How to Write APIs That Will Stand the Test of Time

Jaroslav Tulach

Sun Microsystems
http://www.netbeans.org

Design To Last

First version is always easy

Learn why to strive for good API design and few tricks how to do it from guys who maintain NetBeans framework APIs for more than ten years.

Agenda

Why create an API at all?

What is an API?

API Design Patterns

API Design Anti-patterns

Q&A

Distributed Development

- There is a lot of Open Source Solutions
  - ant, jalopy, velocity, tomcat, javacc, junit
- Applications are no longer written, but composed
  - Linux Distributions, Mac OS X
- Source code spread around the world
- Exact schedule is impossible

Presence of Computer Science

Enormous building blocks

Applications are assembled

http://www.cs.utexas.edu/users/wcook/Drafts/2006/RinardOOPSLA06.pdf

Modular Applications

- Composed from smaller chunks
  - Separate teams, schedule, life-cycle
- Dependency management
  - Specification Version 1.34.8
  - Implementation Version Build20050611
  - Dependencies chunk-name1 >= 1.32
- RPM packagers
- Execution containers like NetBeans

What is an API?

API is used for communication

- build trust, clearly describe plans

Evolution is necessary

- method and field signatures
- files and their content
- environment variables
- protocols
- behavior
- L10N messages

http://openide.netbeans.org/tutorial/api-design.html

Preservation of Investments

- Backward compatibility
  - source vs. binary vs. cooperation
- Knowing your clients is not possible
- Incremental Improvements
- First version is never perfect
- Coexistence with other versions

http://openide.netbeans.org/tutorial/api-design.html

Rules for Successful API design

- Use case driven API design
  - use cases -> scenarios -> javadoc
- Consistent API design
  - An interface that is predictable serves better than one which is locally optimal but inconsistent across the whole set.
- Simple and clean API design
  - less is more - expose only necessary functionality
- Think about future evolution
  - First version is not going to be perfect

Stability of APIs

- It is all about communication
- APIs can serve different purposes
  - Early adopters
  - Internal communications
  - Framework APIs
- We have stability categories
  - Private, Friend
  - Under Development, Stable, Standard
  - Deprecated

