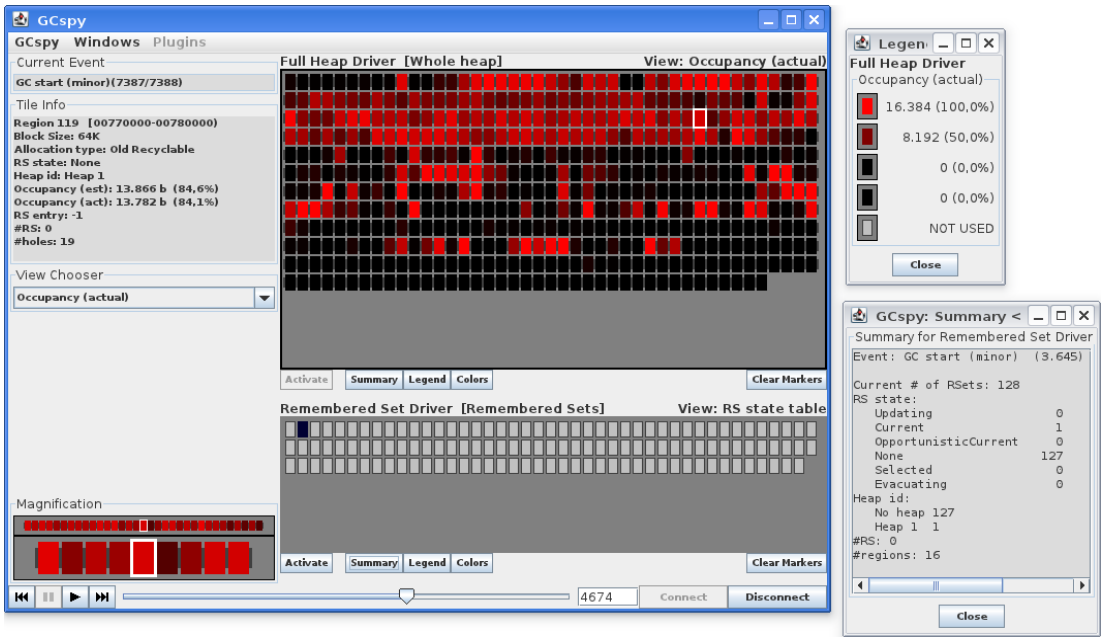

Figure 5.10: Main window and views in external windows in GCspy 1.0.12

The plain-text views are simple to generate and display. However, they require the user to read through the displayed text and put the values in context. In order to allow users to more quickly judge the state of a tile or space, it was decided that lists with additional horizontal fill bars in their rows should replace the text views. The size of a bar should indicate the stream's relative value, for example the percentage of tiles of a particular type in the entire space.

One advantage of the text views was that users could select and copy text to the clipboard and use them in other applications or save them in a file. The new type of view retains this functionality with a small button that allows to copy a text representation of the entire view's content to the system clipboard.

Figure 3.5 on page 17 shows a space summary with this new type of view. *Allocation type, RS state,* and *Heap id* are streams in the *Whole heap* space. The length of the bars of the individual enumeration values is based on the total number of tiles with that value in the space. Other streams for which the server provides a percentage in the summary, such as the *Occupancy (est)* and *Occupancy(act)*, use that percentage for the bar's size. The fourth button from the right in the tab panel at the very top allows the user to copy a text representation of the view's contents to the clipboard.

The properties view shown in Figure 5.13 on page 48 also uses this new type of visualization. This figure also depicts a space summary view that displays the same information as the space summary in Figure 5.10.

Figure 5.11 depicts the classes involved in the new space summary view. Each `SpaceManager`, which represents a space view, has a `SpaceSummary` panel. This panel

contains a `SpaceSummaryView` component which is reponsible for visualizing the space summary. This component uses a `PropertyListView` which in turn relies on `JList` for rendering the list. Class `JList` follows the MVC pattern described earlier in Section 5.3. The `SpaceSummaryView` provides a `SpaceSummaryListModel` as model for `JList`. This model creates an instance of `PropertyItem` for each list row. Such an instance holds the name displayed on the left side, whether the item represents a heading, the value to display on the right side, as well as whether and how a horizontal bar should be rendered. The `PropertyListView` on the other hand provides an instance of `PropertyItemRenderer` to its `JList` to render list items. The `JList` calls the renderer with `PropertyItem` instances from the model. The renderer then updates a private `PropertyItemComponent` instance to reflect the values from the `PropertyItem` and returns it to `JList` for drawing. `PropertyItemComponent` in turn uses the `DiagramBar` component to render the horizontal bar.

