Fundamental Elements of an Extendible Java Framework

Their are many views of what a framework is. Those who have studied them closely and have written about them in an academic sense seem to center their definition around the concept of abstract classes (see Johnson). Many who market them to the industry tend to focus on functionality and APIs. The Java programming language offers a series of underpinnings that make one reconsider the wording of the earlier definitions. When those underpinnings are exploited in the development of a framework, one can get a level of extendability and understandability in a framework which is a step above what one might get in the more established object-oriented programming languages (such as C++ and Smalltalk).

We will discuss the patterns we have used in developing frameworks in Java that encourage extendible and understandable frameworks. While doing this we do not intend to take part in the battles over how much more productive or powerful an environment Smalltalk is, how much more efficient a language like C++ is, etc. We are merely attempting to illustrate how one should approach building a framework in Java to meet these two goals of extendibility and understandability.

There are many patterns in this paper that are tied together in many ways. We are still struggling to figure the best way to help our readers see the big picture. Here are a couple of attempts:

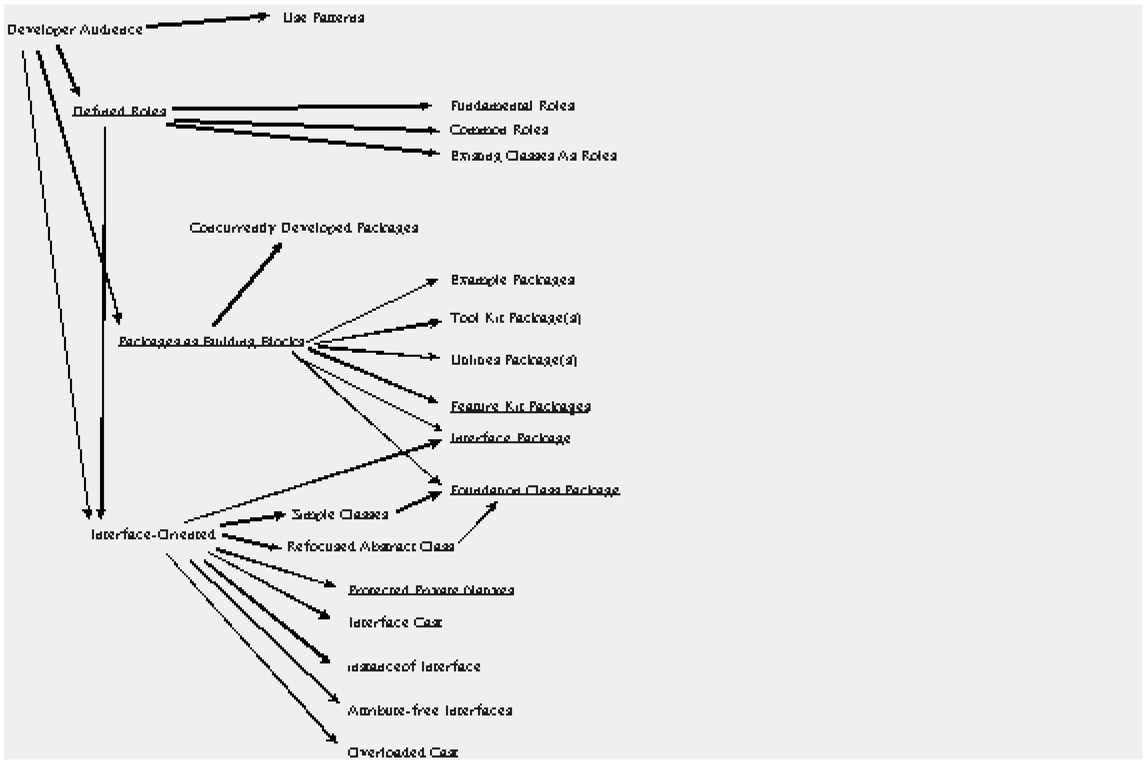

Figure 1: Many Patterns Lead to Other Patterns

One could also read them with the following understanding of the general categories:

General Principles

- Developer Audience

Although we certainly believe that one could use a subset of these patterns without undesirable consequences, we strongly believe that these patterns should be used together to get the most significant leverage in a framework.