http://openide.netbeans.org/tutorial/api-design.html#life

Evaluation of an API Quality

Customer centric – easy to use

Use cases, scenarios, javadoc

Future evolution

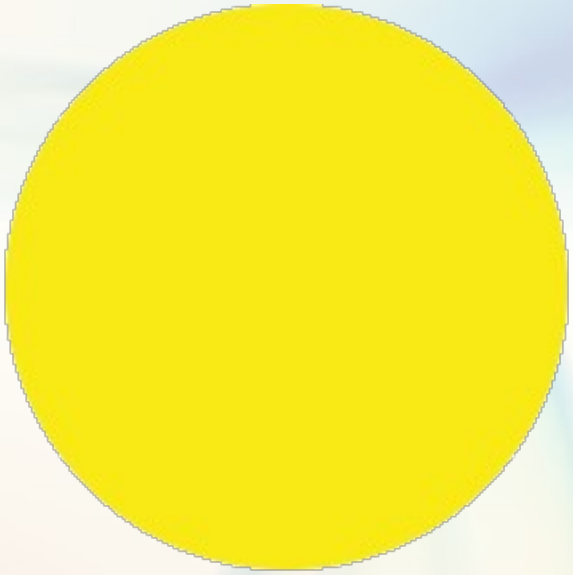
Test coverage

quality = code Δ specification

the "amoeba" model

NetBeans API Reviews http://openide.netbeans.org/tutorial/reviews/

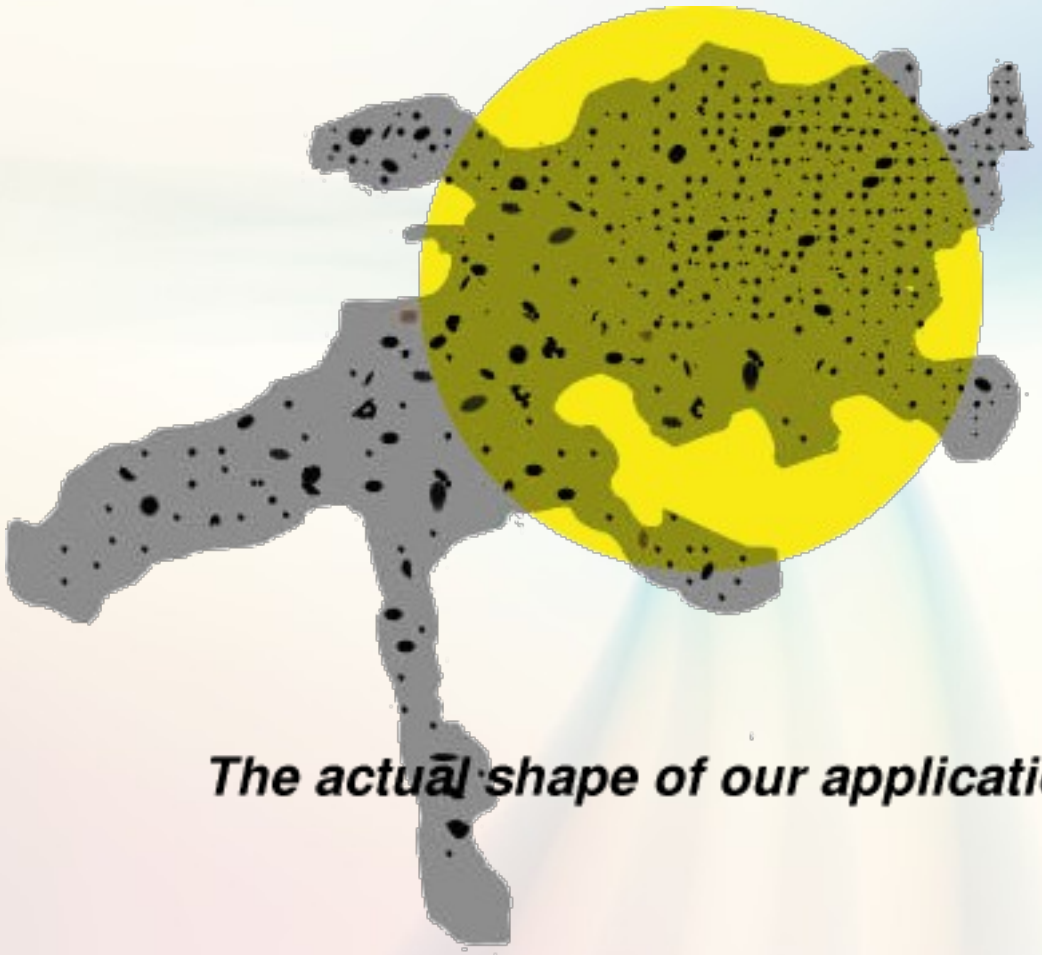
The Amoeba Model



How we think our application looks like

http://openide.netbeans.org/tutorial/test-patterns.html

The Amoeba Model



The actual shape of our application

http://openide.netbeans.org/tutorial/test-patterns.html

The Amoeba Model

Shape of amoeba after next release

http://openide.netbeans.org/tutorial/test-patterns.html

Design Patterns

- “Recurring solutions to software design problems”
  - common name
  - description of the problem
  - the solution and its consequences
- Simplify description of the architecture

http://openide.netbeans.org/tutorial/api-design.html

API Design Patterns

- Design Patterns as well
  - simplify description of the architecture
- API framework vs. internal design
- Main emphasis is on evolution
- First version is never perfect

http://openide.netbeans.org/tutorial/api-design.html

Factory Method Gives more Freedom

Do not expose more than you have to

```
// exposing constructor of a class like
ThreadPool pool = new GeneralThreadPool();
// gives you less freedom then
ThreadPool pool = ThreadPool.createGeneral();
```

The actual class can change in future

One can cache instances

Synchronization is possible

http://openide.netbeans.org/tutorial/api-design.html

Method is Better than Field

Do not expose more than you have to

```
class Person extends Identifiable {
  String name;
  public void setName(String n) {
    this.name = n;
  }
```

Synchronization is possible

Validation of input parameters in setter can be done

The method can be moved to super class

http://openide.netbeans.org/tutorial/api-design.html

Non-Public Packages

Do not expose more than you have to

```
OpenIDE-Module-Module: org.your.app/1
OpenIDE-Module-Public-Packages: org.your.api
OpenIDE-Module-Friends: org.your.otherapp/1
```