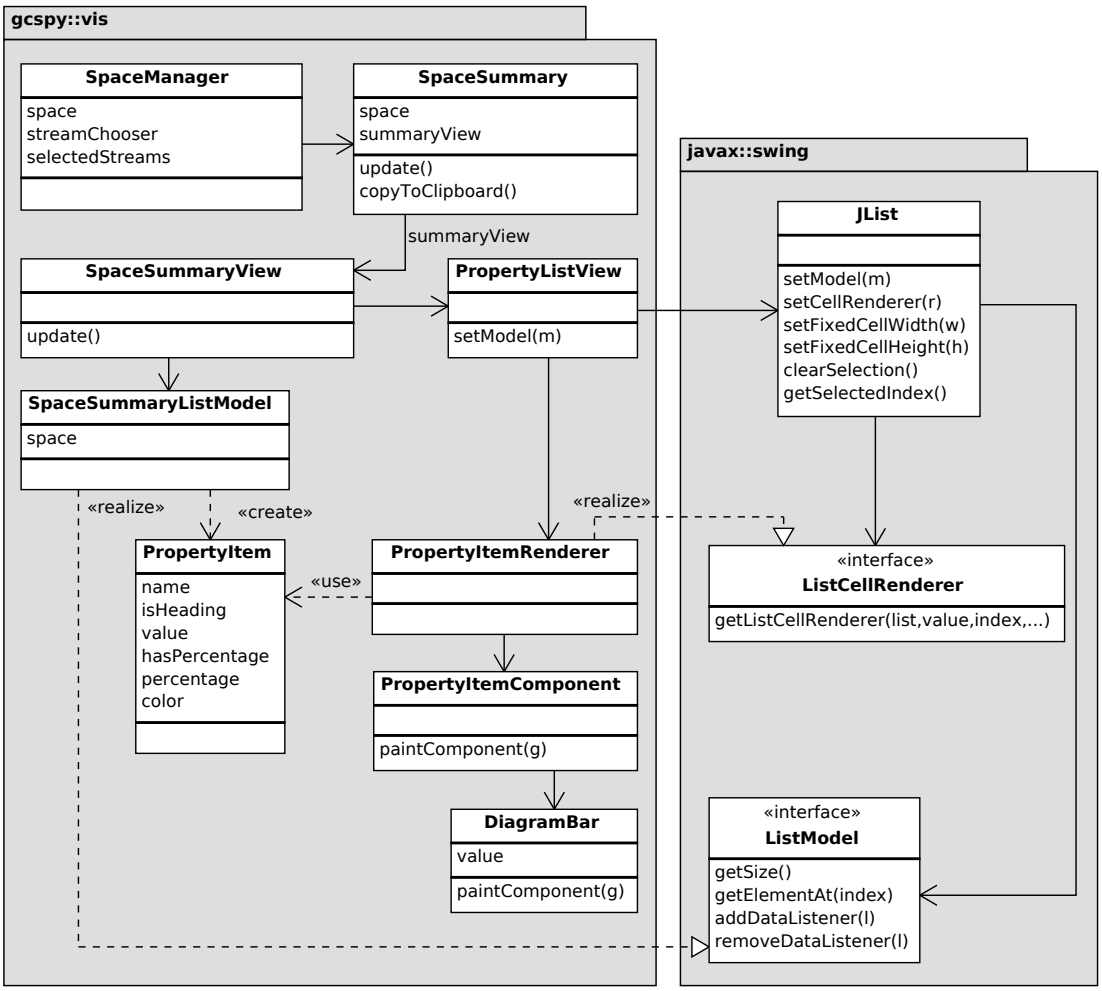

Figure 5.11: Classes of the new summary view

The mechanism for displaying properties is slightly more complex because there is only one property view that the multiple space views share.

