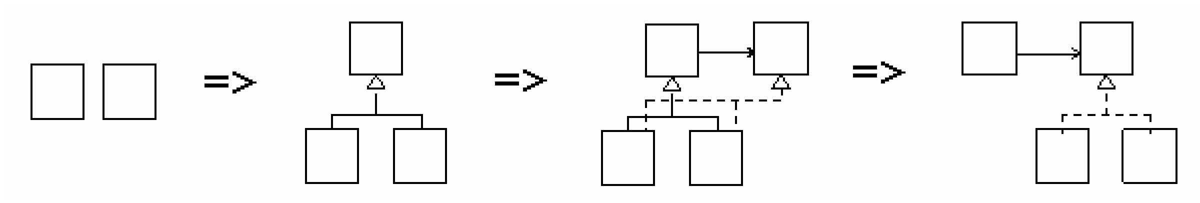
IjX: ImageJ refactored to interfaces for eXtensibility

or Evolving ImageJ to an Extensible Imaging Framework

A Proposal by Grant B. Harris, December 13, 2008.



ImageJ has exploded beyond an image processing application to become a platform used to construct many different tools. In an effort to sustain and grow the ImageJ Community, I am making this proposal to address some of the design issues I have become aware of from the mailing list, in conversation with other developers, and through my own efforts to design and develop plugins and applications that utilize ImageJ.

Many of the design issues I would like to address revolve around GUI issues. ImageJ plugins and custom applications cannot easily incorporate ImageJ components into alternative GUI toolkits (Swing or SWT), or into alternative desktop models (specifically MDIs such as JDesktop/JInternalFrame , tabbed or docking). Some have suggested rewriting ImageJ in Swing, for instance, but I think that would only get us stuck in a new box, as well as create incompatibilities with earlier versions. I'd like to suggest that a better approach would be to refactor ImageJ to interfaces classes.

"… useful abstractions are usually designed from the bottom up, i.e. they are ***discovered, not invented****… We create new general components by solving specific problems, and then recognizing that our solutions have potentially broader applicability."*

Please don't interpret this as being a negative criticism of the original design. Like all software, the design of ImageJ was concieved and has evolved based on the designer's assumptions about how it might be used. I suspect that Wayne Rasband has been surprised at the variety of unanticipated ways in which ImageJ has been used and abused over the years. And, the Java environment has changed and grown in many ways since he started on ImageJ. I offer this as a way of looking forward in recognition of Wayne's vision for ImageJ. His efforts have grown a vibrant community of users and developers. It is my hope that the ideas that I present here in some small ways afford the continued evolution of ImageJ as an open source imaging platform by making its architecture more extensible.

"In systems architecture, extensibility means the system is designed to include hooks and mechanisms for expanding/ enhancing the system with new capabilities without having to make major changes to the system infrastructure. A good architecture provides the design principles to ensure this—a roadmap for that portion of the road yet to be built. Note that this usually means that capabilities and mechanisms must be built into the final delivery which will not be used in that delivery and, indeed, may never be used. These excess capabilities are not frills, but are necessary for maintainability and for avoiding early obsolescence."

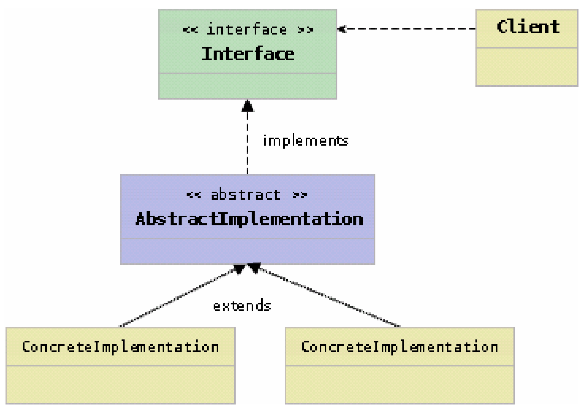
Why Refactor?

In essence, the goal of refactoring is to improve the software design, altering the structure so as to open up the design to new features and extensions, while maintaining compatability with the existing code base, extensions, etc.