- NetBeans allows to specify list of public packages
- Enforced on ClassLoader level
- Possible to enumerate modules that can access them
- Split API classes into one package and hide the rest

http://openide.netbeans.org/tutorial/api-design.html

Separate Interface and Impl

Do not expose more than you have to

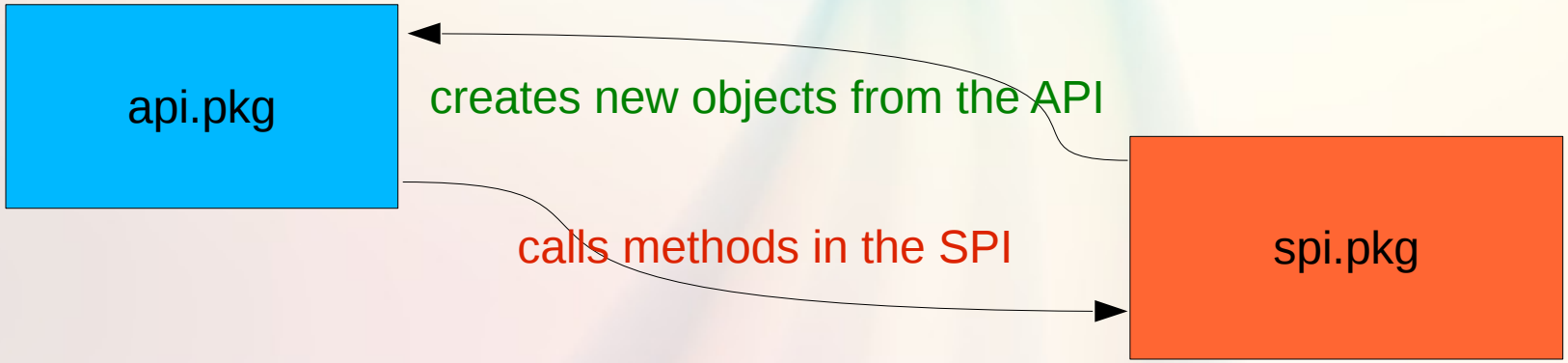
- Common advice in any design book
- Many ways to interpret the advice
  - Interface != Java interface keyword
  - Good API is not just a part of implementation
  - Honest use of API – do not cheat with impl
- Java class vs. interface battle
  - never ending ideological fights
  - pragmatic approach
    - method additions
    - access modifiers
    - subclassing and construction restrictions

http://openide.netbeans.org/tutorial/api-design.html

The Trampoline Pattern

Example of TeleInterface

Myth: There is no containment among packages
there is no way to create “friend” packages



Restrict Access To Friends

Do not expose more than you have to

Use package private classes

Java does not have friend packages, but...

```
public final class api.Item {
    /** Friend only constructor */
    Item(int value) { this.value = value; }
    /** API method(s) */
    public int getValue() { return value; }
    /** Friend only method */
    final void addListener(Listener l) { ... }
}
```

http://openide.netbeans.org/tutorial/api-design.html

The Trampoline Pattern cont.

Do not expose more than you have to

```
/** The friend package defines an accessor
 * interfaces and asks for its implementation
 */
public abstract class impl.Accessor {
  public static Accessor DEFAULT;
  static { Object o = api.Item.class; }

  public abstract Item newItem(int value);
  public abstract void addListener(
       Item item, Listener l);
}
```

http://openide.netbeans.org/tutorial/api-design.html

The Trampoline Pattern cont.

Do not expose more than you have to

```
class api.AccessorImpl extends impl.Accessor {
  public Item newItem(int value) {
    return new Item(value); }
  public void addListener(Item item, Listener l) {
    return item.addListener(l); }
}
public final class Item {
  static {
   impl.Accessor.DEFAULT = new api.AccessorImpl();
  }
}
```

http://openide.netbeans.org/tutorial/api-design.html

The Trampoline Pattern

Demo

The Difference Between Java and C

Separate client and provider API

Imagine API for control of media player in C

```
void xmms_pause();
void xmms_add_to_playlist(char *file);
```

Java version is nearly the same

```
class XMMS {
  public void pause();
  public void addToPlaylist(String file);
}
```

Adding new methods is possible and benefitial

Provider Contract in Java and C

Separate client and provider API

Now let's write the interface for playback plugin in C

```
// it takes pointer to a function f(char* data)
void xmms_register_playback((void)(f*)(char*));
```

Java version much cleaner

```
interface XMMS.Playback {
  public void playback(byte[] data);
}
```

Adding new methods breaks compatibility!

http://openide.netbeans.org/tutorial/api-design.html

Co-variance and Contra-variance

Separate client and provider API

Client API requirements are oposite to Provider API
Very different and complicated in C
Simple in object oriented languages
    Anything subclassable is de-facto provider API
Do not mix client and provider APIs.