Figure 5.12 shows the classes involved in the property display mechanism. `PropertyView` is an interface for components that display properties. Such a `PropertyView` has only a single provider of properties at the same time (or none). Whenever an object other than the current provider calls `setProperties()`, this object becomes the new provider and the `PropertyView` notifies registered `PropertySourceListener` objects that its provider has changed. `AbstractPropertyView` is an abstract class that implements part of the functionality that the `PropertyView` interface requires.

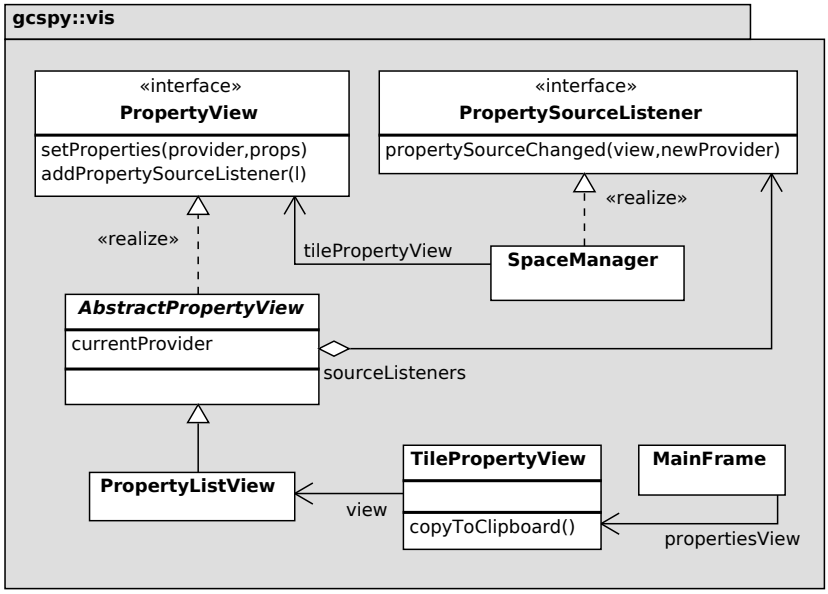

Figure 5.12: Classes involved in the property display mechanism

The `PropertyListView` class described above extends the `AbstractPropertyView` class and takes care of the visualization of properties. The `MainFrame`, i.e. GCspy's main window, has a single property view that uses `PropertyListView`. Each `SpaceManager` instance is provided with a reference to the property view and immediately registers itself as `PropertySourceListener`.

When the user clicks a tile in the space view, that space's `SpaceManager` calls the property view's `setProperties()` method and becomes its new property provider. When the user later clicks another tile in a different space view, that space's `SpaceManager` also calls `setProperties()` on the property view and the property provider changes again. The `PropertyView` now calls the `propertySourceChanged()` method of all listeners, including the `SpaceManager` object which formerly acted as provider. That `SpaceManager` then clears its selection because the user has selected another tile in a different space. As long as a `SpaceManager` is the current property provider (i.e. a tile in its space is currently selected), it updates the properties whenever the tile's values change, typically after an event from the server.

5.7 Docking views

The original main window of the GCspy visualizer, which is shown in Figure 5.10 on page 42, was partitioned into three panels. The *tools panel* on the left side displayed the current event, information about the selected tile, the stream chooser for the active space, and the magnification area. The *space panel* occupied the right side of the main window and contained all space views. The *navigation panel* at the window's bottom showed controls for navigation in the trace and the connect/disconnect buttons.

GCspy uses layout managers to adjust the layout of its panels and views to different window sizes. However, it did not allow to resize or hide single panels or views. As a result, working with multiple large spaces could be inconvenient because all space views were shown at the same time and had to share the available space, even though a user would often work only with a single space at the same time.

Additional views such as the summary and legend views of a space were not shown by default and opened on request in separate windows. Figure 5.10 on page 42 shows two such windows as well. However, these windows did not stay on top of the main window. When working with a maximized main window or with limited space on screen, it was often required to switch between the main window and these separate windows using the task bar, which proved inefficient and inconvenient.

