Evolve Frameworks into Domain-Specific Languages

Authors: Don Roberts, Ralph Johnson, University of Illinois

{droberts,johnson}@cs.uiuc.edu

Introduction

Framework development is expensive.

Therefore, frameworks should be developed only when many applications are going to be developed within a specific problem domain, allowing the time savings of reuse to recoup the time invested to develop them. This pattern language describes how to create a framework by evolving it from its initial applications to a domain-specific graphical language for representing solutions within a particular problem domain.

It is hard to come up with examples of framework evolution because the history of most frameworks is hidden. The public only sees the final version of the framework, and once it is released, it essentially stops evolving. We are using the Model-View-Controller framework as our example because it is widely used and has continued to evolve ever since it was released in the early 1980s. However, these patterns apply to development of any framework in any domain, not just GUI frameworks. We have seen these patterns appear in accounting frameworks, insurance frameworks, and compiler frameworks. Additionally, even though MVC was written in Smalltalk, these patterns also apply to frameworks developed in other languages such as C++.

The patterns in this paper form a pattern language. The patterns are designed to be applied in a particular order. Figure 1 shows the relationship between the various patterns in the language. These patterns describe a common path that frameworks take, but that it is not necessary to follow the path to the end to have a viable framework. In fact, most frameworks stop evolving before they reach the end. In some cases this is because the frameworks die; they aren't used any more and so don't change any more. In other cases, it is because it is better for the frameworks to stay more white-box. We hope that the forces for the various patterns describe why.

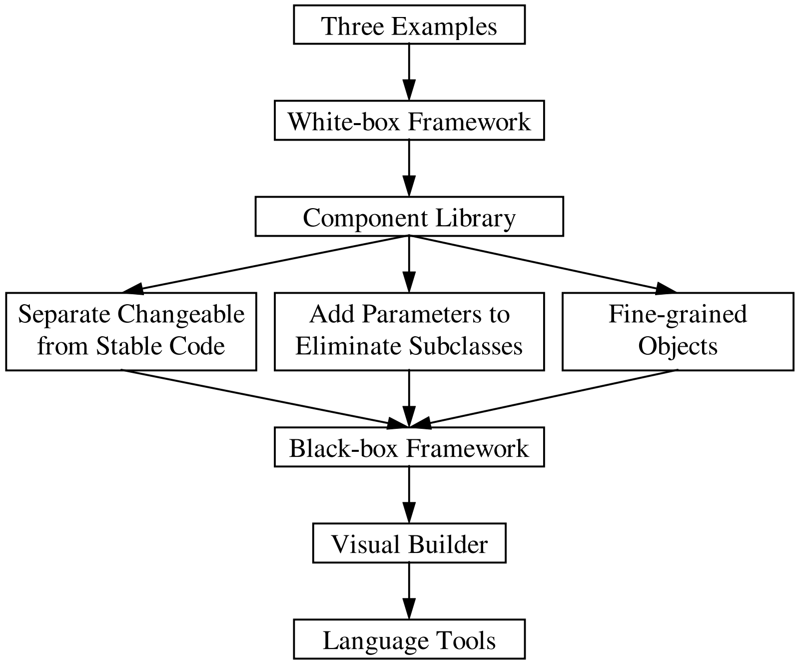

Figure 1 - Relationship between patterns in the pattern language.

Three Examples

Context

You have decided to develop a framework for a particular problem domain.

Problem

How do you start designing a framework?

Forces

- People develop abstractions by generalizing from concrete examples. Every attempt to determine the correct abstractions on paper without actually developing a running system is doomed to failure. No one is that smart. A framework is a reusable design, so you develop it by looking at the things it is supposed to be a design of. The more examples you look at, the more general your framework will be.
- Designing applications is hard. You can't have too many examples or you'll never get your framework done.
- Having a, even marginally useful, framework makes it easier to develop applications. Once you get the first version of the framework, it will be easier to develop more examples.
- Projects that take a long time to deliver anything tend to get canceled.

Solution

Develop three applications that you believe that the framework should help you build.

Rationale

Your framework won't be done after three applications. You can expect it to evolve. However, it should be useful and you can use it to gather more examples. Just don't acquire too many users initially, the framework *will* change!