The client API

The provider API

http://openide.netbeans.org/tutorial/api-design.html

New OutputStream method

Separate client and provider API

Can you add `write(ByteBuffer)` to OutputStream?

```
public void write(ByteBuffer b) throws IOException {
  throw new IOException("Not supported");
}
```

Previous version complicates clients, but there is a way:

```
public void write(ByteBuffer b) throws IOException {
  byte[] arr = new byte[b.capacity()];
  b.position(0).get(arr);
  write(arr);
}
```

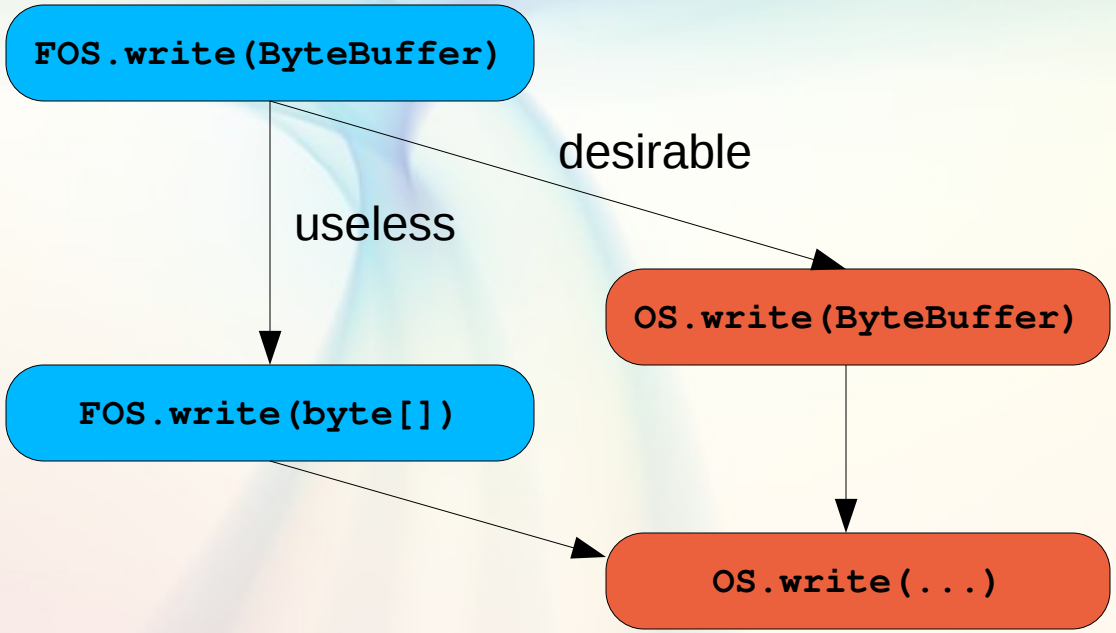
http://openide.netbeans.org/tutorial/api-design.html

The FilterOutputStream problem

Separate client and provider API

Shall `FilterOutputStream` delegate or call super?

```
public void write(ByteBuffer b) throws IOException {
  out.write(b); // super.write(b);?
}
```