Note: This is a work in progress. The raw material of this is taken from a PowerPoint presentation of the same name which is not in pattern form although much of the spirit of patterns (not just the problem and solutions but the whys and the examples) are contained therein. I'm working on moving it into pattern form but got a late start. Due to personal time constraints and focus of "10 pages" for PLoP, I've chosen to ignore the details of the coding techniques patterns, letting them just sit as placeholders until I get feedback on the rest…I might actually have several groups of patterns here. The focus on roles and packages is the heart of what I will be focusing on for PLoP. I'd like to get feedback as to whether they should stand alone and/or whether I should finish the rest of the stuff I put around it.

Developer Audience

Problem

There are so many definitions and sub-definitions of frameworks. These definitions typically assume something about the customer/audience for the framework. When the need for a framework is determined, how should one choose the audience?

Forces

- "In theory, there is no difference between theory and practice; in practice, however, there is" - Brian Neal. The theories about Black-Box, White-Box, Gray-Box are basically ways to categorize frameworks once they are built and they don't always fit so cleanly into a category
- Just about all frameworks are surrounded by people in roles which would like to look at it as a black-box (understand and use the interface, don't care about how it works), white-box (understand the details of how it works to maintain, improve, extend it), or gray-box (don't want to know all the details).
- The only thing developers hate more than long meetings and having to read volumes of documentation is creating the documentation in the first place. They want to code.
- Good developers accept the fact that they need to understand what they are working with before hacking away. However, they are quickly discouraged by being forced to read irrelevant details that do not contribute to the understanding of the particular problem they are attempting to tackle. They want to code.
- Gray-box and Black-box frameworks are derived from white box frameworks via documentation and packaging by people who understand the details well enough to package it. These are its developers. Developers do not want to write prose. They want to code.
- Often, reverse engineering the details of a black-box or gray-box framework from documentation which is not targeted at framework developers is not simple.

Solution

When building a framework, assume there are people who will come after you which would like to look at it as a white-box, with all of the details exposed but who don't want to read a voluminous manual in order to gain the exposure. With that in mind, structure your code and documentation with these people in mind. If this is done, the person who comes behind you will not curse your mother's grave. Realize there is often a strong possibility that the person coming after you will be you.

In order to make the documentation more approachable to developers and achieve the flexible goals of a white-box framework, you will want to Use Patterns. You should be careful in identifying Defined Roles and then write Interface-Oriented Code around these roles to make it extensible and understandable. You will also want to use Packages as Building Blocks to help organize the code to reflect these roles and to allow the developer audience to easily identify components. All of these things will reduce the necessary amount of textual documentation that is difficult to keep in sync with the reality of the framework, and is about the last thing a developer wants to do.

Side benefits of this approach includes many of those found in Coplien's Development Process Patterns such as Developer Controls Process and Architect Also Implements. One could argue that following these patterns might lead you to this conclusion (which is the chicken, which is the egg?).

Examples

Ralph Johnson once stated "the easiest way to document a framework is with commented code". JavaDoc attempts to do just that. On the other hand, following agreed upon standards and patterns has proven to drastically reduce the need for commented code in situations (such as eXtreme Programming) where adherence to those standards are encouraged and used. Both approaches assume a developer audience.

HotDraw has been a very popular framework for a variety of reasons. The openness of its source code is just one. There are at least two implementations in Java which have been studied by many developers. Others who had to write specific applications have used it as their starting point and have benefited greatly from having the source code available and would not have done so without it. Smalltalk versions of it have been used as a black box basis for commercial tools (e.g. Object Explorer, from First Class Software).

Many developers of Java frameworks (e.g. IFC, JFC, etc.) purchased source code licenses for the JDK in order to better understand how to build on top of it.

Use Patterns

Problem

How do I build my framework in such a way as to minimize documentation and allow for flexibility as the framework is used in a variety of ways?