There are two approaches to developing these applications. In the first approach, the applications are developed in sequence by a single team. This allows the team to begin reusing design insight immediately at the possible expense of narrowness. In the second approach, the applications are developed in parallel by separate teams. This approach allows for diversity and different points of view at the expense of the time it will take to unify these applications in the future.

Some people have built a particular kind of application many times, so they might be able to design a framework without first building an example. They are not counterexamples, they just built their examples before they decided to start the framework.

Example

We don't know what the original examples were that the designers of MVC had in mind. We do know that the usage of MVC in many applications has affected how it evolved from its roots into the framework currently implemented in VisualWorks 2.5.

Related Patterns

The initial versions of the framework will probably be white-box frameworks.

White-box Framework

Context

You have started to build your second application.

Problem

Some frameworks rely heavily on inheritance, others on polymorphic composition. Which should you use?

Forces

- Inheritance results in strong coupling between components, but it lets you modify the components that you are reusing, so you can change things that the original designer never imagined you would change.
- Making a new class involves programming.
- Polymorphic composition requires knowing what is going to change
- Composition is a powerful reuse technique, but it is difficult to understand by examining static program text.
- Compositions can be changed at runtime.
- Inheritance is static and cannot be easily changed at runtime.

Solution

Use inheritance. Build a white box framework [Johnson, Foote, 1988] by generalizing from the classes in the individual applications. Use patterns like *Template Method* and *Factory Method* [Gamma et al., 1995] to increase the amount of reusable code in the superclasses from which you are inheriting. Don't worry if the applications don't share any concrete classes, though they probably will.

Rationale

Once you have a working framework, you can start using it. This will show you what is likely to change and what is not. Inheritance is the most expedient way of allowing users to change code in an object-oriented environment.

Example

The Model-View-Controller framework for graphical user was originally a white-box framework. New view and controller classes were created by making subclasses of the **View** and **Controller** classes, respectively. For instance, to create a scrolling view, a programmer would have to create a new subclass of **ScrollController** to handle the scrolling behavior for the view.

Related Patterns

As you develop additional applications, you should begin to build a component library.

Component Library

Context

You are developing the second and subsequent applications based on the white-box framework.

Problem

Similar objects must be implemented for each problem the framework solves. How do you avoid writing similar objects for each instantiation of the framework?

Forces

- Bare-bones frameworks require a lot of effort to use. Things that *work out-of-the-box* are much easier to use [Foote, Yoder, 1996].
- Up front, it is difficult to tell which components users of the framework will need. Some components are problem-specific while other occur in most solutions.

Solution

Start with a simple library of the obvious objects and add additional objects as you need them.

Rationale

As you add objects to the library, some will be problem-specific and never get reused. These will eventually be removed from the library. However, these objects will provide valuable insight into the type of code that users of the framework must write. Others will be common across most or all solutions. From these, you will be able to derive the major abstractions within the problem domain that should be represented as objects in the framework.

Example

The first step in creating a component library for MVC, which took place in the early 80's, was the creation of *pluggable views*. This provided a way to adapt a controller and view pair to the interface of a particular model. For example, **SelectionInListView** was a pluggable view that provided a list view that could be adapted to any model that contained a list. MVC as a whole was still fairly white-box, since you had to make a new subclass to do anything other than adapt to a particular model.

Related Patterns

As components get added to the library, you will begin to see recurring code that sets of components share. You should separate the changeable from stable code.

Separate Changeable from Stable Code

Context

You are adding components to your component library.

Problem

As you develop applications based on your framework, you will see the same code being written over and over again. How do you eliminate this common code?

Forces

- If changeable code is scattered across an application, it is difficult to track down and change.
- If changeable code is located in a common location, program flow can be obfuscated.

Solution

Separate code that changes from the code that doesn't. Ideally, the varying code should be encapsulated within objects whenever possible, since objects are easier to reuse than individual methods. Good names for the objects will make the control flow less important to understanding the framework [Beck]. The following table shows possible design patterns to use when different portions of the framework change from application to application: [Gamma et al., 1995]

What varies	Design Pattern
Algorithms	*Strategy*
Actions	*Command*
Response to change	*Observer*
Interactions between objects	*Mediator*
Object being created	*Factory Method*
Object interfaces	*Adaptor*

Rationale

If the framework is being reused extensively (as it should be), certain pieces will vary often. By gathering the code that varies into a single location (object) it will both simplify the reuse process and *show users where the designers expect the framework to change*.