In order to resolve these issues, it was decided that GCspy should be equipped with a more flexible and modern user interface. In particular, the GCspy visualizer should implement the docking views paradigm that many applications use, most notably the major integrated development environments (IDEs) such as the Eclipse IDE, NetBeans IDE or Microsoft Visual Studio.

In this paradigm, the user can resize panes and views by clicking and dragging their borders. Panes and views also provide buttons to hide, minimize or maximize them in the main window. The user can click and drag a view or pane within the main window to change its position, but also drag it outside the main window so it becomes a separate, independent window.

There are several implementations of docking functionality for Java that are available for non-commercial use. Many of them were evaluated and considered for use in GCspy. The following description characterizes these implementations and reflects the situation at the time of writing, April 27, 2011.

NetBeans RCP. The NetBeans Rich Client Platform (RCP) provides well-developed and feature-rich docking capabilities to applications built on it [NetBeansRCP]. However, NetBeans is a rather heavy dependency for the otherwise lightweight GCspy client. Moving GCspy to NetBeans RCP as base would require considerable effort that seems unjustified for the gained functionality.

Eclipse platform. The Eclipse platform also provides mature and rich docking to its applications [EclipsePP]. But like NetBeans RCP, it constitutes a heavy dependency and would require substantial changes to GCspy. In addition, Eclipse is based on the *Standard Widget Toolkit* (SWT) while GCspy uses the Swing toolkit. Although Swing code is supported in Eclipse, it could lead to visual inconsistencies and other cross-toolkit issues [Hirsch07].

Docking Frames Docking Frames is an open source docking framework for Swing licensed unter the GNU Lesser General Public License (LGPL) [DockingF]. Its goal is to offer a maximum of flexibility. The project is very active and frequently releases new versions: the most recent version 1.1.0 was released only several days ago. Docking Frames also comes with an exhaustive documentation.

VLDocking. VLDocking is a Swing docking framework that claims to provide rich docking features and easy integration into existing applications [VLDocking]. VLDocking was developed by the company *VLSolutions* which just closed when the framework was first evaluated. However, development of the framework seems to continue slowly, although no releases are currently offered for download.

Flex Dock. Flex Dock is a stand-alone windowing and docking framework for Swing [FlexDock]. The project appears to be barely active. Its website on *java.net* only provides access to the source code repository and does not offer any downloads[1].

JDocking. The JDocking project provides the docking functionality from NetBeans as a stand-alone framework. However, the project appeared to be unmaintained when first evaluated and at the time of writing, the project's website on *java.net* can no longer be reached[1].

There are also two commercial docking frameworks for Swing that are dual-licensed as free software under the GNU General Public License (GPL): *InfoNode Docking Windows* [IDockWin] and *Sanaware Java Docking* [SanaJDock]. However, since GCspy is licensed under a BSD-like license which is incompatible to the GPL, it could not use either of them. Also, the most recent releases of both projects are more than two years old, which indicates low activity.

It was finally decided that GCspy's new docking functionality will be implemented with the *Docking Frames* framework. Docking Frames provides a high level of flexibility, but comes with a "common" library that allows to use frequently used functionality with little effort. The framework includes more than 100 pages of illustrated documentation and the source code is annotated with helpful JavaDoc comments. The project frequently

[1] Earlier in 2011, Oracle moved *java.net* to *Project Kenai*. Not all projects were migrated and some content may have been lost.

publishes new releases, which is an indicator that the framework will continue to be maintained.

The contents of GCspy's main window, with the exception of the menu bar at the top and navigation panel at the bottom, were replaced with a single large *docking area*. This docking area can hold an arbitrary number of *dockables* and is managed by an instance of class `CControl` from the Docking Frames Common library.

Each space view as well as the tile info view were placed inside `CDockable` instances that can be docked in the docking area. The space view's summary and legend views which were previously opened in external windows were turned into ordinary panels and placed inside dockables as well. In addition, most other windows such as general information about a trace, event counters and timer info are now also dockable.

In the course of the transition, the default layout of the main window changed as well. Figure 5.13 depicts the new main window with dockable views. It shows the same trace and displays the same data as Figure 5.10 on page 42. The space views now occupy the left two thirds of the docking area. The tile properties view (formerly named *tile info*) is in the column on the right and by default occupies the entire column. GCspy places additional views that are not shown by default, such as the legend and summary views shown in the figure, below the properties view. The user can rearrange all dockables at will and close them as well. Closed space views can be reopened from the new *Spaces* menu.