Forces

- Frameworks, by their very nature are expected to be used in a variety of contexts. Frameworks are often built to handle a series of related complex problems. This may encourage a lot of documentation to explain how the framework achieves these goals.
- Good frameworks aren't really built, but they evolve (see Johnson & Roberts), therefore it is important that flexibility be thought of as early as possible and consistency be encoded and maintained in the framework (or potential framework) from start to finish.
- Flexibility often means a level of abstraction must be encoded that is very hard to describe to people without a common vocabulary with which to do it.
- Developers are turned off by shelf-feet of documentation.
- As frameworks evolve, there must be cooperation between their users and their developers or the typical goals of building the framework (e.g. total cost savings, scalability, maintainability, leverage from changes) will never be realized. This requires efficient communication in order to allow for time where the actual work occurs.

Solution

Use patterns which capture best practice in meeting your goals whenever possible. Organizational patterns may help you structure your team(s) well. Many design patterns have been mined from successful frameworks and have flexibility as one of their primary motivations. Implementation (or coding) patterns provide for a consistent style and cause less surprises when looking at code for the first (or Nth) time. Lastly, patterns have proven to provide significant leverage in both documents illustrating the use of a framework, and the design of a framework.

Examples

HotDraw has made pervasive use of design patterns (e.g. those found in GOF) and implementation patterns (derived from Smalltalk Best Practice Patterns, Self-Encapsulation, and Doug Lea's Java Coding Standards). There was at least one version in Java implemented by a Solo Virtuoso who has also worked on many successful pre-Java frameworks which followed many other organizational patterns such as

Architect Also Implements. A Smalltalk version of HotDraw was successfully documented with patterns in 1992, and many have followed this example. The author has successfully documented the design of several frameworks leveraging patterns throughout to significantly reduce the total new documentation for the framework which needed to be maintained as the framework evolved to a manageable size (e.g. a Database Broker framework, and an GUI Application Framework). Even in a framework which has no design documentation, the author's HotDraw for Java was explained to another developer with the aid of the use of patterns in a few hours… both parties agreed this could not have been so effectively and efficiently communicated without reference to the patterns.

Defined Roles

Problem

How do you begin to determine what objects are important in providing functionality in a framework?

Forces

- Classes have been described by many as the basic unit of object-oriented design.
- There have been many concepts added to OO methods and languages that introduce complexity when it is found that objects play multiple roles.
- Some have tried to reduce classes to data types and have missed the entire responsibility aspect of objects.
- Many have claimed that objects can be used to communicate from developers to non-developers and at many levels. However, when complex implementation details impact the details of objects, communication with non-experienced developers is lost before the implementation level of detail is relevant.

Solution

Define the signficant roles in the system before determining which objects fulfill them. Throughout implementation, focus on the roles objects are playing and define roles that weren't discovered during higher level activities. Roles map better to use cases and tends to be a better means of communication for many activities with a wider variety of participants than just developers. Additionally, it is easier to focus on the purpose of individual objects when their roles are clearly defined. There are many experienced OO developers (see Roles Before Objects) that are promoting roles as "what gets defined before implemented objects".

It is great when you can identify the significant roles played in the framework and document them. There should be significantly fewer roles than classes. If all the roles can be understood by the audience, the chance of the understandability of the entire framework is drastically increased as developers can place each class the come to in a mental model of the entire framework. If the entire framework can be described in 7+/-2 significant roles, you have gained a major victory. This allows people to more easily understand the entire framework as they can keep it all in their memory at once <need reference to 7+/-2>. In order to really use a framework it must be understood. There is a better chance of getting "7+/-2" from roles than classes. Classes can be mapped to roles much more easily than the other way around. So, start with roles before going for a deeper level of understanding.

Defined Roles can be mapped to code in three basic ways: Fundamental Roles, Common Roles, and Existing Classes as Roles.

Interface-Oriented Code

Problem

Once roles have been defined, how do we preserve the separation of roles and implementation?

Forces

- Roles are concepts. Implementation is hard cold logic
- Multiple objects may be able to play the same roles.
- Framework developers are not omniscient or omnipresent. In frameworks, it is expected that each application employing a framework will define custom objects which fill the generic roles.
- When someone realizes they want another class to fulfill a particular role, he'd rather not have to modify other parts of the framework or even have to understand the rest of the framework at a level of depth to know if this new approach to fulfilling a role will impact it. The barrier to entry of a new object to fulfill a role should be rock bottom.
- Roles, although only concepts, imply a cohesive set of behaviors. Java method signatures imply atomic behavior. Java interfaces imply a cohesive set of atomic behaviors.
- Use of interfaces adds a bit of overhead that direct access to attributes don't add.
- Interfaces allow multiple players of same roles.
- Interfaces are very lightweight.