Example

Objectworks 4.0 began to make extensive use of wrappers. The stable code was stored in the wrappers and the varying code was still in subclasses of **View**. This was in response to users of the framework having to write similar code every time they wanted to add a particular function to a view or controller.

Related Patterns

To separate the changeable code from the stable, you will often have to create finer grained objects. Often these fine-grained objects will cause your framework to become more black-box.

Add Parameters to Eliminate Subclasses

Context

You are adding components to your component library.

Problem

Most of the subclasses that you write differ in trivial ways. (e.g., Only one method is overridden) How do you avoid having to create trivial subclasses each time you want to use the framework?

Forces

- New classes, no matter how trivial, increase the complexity of the system.
- Complex sets of parameters make parameterized classes more difficult to understand and use.

Solution

Design adaptable subclasses that can be parameterized with messages to send, indexes to access, blocks to evaluate, or whatever else distinguishes one trivial subclass from another. To make the many parameters comprehensible, make sure that you choose a good name for the object creation method.

Rationale

If the differences between subclasses is trivial, creating a new subclass just to encapsulate the small change is overkill. Adding parameters to the instance creation protocol provides for reuse of to original class without resorting to programming.

Example

In VisualWorks, the class **AspectAdaptor** is designed to allow any object to behave as if it were a **ValueHolder**, (i.e., respond to the value and value: messages by getting and setting a value, respectively). It is parameterized by two instance variables, getSelector and putSelector. Without this parameterization, a new subclass would have to be made every time that an new type of object needed to be adapted.

Related Patterns

Creating adaptable subclasses is one way to separate changeable from stable code. The parameters can be automatically supplied by a builder.

Fine-grained Objects

Context

You are refactoring your component library to make it more reusable.

Problem

How far should you go in dividing objects into smaller objects?

Forces

- The more objects there are in the system, the more difficult it is to understand.
- Applications can be created by simply choosing the objects that implement the functionality that is desired within the application. No programming is required.

Solution

Continue breaking objects into finer and finer granularities until it doesn't make sense to do so any further. That is, dividing the object further would result in objects that have no individual meaning in the problem domain.

Rationale

Since frameworks will ultimately be used by domain experts (non-programmers) you will be providing tools to create the compositions automatically. Therefore, it is more important to avoid programming. The tools can be designed to manage the proliferation of objects.

Example

Beginning with Objectworks 4.0, the Model-View-Controller became more fine-grained. This was accomplished by using new objects that represented finer grained concepts than the original **Model**, **View**, and **Controller** classes. A couple of examples of these objects were **Wrappers**, which allowed a unit of functionality to be added to any view, and **ValueHolders**, which allowed views to depend only on a portion of a model rather than the entire model. With this version, to make a view scrollable, the programmer only needed to apply a **ScrollWrapper** to his view.

Related Patterns

As the objects become more fine-grained, the framework will become more black-box.

Black-box Framework

Context

You are developing adaptable subclasses by separating changeable code from stable and making fine-grained objects.

Problem

Some frameworks rely heavily on inheritance, others on polymorphic composition. Which should you use?

Forces

- Inheritance results in strong coupling between components, but it lets you modify the components that you are reusing, so you can change things that the original designer never imagined you would change.
- Making a new class involves programming.
- Polymorphic composition requires knowing what is going to change
- Composition is a powerful reuse technique, but it is difficult to understand by examining static program text.
- Compositions can be changed at runtime.
- Inheritance is static and cannot be easily changed at runtime.

Solution

Use inheritance to organize your component library and composition to combine the components into applications.

Rationale

Black-box frameworks are named as such since reusing component in a black-box framework involves plugging them together and not worrying about how they accomplish their individual tasks [Johnson, Foote, 1988]. Contrast this with white-box frameworks that require an understanding of how the classes work so that correct subclasses can be developed.

People like to organize things into hierarchies. These hierarchies allow us to classify things and quickly see how the various classifications are related. By using inheritance, which represents an is-a relationship, to organize our component library, we can rapidly see how the myriad of components in the library are related to each other. By using composition to create applications, we both avoid programming and allow the compositions to vary at runtime.