"Refactoring is a technique for restructuring an existing body of code, altering its internal structure without changing its external behavior… It starts with software that currently works but is not well suited to an enhancement you wish to make. Refactoring is the controlled process of altering the existing software so it's design is the way you want it now, rather than the way you wanted it then. It does this by applying a series of particular code transformations, each of which are called refactorings." – Fowler

Toward an Interface-Oriented Design

"Program to an interface, not an implementation"



After 6 years of programming in Java (following 20 years of programming in various procedural languages), I am beginning to understand why interface-oriented design is valuable. (So I'm a little slow…) Having looked at only-the-gods-know-how-many APIs, toolkits, frameworks and libraries, I've begun to recognize what makes a good design, and to understand the design issues I have been struggling with. Until recently, one of the things I lacked was an appreciation of how interfaces and abstract classes enable the design of more general and reusable design solutions.

"Using the abstract interfaces of objects, polymorphism can be used to create extensible and loosely-coupled programs. The benefit of this is that if new types are added, and they adhere to the common interface specified, then their impact on changing the system will be minimal… When using polymorphism, we are able to adhere to this concept, and abstract out implementations in such a way our system's interface will not change when implementation requirements do. Once our system depends on interfaces only, we are decoupled from the implementation. The implementations can vary while our interfaces remain the same. This promotes flexibility." 1

Interface-Oriented Design

Software designed with interfaces and abstract classes allows for future changes and extensions because classes are more loosely coupled.

The essential ImageJ user interface-related classes *extend* specific `java.awt` GUI components. As a result, ImageJ's class hierarchy is directly coupled to the `java.awt` hierarchy. A class can only extend one superclass, as is the case of `ij.gui.ImageWindow`:

```
public class ImageWindow extends Frame
   implements FocusListener, WindowListener, WindowStateListener, MouseWheelListener,
```

[[Notes: This precludes multiple interface inheritance and …but a class can implement many interfaces.On designing to interfaces… coupling = implementation dependencies; decouple using interfaces… , polymorphism enables substitution; the implementation of interfaces in Java is similar to multiple inheritance; provide affordances (Affordances describe possible as opposed to intended usage [Norman1989])]]

Applying this insight to ImageJ, appears that by 'abstracting out' a number of *interfaces*, it is possible to make it more flexible and extensible. Make it GUI toolkit independent… Abstract out the methods that make an ImageJ frame or window different from an awt.Frame.