In retrospect, the transition to a flexible user interface of dockable views with Docking Frames turned out to be rather simple and straight-forward. For the most part, adoption of the framework only required wrapping the existing views into dockables.

5.8 User interface look and feel

GCspy was originally designed and implemented using the cross-platform *Metal* look and feel for Swing. The appearance and behavior of this look and feel is the same on all different platforms and environments such as Microsoft Windows or Sun Solaris [SwingLAF]. While this can be considered an advantage from a developer's point of view, it also means that GCspy did not integrate very well with these environments.

In order to improve integration with its environment, GCspy was modified to use the look and feel of the system it is running in. This change was not entirely unproblematic. GCspy has a widely used class `Factory` with methods mostly for creating user interface components. Some of these methods used certain fonts or font sizes that fit the Metal look and feel, but seemed out of place with other look and feels. Therefore, the methods of `Factory` were modified to be aware of the active look and feel and use its fonts and colors. In the process, several overly trivial methods or methods that did not actually create an object were removed from `Factory`.

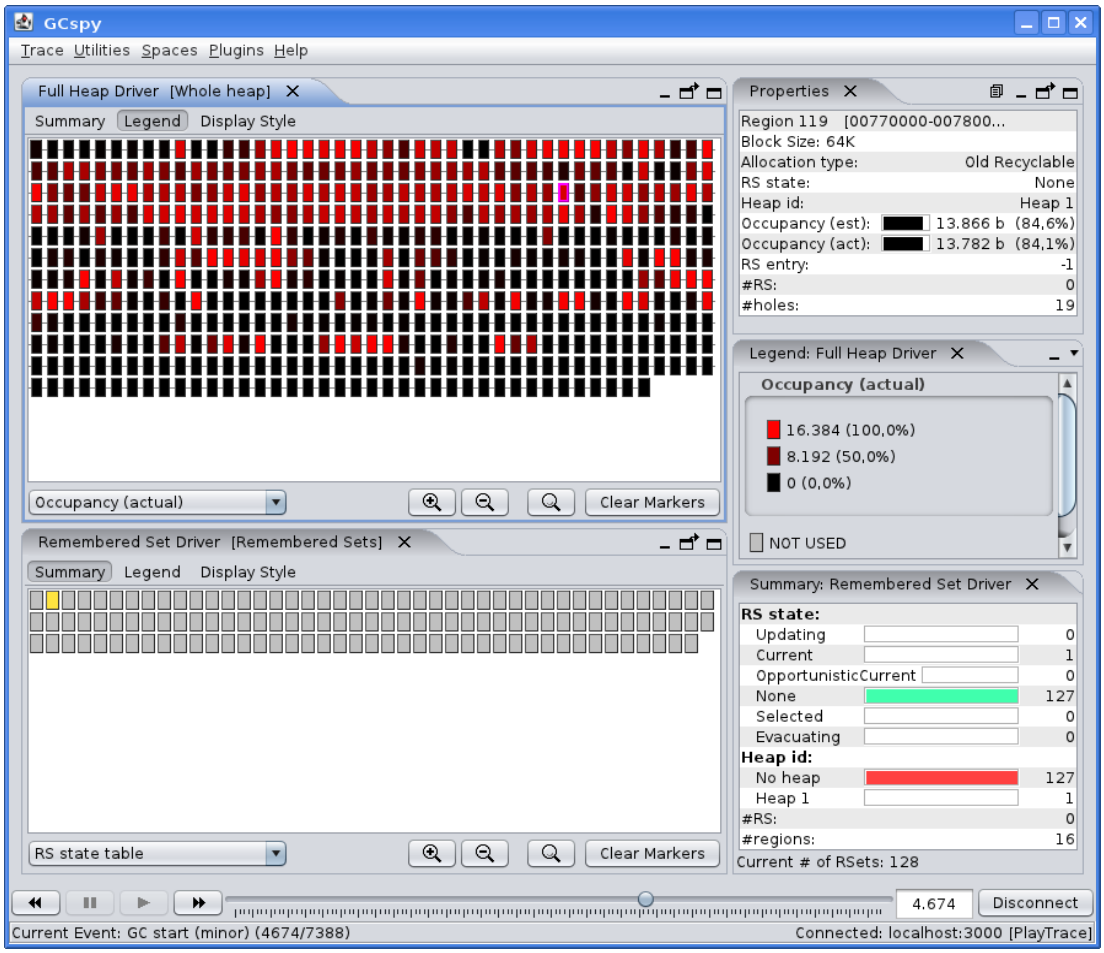

Figure 5.13: New main window with dockable views arranged in the default layout

5.9 Asynchronous event handling

In GCspy 1.0.12, when the client received a command from the server, the `ClientInterpreter` first deserialized the command and then immediately called all listeners subscribed for that type of command from its own main loop thread. The listeners then processed this event, typically by updating the part of the user interface they were responsible for. Afterwards, `ClientInterpreter` continued execution and resumed communication with the server.

There were two major problems with this synchronous approach. First, it led to unnecessary delays in the client when the server already sent another command that could be deserialised while the listeners still processed the current event. The server had to wait for the client as well when there were too many unprocessed commands. While this problem occurred less frequently with "real" servers, it did matter when replaying a trace from a file.

The second issue was that most of the listeners performed user interface updates directly from within the thread they were called in. However, most of the Swing user interface

code executes in a special thread called the *Event Dispatch Thread* (EDT). The majority of Swing components used in the GCspy client's user interface are not thread-safe and their methods must be also be invoked from this thread. Violations of this constraints may result in unpredictable errors that can be hard to reproduce [SwingConc].

As a first important step, the `UIInterpreterProxy` class was introduced even before this thesis. This class acts as a central hub that dispatches events from a `ClientInterpreter` to listeners in the user interface. `UIInterpreterProxy` and its super-interface `InterpreterEventProvider` are described in more detail in Section 4.3.2.