Solution

Define roles with interfaces which are basically a group of related method signatures which implies behavior. Individual classes which fulfill the roles may often vary within the framework. When new applications employ the framework there will be additional classes fulfilling the same roles. All code which refers only to interfaces allows any object which implements the interface to play the implied role without necessitating changes.

Always refer to interfaces when writing code rather than concrete (or even abstract) classes. The only time code can not be written referring to interfaces is at object creation time. This can typically be limited to a handful of methods that can easily be overridden by specialized subclasses and everything will work as advertised. You can get around limitations of Java by respecting Protected Private Natives, making Interface Casts, testing for necessary properties via instanceof Interface, using Attribute-free Interfaces, and avoiding false implications via Overloaded Casts

Taking this approach will lead to Refocused Abstract Classes. Examples of and access to useful classes that fulfill the roles implied by this interface-oriented approach can be created and found as Simple Classes. All of these increase the approachability of the framework to the developer.

A disadvantage of this is that interfaces may share the same namespace as classes. This can be helped by Interface Packages which provide their own namespace, but using full package names to identify classes and interfaces makes code more awkward to read.

In practice, a well-factored framework usually doesn't suffer noticeable performance problems due to the use of interfaces. Significant performance problems usually come from less optimal approaches to the problem at a higher level than language features. Some developers worry about performance issues too early and at the wrong level. They should "Make it run, Make it right, Make it fast" (see Lazy Optimization).

Refocused Abstract Class

Problem

If the framework is oriented around interfaces, what is the role of the abstract class which the literature points to as a fundamental unit of a framework?

Forces

- Traditional abstract classes are responsible for defining

1. The interface of a set of concrete classes
2. The types or roles in a system
3. Much of the concrete behavior implied by the abstract class

- Interfaces provide the first two responsibilities

Solution

Create an abstract class to provide as much concrete behavior as possible for concrete classes which implement a non-trivial Defined Role. Provide default behavior wrapped around a few critical methods whose implementation must be defined by its subclasses. Typically, you won't create these abstract classes until you find that you need at least two concrete classes which will implement the role (possibly while implementing Feature Kit Packages). However, there are times when foresight can lead to this. Often this occurs when developing Simple Classes. Abstract classes will be more common for Fundamental Roles than other Defined Roles.

Why the term "Refocused"? "Abstract Class" has been defined by previous literature (e.g. Johnson and Woolf). There is a lot this pattern leaves out about Abstract Classes that is in this other literature. This literature also assigns abstract classes all the responsibilities of interfaces. This was mostly due to the fact that previous mainstream OO languages, namely C++ and Smalltalk, did not provide a simple mechanism for separating these responsibilies. In conversations with the others of both of these named authors, they agree that Java's interfaces rightly encourage this separation of concerns.

Examples

In version 1.0 of HotDraw for Java, there is an interface which defines Figure. Figure defines 32 public protocols. BasicFigure is an abstract class which defines 46 methods, many protected. Only 3 of those methods are abstract and must be defined by subclasses to get meaningful, comprehensive behavior. The remaining methods rely directly or indirectly on those core 3 methods.

Fundamental Roles

Problem

During early design activities there are several roles that become obvious. How do you map these to the implementation of a framework?

Forces

- Roles can best be defined as interfaces.
- For most domains, there is a lingua franca used among non-developers that will show up during enumeration of use cases.

Solution

Define interfaces for each of the Fundamental Roles identified during exploration and early design activities. Realize that this interface will evolve, but the names of these interfaces will have a one to one mapping with concepts in the language of non-developers of the framework. Keeping these interfaces prominent will help developers and non-developers in their communication of critical concepts. In some cases these Fundamental Roles will be indistinguishable from either Common Roles or Existing Classes as Roles. In these situations, these should be explicitly identified when talking to non-developers as synonyms.