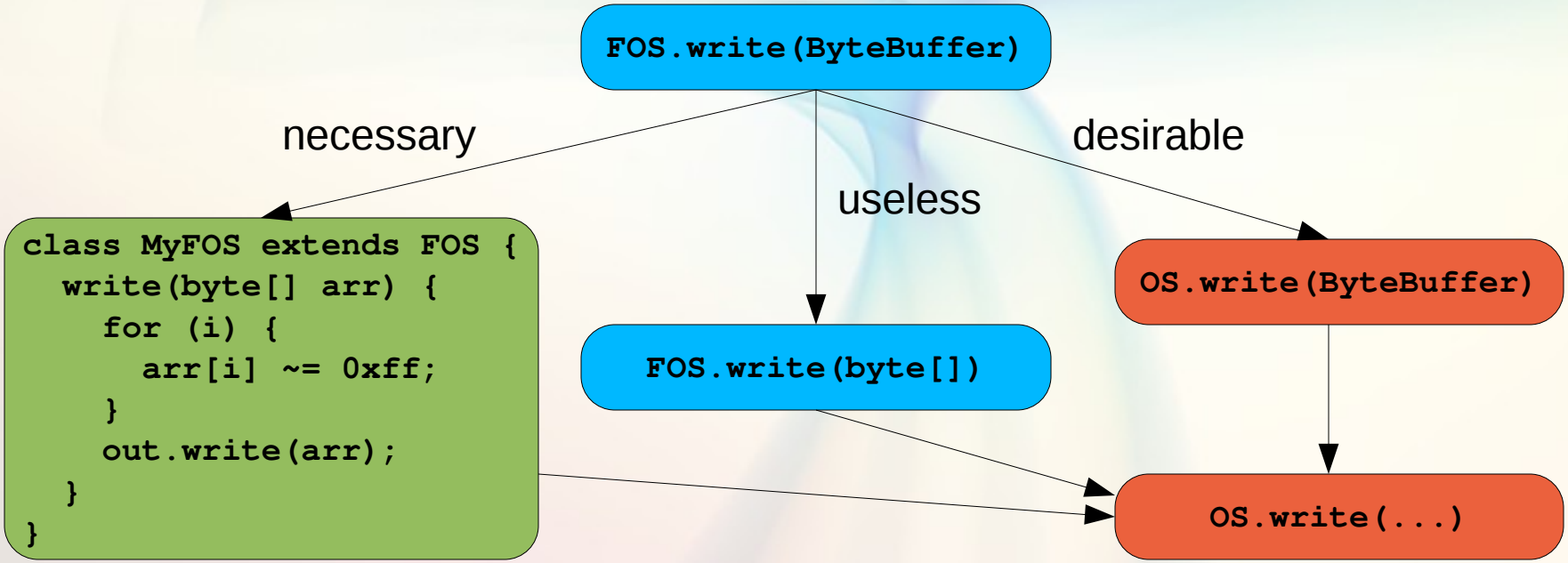
http://openide.netbeans.org/tutorial/api-design.html

The FilterOutputStream problem

Separate client and provider API

Shall `FilterOutputStream` delegate or call super?

```
public void write(ByteBuffer b) throws IOException {
  out.write(b); // super.write(b);?
}
```



http://openide.netbeans.org/tutorial/api-design.html

Fixing FilterOutputStream problem

Separate client and provider API

Fixing existing problem

Delegate iff **`FOS.write(ByteBuffer)`** is not overriden

Think about evolution during API design. For example:

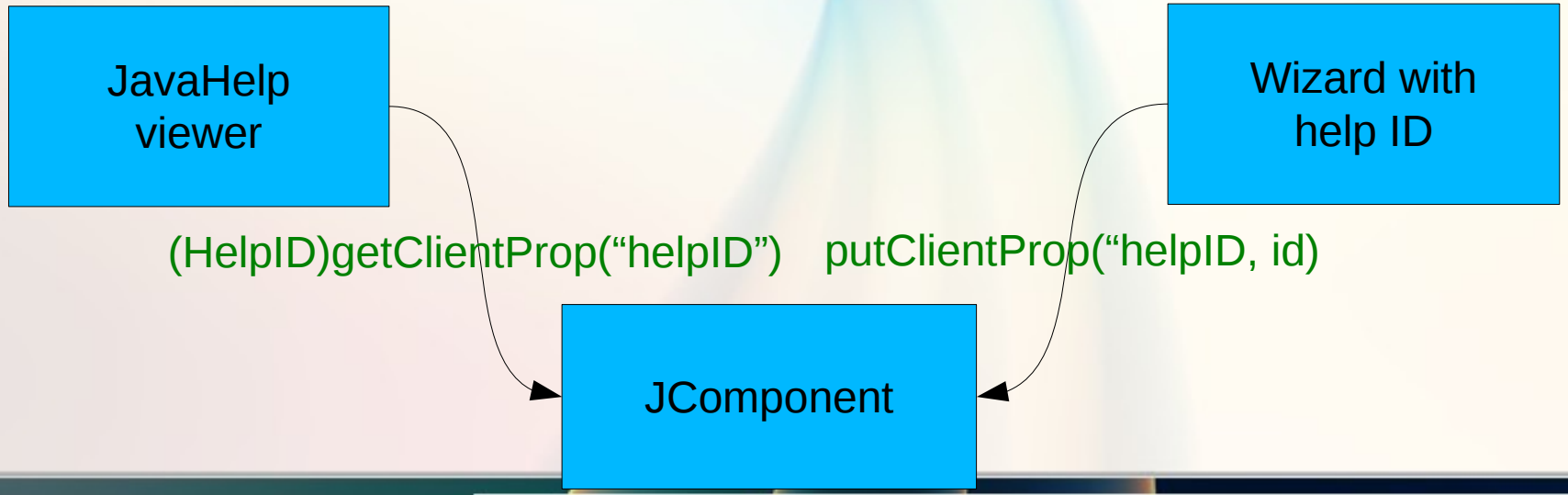
```
public /*final*/ class OutputStream extends Object {
  private Impl impl;
  public OutputStream(Impl i) { impl = i };
  public final void write(byte[] arr) { impl.write(arr); }
  public interface Impl {
    void write(byte[] arr);
  }
  public interface ImplWithBuffer extends Impl {
    void write(ByteBuffer arr);
  }
}
```