In order to resolve both described problems, `UIInterpreterProxy` was enhanced to use *runnables* together with Swing's event queue. `Runnable` is an interface that declares a single method `run()`. Realization of this interface implement this method to perform a particular task. `Runnable` is typically used to execute code across threads or in thread pools.

When `UIInterpreterProxy` receives an event from the `ClientInterpreter`, it creates a `Runnable` instance for each listener to dispatch the event to that listener. It then enqueues the `Runnable` instances in the user interface event queue using the `SwingUtilities.invokeLater()` method. At this point, `UIInterpreterProxy` already returns control to `ClientInterpreter`, which can immediately continue execution and resume communication.

The EDT on the other hand keeps processing events from the event queue and eventually takes the `Runnable` instances from the queue and invokes them. At this point, the listeners are notified of the event that occurred earlier and can update their user interface from the proper thread.

This new asynchronous mechanism has greatly reduced the delays in processing server commands, which is particularly noticeable when replaying a trace from a file. Listeners still receive event notifications in the correct order because of the first in, first out (FIFO) nature of the event queue. Correct usage of the EDT also prevents the GCspy client from random problems as a result of threading issues in the user interface.

5.10 Logging and error handling

Some of the components of the GCspy client reported debug information or error messages to the terminal. They either used the the `System.out` or `System.err` streams or GCspy's own `Verbose` class which provided elementary logging functionality.

However, apart from not being uniform, these approaches had some drawbacks. It was not possible to limit the level of detail or disable output entirely. The messages also did not have a consistent format or a time stamp. Finally, redirection of GCspy's output

into a file was problematic because some messages were sent to the standard output stream while others were sent to the standard error output.

Hence, these methods for output were replaced with actual logging using the *Java Logging API* which is part of the Java standard edition [JLogging]. The Java Logging API allows multiple named `Logger` objects to record log messages. Each log message has a *level* that is an indicator for the importance of the message and ranges from *severe* to *finest*.

The Java Logging API includes several types of *handlers* that can write log records to different destinations such as the terminal or into a file. Log messages can be filtered by source (e.g. the originating class or logger) or by their log level. A typical application is to hide fine-grained debugging messages that are not of interest to users. Users can configure filters and handlers without changing the source code by adapting the `logging.properties` file.