Sometimes when you have more than 7+/-2 Defined Roles, you may still only have 7+/-2 Fundamental Roles. By focusing on these as the primary roles (which they usually are) one may be able to fully understand the framework at a fundamental level.

Examples

HotDraw for Java has about seven fundamental roles: Figure, DrawingCanvas, DrawingStyle, Handle, Tool, Locator, and Rectangle. In addition there are a couple of specialized roles: LineFigure and PolygonFigure. All but two of these had an interface defined by name. At implementation time it was found that Tool was no different than the Common Role of EventHandler. It was also decided that the class Rectangle fell into the category of an Existing Class as Role.

Common Roles

Problem

There are often commonalities seen between objects during implementation that were missed completely during high-level design. When these are found, what should be done?

Forces

- Multiple inheritance is very difficult to maintain and resolve conflicts between inherited versions of messages with identical signatures.
- Objects often play multiple roles
- Single inheritance can lead to misleading assumptions about the fact that an object plays multiple roles.

Solution

As you find duplicate code acoss object groups, determine if there is some Common Role that can be named that these multiple objects are playing. For each one, create an interface, and then have each of the classes which play the role implement and adhere to it.

Existing Classes As Roles

Problem

The JDK and other libraries don't consistently take the same view of the role of interface. What do we do when we find classes that fulfill much of the responsibility but do not have a correponding interface.

Forces

- If part of the audience is future developers, confusing them by redefining terms they are used to can be counterproductive
- Redefining others classes, especially foundational ones, is srongly discouraged by Java's loading model.

Solution

When classes exist which accurately fulfill an identified role, and there is no obvious alternative implementations on the drawing board that are worth a wait, simply treat the existing class as you would an interface until such time as alternative implementation of the role is required.

Examples

HotDraw for Java uses Rectangle out of the box (java.lang.Rectangle) to fulfill the role of the rectangular boxes often used to determine the size, shape, etc. of a Figure.

Packages as Building Blocks

Problem

So you've created some interfaces, abstract classes, and concrete classes. How do we put them together in a form that meets the needs of those using the framework without unduly overloading them with information that might not be relevant?

Forces

- White box frameworks expose all the details of all the classes and interfaces being used.
- Until users become familiar with the framework, they will be spending time reading code.
- Unfocused reading often makes it difficult to glean information efficiently.

Solution

Divide the framework into several packages which will encourage the user to know exactly where to look for a prescibed context. Categorize and name the package so they fall into the following categories for easy identification: Interface Package, Foundation Package, Utility Package(s), Feature Kit Packages, Example Package(s), and optional Tool Kit Package(s). These packages should build cleanly on one another so that they can be easily replaced by alternate versions without impacting other packages. See Figure 2.