http://openide.netbeans.org/tutorial/api-design.html

Allowing for extensibility

Example of TeleInterface

Myth: By tunneling of data you lose type safety



Allowing for extensibility – tunnel data

```
package javax.swing;
public final class JComponent {
    public <T> T getCapability(Class<T> clazz) {
       return impl.lookup(clazz);
    }
}

package javax.help;
public interface HelpID {
    public void showHelp();
}

HelpID id = logicalWindow.getCapability(HelpID.class);
if (id != null) id.showHelp();
```

Allowing for extensibility – Lookup

```
package javax.swing;
public final class JComponent {
    public <T> T getCapability(Class<T> clazz) {
       return impl.lookup(clazz); // what is the impl?
    }
}
```

http://www.netbeans.org/download/6_0/javadoc/usecases.html#usecase-Lookup

Lookup

Demo

Foreign Code From Constructor

Anti Patterns

Accessing not fully initialized object is dangerous

- Fields not assigned
- Virtual methods work

`java.awt.Component` calls **`updateUI`**

`org.openide.loaders.DataObject` calls **`register`**

Wrap with factories, make the constructors lightweight

http://openide.netbeans.org/tutorial/api-design.html

Foreign Code In Critical Section

Anti Patterns

Calling foreing code under lock leads to deadlocks

Sometimes hard to prevent

```
private HashSet allCreated = new HashSet ();
public synchronized JLabel createLabel () {
  JLabel l = new JLabel ();
  allCreated.add (l);
  return l;
}
```

`java.awt.Component` grebs AWT tree lock

`HashSet.add` calls `Object.equals`

http://openide.netbeans.org/tutorial/api-design.html

Verification

- Mistakes happen
- Automatic testing of global aspects
  - Signature tests
  - Files layout
  - List of exported packages
  - Module dependencies
  - Automated tests
- Executed after each daily build

http://openide.netbeans.org/proposals/arch/clusters.html#verify

Summary

Be client centric

Be predictable

Always think about evolution

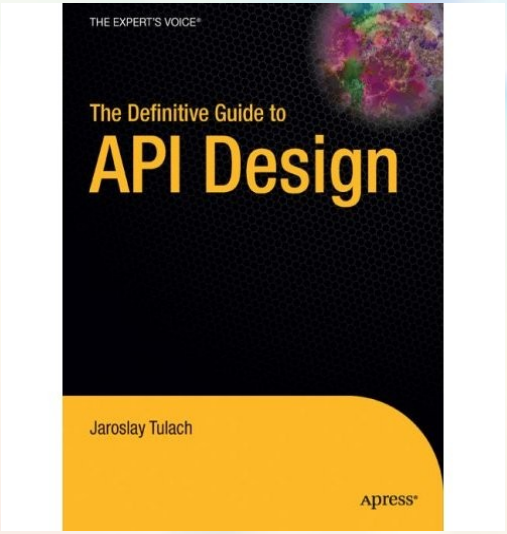
Design to last

Questions & Answers

Practical API Design

Confessions of a Java Framework Architect

ISBN-10: 1430209739



Jaroslav Tulach
http://www.netbeans.org