In the process of rewriting logging and error handling, several calls to `System.exit` were removed from exception handlers. Their purpose was to exit to program when severe errors occurred. However, they sometimes lead to unexpected termination on minor, recoverable errors.

Chapter 6

Conclusion and perspective

This thesis examined and described the architecture of the GCspy framework and the data structures it provides to abstract the heap layout. It also compared GCspy with other tools that offer similar functionality. The thesis further identified and illustrated the GCspy client's main use cases and functionality.

The thesis' technical documentation gave an overview over the implementation of GCspy. It described the layout of the source tree and identified and characterized GCspy's most relevant classes. The thesis then identified certain disadvantages and limitations in the original GCspy implementation and the means by which they were solved.

In summary, the GCspy client now provides considerably enhanced visualization capabilities and its modernized user interface offers a better user experience. Spaces can be used independent of each other and can be navigated by keyboard. New and improved tile visualization modes allow users to more accurately estimate values or recognize the type of a tile. Values of a tile can now be compared by visualizing multiple streams of a space at the same time. The new magnification allows users to conveniently inspect large spaces. Improved tile properties and space summary views visualize values with bar diagrams to allow quicker judgement of the data. Finally, the implementation of the docking views paradigm as well as improvements to the look and feel result in a generally more modern and robust user interface.

Future work on GCspy might revisit the plugin mechanism, where plugins currently need knowledge of internal data structures instead of using pre-defined interfaces. The GCspy client itself could be modularized and perhaps be built on one of the available rich client platforms. These typically provide a mature plugin framework and graphing libraries that would allow for even more extensive visualization functionality.

In conclusion, GCspy now provides a robust tool with rich visualization capabilities and a modern user interface to developers of memory management systems.

Appendix A

GCspy communication commands

The following is a complete list of the commands used in communication between client and server.

A.1 Client to server

Pause Request commands are sent to request suspending the execution of the application at the earliest convenience so no new events will be produced for the time being.

Restart commands are sent when the server is in paused state to continue execution.

Play One commands in paused state request to continue execution until the next event occurs, then return to the paused state.

Shutdown Requests are for initiating the end of the connection.

Event Filter commands modify the event filters in place. Event filters are used to disable or delay transmission of certain events to the client.

A.2 Server to client

Pause commands (as opposed to *pause requests* by the client) are sent by the server when it enters a paused state.

Shutdown is sent by the server to end its connection with the client (typically after a shutdown request by the client).

Stream commands are sent when the data of the stream of a space has changed and include the updated data.

Event commands indicate that an event occured and include the type of the event and timing information.

Control commands indicate that control information of a space has changed (such as the placement of separators or links) and include the new control data.

Event Count commands are used to update the event count at the client.

Summary commands are sent when the summary information of a space's stream changes and include the updated summary information.

Space Info commands indicate that the description of a space has changed and include the new description.

Space commands are sent when a space changes in its entirety. The new space information is included with the command.

List of abbreviations

API Application Program Interface
BSD Berkeley Software Distribution
EDT Event Dispatch Thread (Swing)
FIFO First In, First Out
GC Garbage collection, garbage collector
GNU GNU's Not UNIX, The GNU Project (free software project)
GPL GNU General Public License
HSB Hue, Saturation, Brightness (alias for HSV)
HSV Hue, Saturation, Value (color model)
IDE Integrated Development Environment
IP Internet Protocol
JAI Java Advanced Imaging (API)
JAR Java Archive
JDK (Sun) Java Development Kit
JMX Java Management Extensions
LGPL GNU Lesser General Public License
MVC Model-View-Controller (architectural pattern)
RCP Rich Client Platform
RVM Research Virtual Machine (Jikes RVM)
SDK Software Development Kit
TCP Transmission Control Protocol
UI User Interface
VM Virtual Machine

List of Figures

Bibliography

[Baldwin05] J. Baldwin, B. Gallop, C. Matthews, *JamOLizer: Garbage Collection Visualizer in the JamVM*, 2005.