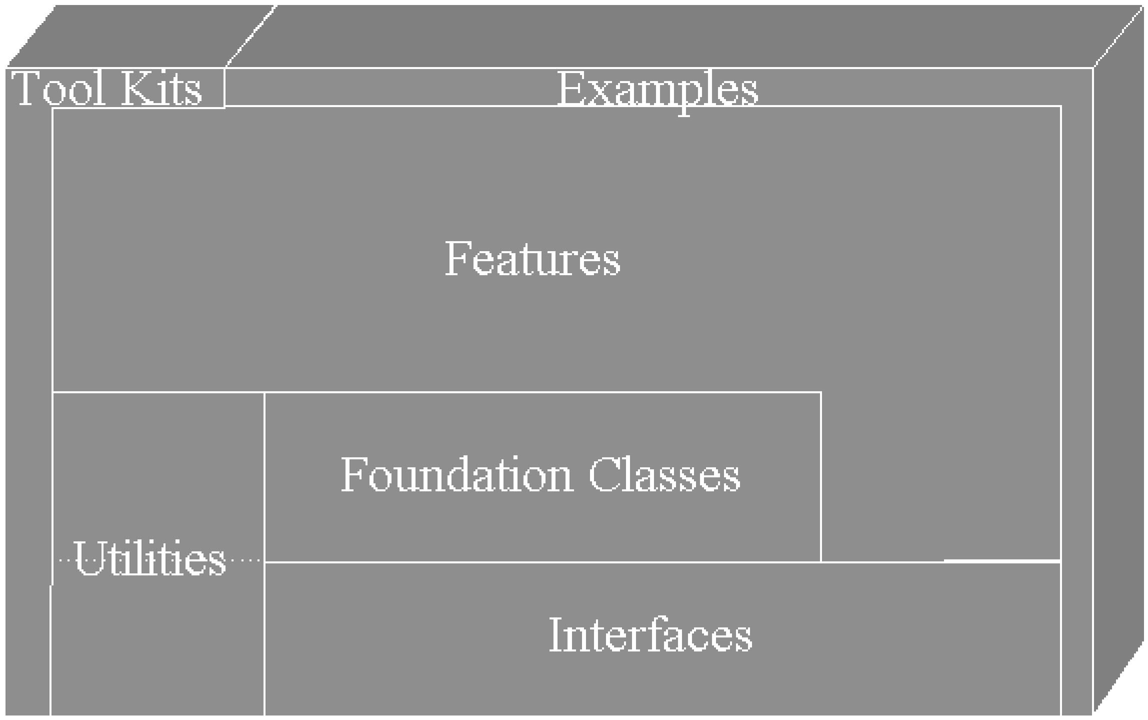

Figure 2: Packages as Building Blocks

Interface Package

Problem

How do we define and identify the foundational concepts of our framework?

Forces

- Interfaces define roles
- Code should be written to interfaces

Solution

Put all interfaces that are specific to the framework in a single package. Therefore, the basic roles of a framework can be easily identified without dragging in all of the gory details.

Some have suggested that the core of Framework documentation should be this package and some diagrams to show the relationship between the interfaces (or Defined Roles). We believe this could be a fundamental part of any framework documentation. Unfortunately we don't know of enough thorough and proven examples to back this up. We are also suspect of documentation that does not include prose, as a picture takes a thousand words to describe.

Foundation Class Package

Problem

Other than method signatures, interfaces give no leverage for concrete classes. How do we quickly find the leverage we need so we can avoid implementing relevant interfaces from scratch.

Forces

- There is some basic collection of concrete or nearly concrete classes that can give us an almost working application of the simplest type that uses the framework.
- Finding these should be the first step in getting an application up in some form.

Solution

Put all Refocused Abstract Classes and/or Simple Classes in a single package that is dependent on no other packages than the interface package and/or possibly some interfaces and/or classes in Utilities Packages.

These classes typically implement interfaces defined in the Interface Package. One might question whether there may be exceptions to this. For example, some Simple Classes may need helper classes to be useful but they may never need to be identified or modified by anyone. Would these helper classes still go in this package? I would think these would be associated with Common Roles. The interface should be created if it is not already. If not, you would be limiting users of your framework if they did want to make a modification because you didn't write Interface-Oriented Code.

Utilities Package(s)

Problem

Another package could use a few of the classes in your framework, but it is overkill to reuse the whole framework. How do you develop and communicate their reusability?

Forces

- Frameworks are often used in larger systems. Parts of these systems aren't even aware of a particular framework. Developers are not about to dig through the framework in hopes that they might find the one gem that will help them without some strong impetus.
- If the gem exists, you certainly don't want other developers to miss it.

Solution

Create one or more packages for the utilities interfaces and classes which do not rely on anything. This will allow other applications which do not use the framework benefit from the utilities developed without having to take the rest of the framework.

Examples

HotDraw for Java uses an interface, StringRenderer, and some classes that implement the interface in order to efficiently draw text in a rectangular region. Some of these classes figure out where to wrap so words don't get awkwardly divided. This could be useful for many applications that don't use HotDraw and is included in a separate utility package. This same utility package provides a ReverseEnumerator to go iterate through Vectors in reverse order. Since there are not many utility classes they are all put into one package. However, one could easily imagine reasons for creating several packages: some for UI issues, others for useful classes which add functionality for standard non-graphic or I/O classes, etc.

Feature Kit Packages

Problem

There are many features that are optional in a framework. How should these features be made accessible without overwhelming the user with all the details?

Forces

- Many developers want one feature, but not another.
- Developers want to easily be able to identify the source of a feature they desire.