---

[1] http://www.muellerdesigns.net/dasblog/2006/05/29/WhatIsPolymorphism.aspx

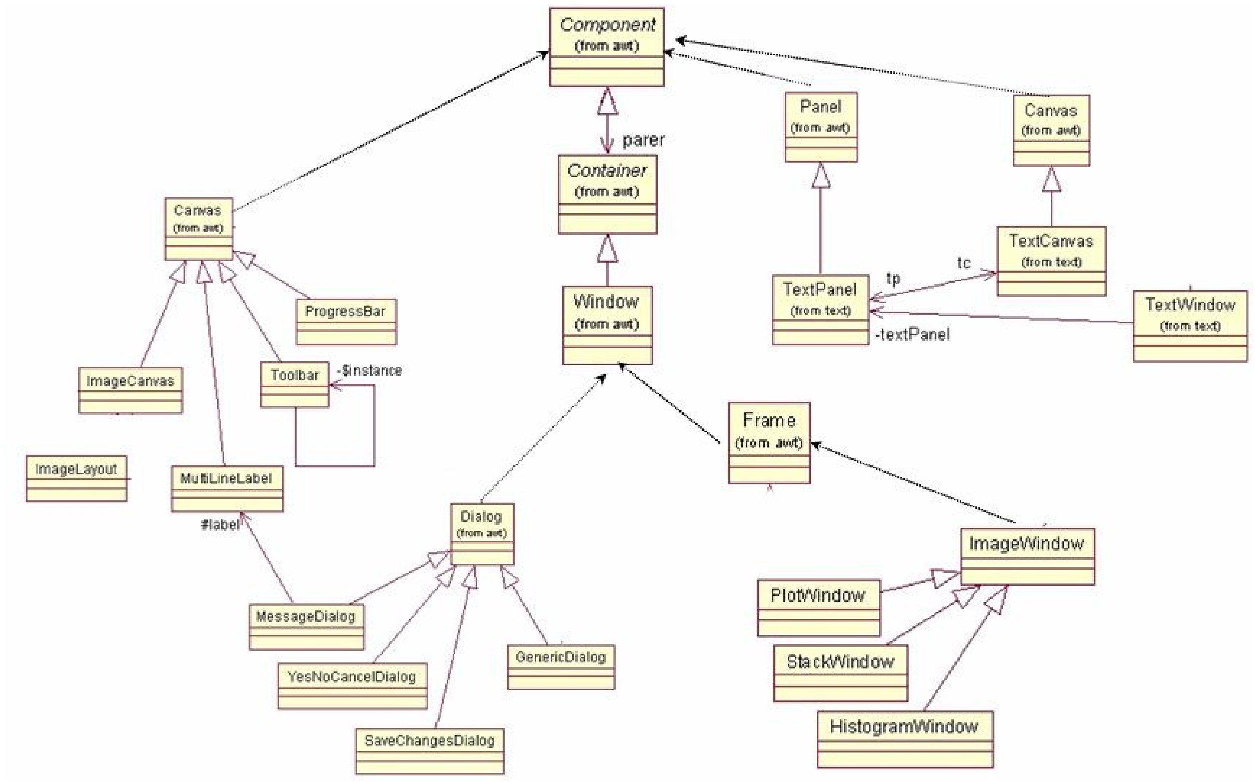


Figure 1 GUI-related Classes in ImageJ

I've been exploring ways to build a class hierarchy that would decouple ImageJ from AWT (or any specific GUI). The essential ImageJ-specific functionality of the ij.gui.ImageWindow class could be incorporated into an interface:

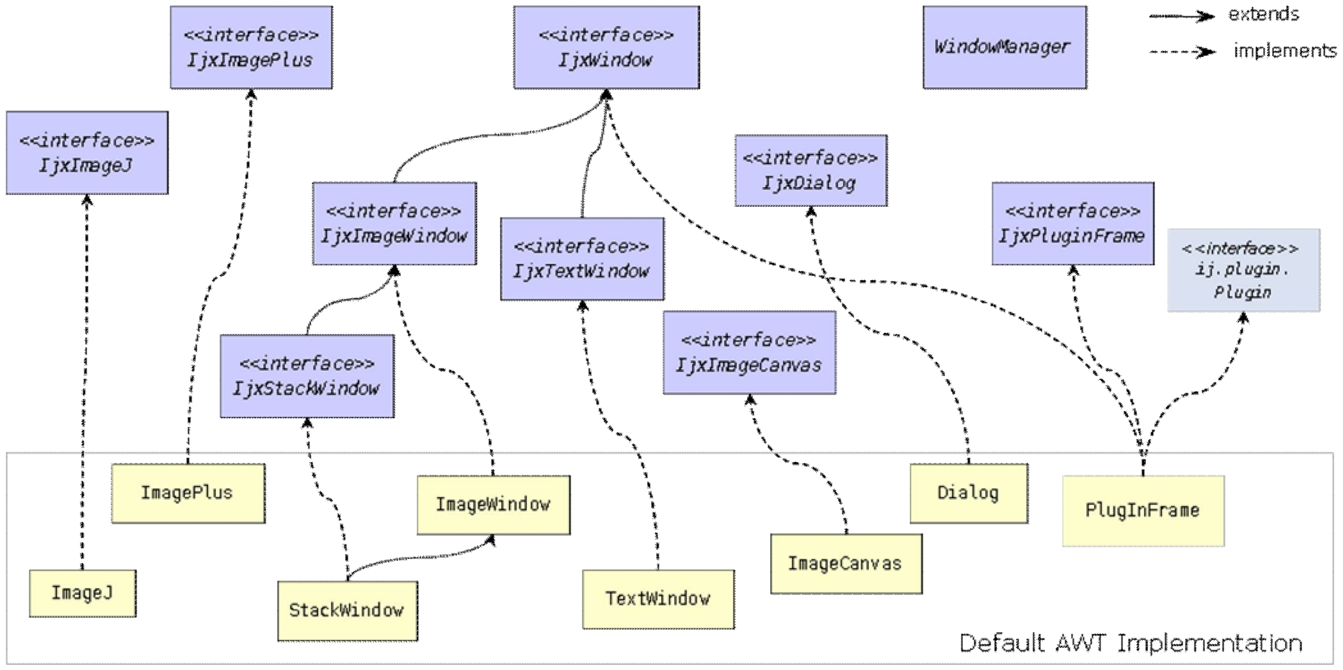

<<interface>> IjxImagePlus
<<interface>> IjxWindow
WindowManager
extends
implements
<<interface>> IjxImageJ
<<interface>> IjxDialog
<<interface>> IjxImageWindow
<<interface>> IjxTextWindow
<<interface>> IjxPluginFrame
<<interface>> ij.plugin.Plugin
<<interface>> IjxStackWindow
<<interface>> IjxImageCanvas
ImagePlus
ImageWindow
Dialog
PlugInFrame
ImageJ
StackWindow
TextWindow
ImageCanvas
Default AWT Implementation

Figure 2 Proposed ImageJX Interface Hierarchy

Possible Implementations:

- AWT/SDI (default)
- Swing/SDI
- Swing/MDI
- Swing/Docking

Having a generalized ImageJ window, <IjxWindow>, we can implement it with a class that extends the appropriate GUI component for a specific desktop model and toolkit.

```
public interface IjxWindow { }
public interface IjxTextWindow extends IjxWindow{ }
public interface IjxImageWindow extends IjxWindow{ }
```

Then, a conventional ImageWindow would look like:

```
public class ImageWindow extends java.awt.Frame implements IjxImageWindow {
…}
```

And a Swing-based ImageWindow might look like:

```
public class ImageWindowX extends javax.swing.JFrame implements IjxImageWindow {
…}
```

Or you might implement them as JPanels so you could put them into a docking framework:

```
public class ImagePanel extends javax.swing.JPanel implements IjxImageWindow {
…}
```