[Cheadle06] A. M. Cheadle, A. J. Field, J. W. Ayres, N. Dunn, R. A. Hayden, J. Nystrom-Persson, *Visualising Dynamic Memory Allocators*, Proceedings of the 5th international symposium on Memory management (ISMM '06), June 2006, pp. 115-125.

[DockingF] B. Sigg, *DockingFrames, an open source Java Swing docking framework.* `http://dock.javaforge.com/`. Retrieved on April 27, 2011.

[EclipsePP] The Eclipse Foundation, *Eclipse Platform Project.* `http://www.eclipse.org/platform/`. Retrieved on April 27, 2011.

[FlexDock] java.net, *Flexdock, Swing windowing and docking framework.* `http://java.net/projects/flexdock/`. Retrieved on April 27, 2011.

[GCspy] University of Kent School of Computing, *GCspy: visualizing the heap.* `http://www.cs.kent.ac.uk/projects/gc/gcspy/`. Retrieved on April 22, 2011.

[Hirsch07] G. Hirsch, *Swing/SWT Integration.* `http://www.eclipse.org/articles/article.php?file=Article-Swing-SWT-Integration/index.html`. Retrieved on April 27, 2011.

[IDockWin] NNL Technology AB InfoNode, *InfoNode Docking Windows: a pure Java Swing based docking windows framework.* `http://www.infonode.net/index.html?idw`. Retrieved on April 27, 2011.

[JAI] Sun Developer Network, *Java Advanced Imaging (JAI) API.* `http://java.sun.com/javase/technologies/desktop/media/jai/`. Retrieved on May 17, 2011.

[JList] Oracle Technology Network, *The Java™ Tutorials: How to Use Lists.* `http://java.sun.com/docs/books/tutorial/uiswing/components/list.html`. Retrieved on April 24, 2011.

[JLogging] Oracle Technology Network, *Java™ Logging Technology.* `http://download.oracle.com/javase/6/docs/technotes/guides/logging/`. Retrieved on February 8, 2011.

[JMX] Sun Developer Network, *Java Management Extensions (JMX) Technology.* `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/`. Retrieved on September 27, 2010.

[jvmstat] Sun Developer Network, *jvmstat: light weight performance and configuration instrumentation for the HotSpot JVM.* `http://java.sun.com/performance/jvmstat/`. Retrieved on September 27, 2010.

[Marion05] S. Marion, R. Jones, *GCspy port to SSCLI (Rotor)*, 2005.

[NetBeansRCP] Oracle Corporation, *The NetBeans Platform.* `http://platform.netbeans.org/`. Retrieved on April 27, 2011.

[Printezis02] T. Printezis, R. Jones, *GCspy: An Adaptable Heap Visualisation Framework*, Proceedings of OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications, November 2002, pp. 343-358.

[Printezis02b] T. Printezis, A. Garthwaite, *Visualising The Train Garbage Collector*, Proceedings of the 3rd international symposium on Memory management (ISSM '02), June 2002, pp. 50-63.

[SanaJDock] Sanaware, *Sanaware Java Docking.* `http://www.javadocking.com/`. Retrieved on April 27, 2011.

[Singh07] V. Singh, S. Ranu, *Extending GCspy for Jikes RVM*, March 2007.

[SwingConc] Oracle Technology Network, *The Java™ Tutorials, Lesson: Concurrency in Swing.* `http://java.sun.com/docs/books/tutorial/uiswing/concurrency/`. Retrieved on April 27, 2011.

[SwingLAF] Oracle Technology Network, *The Java™ Tutorials: How to Set the Look and Feel.* `http://java.sun.com/docs/books/tutorial/uiswing/lookandfeel/plaf.html`

[VisualGC] Sun Developer Network, *VisualGC: Visual Garbage Collection Monitoring Tool.* `http://java.sun.com/performance/jvmstat/`. Retrieved on August 15, 2010.

[VisualVM] java.net, *VisualVM: All-in-One Java Troubleshooting Tool.* `https://visualvm.dev.java.net/`. Retrieved on August 15, 2010.

[VLDocking] L. Chamontin, *VLDocking, the Swing Docking Framework.* `http://code.google.com/p/vldocking/`. Retrieved on April 27, 2011.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 22. August 2011

Peter Hofer