Solution

For each family of features, create a software package. Make sure each package employs only the Interface, Foundation, and Utility Packages whenever possible. More abtractions can be added as necessary to allow siblings to gracefully be excluded. This is somewhat of a refinement of the Component Library pattern in Patterns for Evolving Frameworks.

Examples

HotDraw for Java has feature kits for Lines, Text, Shapes, Polygons, and Rectangles.

Example Packages

Problem

How do you verify the framework is complete enough to serve its purpose and prove this to others?

Forces

- Prerequisite for reuse is use
- Nothing seems to explain the abstract like concrete examples.

Solution

Write some simple example applications which use the framework and include them in one or more example packages. This will both verify the framework is viable and serve as useful documentation.

Tool Kit Package(s)

Problem

Frameworks often have unique properties that invite customized tools to help developers understand what's going on, tune performance, etc. Where would we put the classes that make up these tools?

Forces

- These tools help make the framework developer and user more effective. They should be available as part of the framework
- When deploying applications which use the framework, these tools just take up space as they are meant for development only.

Solution

Put all of the tools into a package that is separate from those needed to provide end user functionality. This package can then be removed from deployed applications which use other parts of the framework Typically all of the tool classes will go in one package. However, when there are a lot of tools available, one should consider breaking them into smaller subsets for organizational purposes.

One drawback of this pattern in Java is the lack of ability to add methods to a class at load time due to the security model. In Smalltalk and other dynamically bound languages without such a security model, methods that exist solely for development purposes can easily be added to or removed from a class. In Java, these methods have to live in the package the class lives in. This encourages keeping these methods to a minimum to avoid baggage at run-time.

Simple Classes

Problem

If frameworks are based on collaboration of abstract classes, each application may have to provide several concrete classes before any one can be tested. How can this be avoided?

Forces

- Want to minimize barrier to entry
- Want to maximize flexibility

Solution

For each fundamental role, provide simple versions of classes that are completely functional. These classes can be used as stubs when application developers are working on the details of their own individual classes. They also serve as an example of how one might create their own concrete versions of objects fulfilling particular roles.

Null Object is actually an often useful specialization of Simple Class.

Examples

In HotDraw for Java, SimpleDrawingCanvas doesn't do any double buffering for animation, or anything else more advanced, but it works if you want to use it while working on other things. In the JDK 1.0.2 version of the framework, SimpleEventHandler is really a Null Object because it doesn't do anything with any of the events it receives.

Concurrently Developed Packages

Problem

In what order does one develop each of these packages

Forces

- It is difficult to go from the abstract to the concrete.
- Abstractions can not be tested, only concrete classes can.

Solution

Develop these packages simultaneously. Realize that getting concrete classes to work may imply fixing abstract classes and interfaces. Additionally, abstractions are often not realized until two or more concrete examples of common code are written.

Protected Private Natives

Problem

Forces

Solution

Interface Cast

Problem

Forces

Solution

instanceof Interface

Problem

Forces

Solution

Attribute-free Interface

Problem

Forces

Solution

Overloaded Cast

Problem

Forces

Solution

Acknowledgements

Special thanks go to Ralph Johnson who shepherded this paper and made many excellent observations and made me rethink many of the patterns (and discard several of the ones that weren't). He also helped me bring a better focus to the raw material. I couldn't have asked for a more qualified shepherd. I'd also like to thank the subgroup of TriPLUG (Dan Holmes, Duff O'Melia, Margaret Mahoney, Graham Poor, Matt Walter) who exchanged the experience of a Writer's Workshop for what I learned from being the fly on the wall for an earlier version of this paper… I got the better end of the deal. I'd also like to thank Bobby Woolf for moderating that event. Most importantly, I'd like to thank Jesus Christ for not only being my Lord and Savior, but also giving me whatever talents I have and the ability to put my ego aside for at least long enough to take the constructive criticism I need to help improve the paper.

---

last updated 3-Aug-98

Ken Auer <kauer@rolemodelsoft.com>
RoleModel Software, Inc.
5004 Rossmore Dr.
Fuquay-Varina, NC 27526

(v) 919-557-6352
(f) 919-552-8166