Example

VisualWorks has essentially taken MVC and made it almost a completely black-box framework. Now, rather than creating various subclasses of **View**, or even reusing various subclasses of view, we take a generic view and add wrappers to it corresponding to the various behaviors that we require. In the same way, rather than creating a complex model class and ensuring that dependencies get updated correctly, we simply compose a bunch of **ValueHolders** to hold the values in our model and let them worry about updating their dependents.

Related Patterns

By organizing the component library in the manner, we support the creation of a builder that allows the library to be browsed and compositions to be created graphically.

Visual Builder

Context

Your now have a black-box framework. You can now make an application entirely by connecting objects of existing classes. The behavior of your application is now determined entirely by how these objects are interconnected. A single application consists of two parts. The first part is the script that connects the objects of the framework together and then "turns them on." The second part is the behavior of the individual objects. The framework provides most of the second part, but the application programmer must still provide the first part.

Problem

The connection script is usually very similar from application to application with only the specific objects being different. How do you simplify the creation of these scripts?

Forces

- The compositions that represent applications of the framework are convoluted and difficult to understand and generate.
- Building tools is expensive.
- Domain experts are rarely programmers.

Solution

Make a graphical program that lets you specify the objects that will be in your application and how they are interconnected. It should generate the code for an application from its specification.

Rationale

Since the code is basically just a script, the tool can generate it automatically. The tool will also make the framework more user-friendly by providing a graphical interface to it that should draw on the standard notations present in the problem domain. At this point, domain experts can create applications by simply manipulating images on the screen. Only in rare cases should new classes have to be added to the framework.

Example

In Visualworks 1.0, ParcPlace provided an interface builder to allow users to "paint" the GUI on a canvas. The builder takes the graphical description of the GUI and creates the application from it. It does this by creating an **ApplicationModel** with **ValueHolders** of the appropriate type for each control. It creates composite **View** objects that make up the main View and adds Wrappers to create the appropriate functionality for each widget. All of this information is stored in a windowSpec, which is a declarative description of the entire GUI. At runtime, the windowSpec is fed to a **UIBuilder** that interprets it, creates the objects it describes, and composes them to create the final user-interface.

Related Patterns

Congratulations! You have just developed a visual programming language. Note that this implies that you will need language tools, just like any other language.

Language Tools

Context

You have just created a builder.

Problem

The visual builder creates complex composite objects. How do you easily inspect and debug these compositions?

Forces

- Existing tools are usually inadequate for dealing with the specialized composition relationships present in the framework.
- Building good tools is an expensive task that can be viewed as overhead.

Solution

Create specialized inspecting and debugging tools.

Rationale

Since the system you have created is essentially a graphical domain-specific programming language, it will require language tools to help debug and understand it. The tools that came with the language that you built your framework in will probably not be as good as they should be, because your framework will be filled with little objects that all look alike, and half of them will be completely uninteresting to someone who just wants to build an application.

Example

Visualworks doesn't have this yet, but it is the next logical step in its evolution. Many of the long-time MVC programmers have complained that the current framework is more difficult to use since the views are large hierarchical structures of composed wrappers on wrappers. The real problem is that there are no inspectors or debuggers specifically designed for handling these compositions. If such tools existed, debugging or inspecting a view should be no more difficult than inspecting any other object in the system.

Acknowledgments

We would like to thank Ward Cunningham for his shepherding and valuable insight that is reflected in the final version of this paper. We would also like to thank the UIUC patterns group for their scathing reviews that helped us discover some of the weaknesses in our presentation.

References

Beck K. *Smalltalk Best Practice Patterns — Volume 1: Coding*. (to be published).

Foote B, Opdyke W. Life Cycle and Refactoring Patterns that Support Evolution and Reuse. *First Conference on Pattern Languages of Programs (PLoP'94)*. Monticello, Illinois, August, 1994. *Pattern Languages of Program Design*. Edited by James O. Coplien and Douglas C. Schmidt. Addison-Wesley, 1995.

Foote B, Yoder J. Attracting Reuse. *Third Conference on Pattern Languages of Programs (PLoP'96)*, Monticello, Illinois, September 1996.

Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Object-Oriented Software*. Addison-Wesley, 1995.

Johnson R, Foote B. Designing Reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.