Default Implemention for AWT

A default implementation based on the java.awt toolkit…

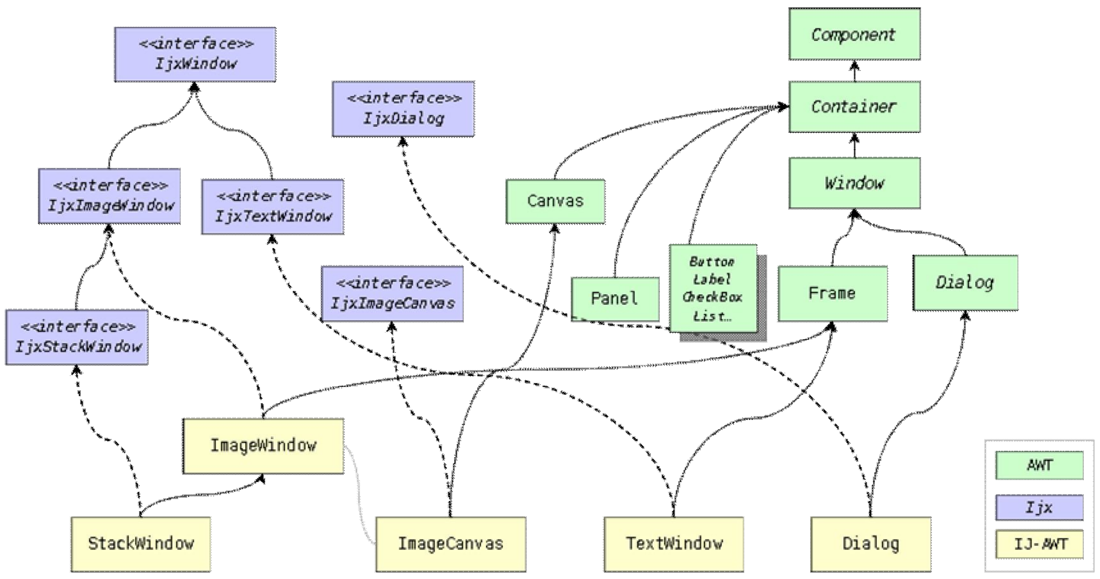


Figure 3 AWT Implementation

Implemention for Swing / SDI

An implementation based on the javax.swing toolkit, with an SDI desktop model…

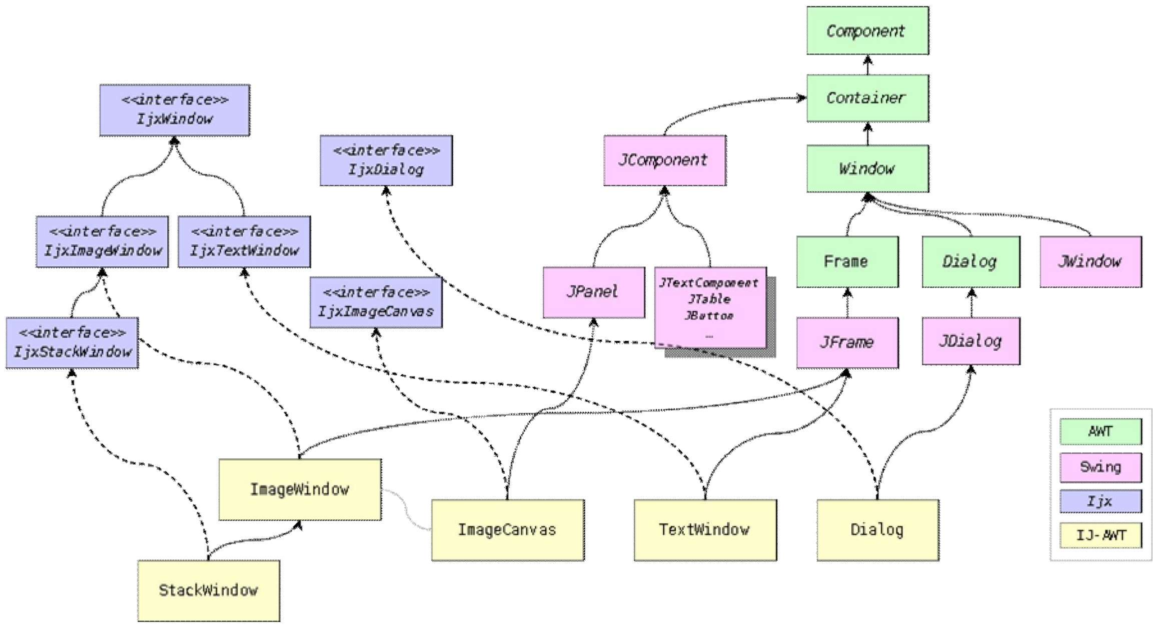


Figure 4 Swing / SDI Implementation

An Interface for Plugins?

[[a richer API for Plugins – existing are ad hoc, not interoperable.

Use of dependency injection?]]

The ways ImageJ is used…

- as an application with plugins
- as a library of components and image processing functions
- as a embedded application

Some Objectives

With IjX, I am hoping that some of the following will become possible:

- Support both the AWT and Swing GUI toolkits (and perhaps others)
- Enable rich integration of ImageJ components into custom applications
- Enable a variety of desktop models (SDI / MDI / Tabbed /Docking)
- Expand the possible types of plugins and extensions
- Take advantage of new Java language features in 1.5 & 1.6, including Generics, Annotations, Compiler API, Scripting API
- (+) Actions…
- Utilize other open source frameworks and libraries, including SwingAppFramework, SwingX, Binding, Look & Feel

Some Constraints

At least the following constraints need to be adhered to:

- maintaining the elegance and simplicity that makes ImageJ so usable.
- maintain (at least *source*) compatibility with existing plugins
- maintain backwards compatibility for earlier JVMs (perhaps at least 1.3 or higher)
- provide the same or better performance

Example Implementations

Then, a conventional ImageWindow would look like:

```
public class ImageWindow extends java.awt.Frame implements IJImageWindow {
…}
```

And a Swing-based ImageWindow might look like:

```
public class ImageWindowX extends javax.swing.JFrame implements IJImageWindow {
…}
```

For docking framework, panels would typically be added to the desktop:

```
public class ImagePanel extends javax.swing.JPanel implements IJImageWindow {
…}
```

Examples

Examples demonstrating different implementations (SDI/MDI/Docking).

Appendix

Designing to Interfaces

Discovering the Necessary Interfaces…

"It takes a great deal of inspiration to construct a good class hierarchy."

*"These rules are based on the fact that **useful abstractions** are usually designed from the bottom up, i.e. they are **discovered, not invented**. We create new general components by solving specific problems, and then recognizing that our solutions have potentially broader applicability… Classes usually start out being application dependent. It is always worthwhile to examine a nearly-complete project to see if new abstract classes and frameworks can be discovered. They can probably be reused in later projects, and their presence in the current project will make later enhancements much easier. Thus, **creating abstract classes and frameworks is both a way of scavenging components for later reuse and a way of cleaning up a design**. The final class hierarchy is a description of how the system ought to have been designed, though it may bear little relation to the original design."* -- Designing Reusable Classes, 1988 (www.laputan.org/drc/drc.htm)

Some Design Links/References

Practical API Design: http://wiki.apidesign.org/wiki/Main_Page

Implementation Patterns by Kent Beck, Addison-Wesley Professional, 2008

Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks:
http://st-www.cs.uiuc.edu/~droberts/evolve.html

A Rule-Based Approach to Framework Evolution:
http://www.jot.fm/issues/issue_2006_01/article3

Interface Oriented Design

Interface-Oriented Design: http://www.pragprog.com/titles/kpiod/interface-oriented-design
(A chapter: http://www.developerdotstar.com/printable/mag/articles/pugh_interface_oriented.html)

Designing Reusable Classes: http://www.laputan.org/drc/drc.html

Polymorphism and Interfaces: http://www.artima.com/objectsandjava/webuscript/PolymorphismInterfaces1.html

Implementing Basic Design Patterns in Java: http://gee.cs.oswego.edu/dl/cpj/ifc.htm

Interfaces and Abstract Classes - the pillars of software design: http://javalive.com/modules/articles/article.php?id=15

Basic Design Principle: Loosely Couple: http://www.tattvum.com/Articles/2002/2002-03/2002-03-24/Ramu-SE-20011113-LooselyCouple.html

Why extends is evil - Improve your code by replacing concrete base classes with interfaces:
http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html

Effective use of Interfaces and Abstract Classes: http://javalive.com/modules/articles/article.php?id=17

Refactoring

Refactoring Thumbnails: http://www.refactoring.be/thumbnails.html

An introduction to Software Refactoring (Tom Mens)

Swing/AWT Issues

Problems with using Swing and AWT together can be managed. These issues have to do with the mixing lightweight and heavyweight components and painting differences.

See http://java.sun.com/products/jfc/tsc/articles/mixing/ and http://java.sun.com/products/jfc/tsc/articles/painting/.

Desktop Models

SDI, MDI, Tabbed/Docking

User Interface Design for Business Applications:
http://richnewman.wordpress.com/2007/10/26/user-interface-design-for-business-applications/

Damien Farrell's MDI version of ImageJ:
http://homepage.ntlworld.com/jfarrell/imagejmdi/

An interesting approach that generally address the SDI/MDI conundrum here: (It allows you to switch from a JInternalFrame to a JFrame in the same application)
http://www2.sys-con.com/ITSG/virtualcd/java/archives/0901/brett/index.html

Understanding Containers:
http://java.sun.com/products/jfc/tsc/articles/containers/index.html