Design Fragments Make Using Frameworks Easier

George Fairbanks, David Garlan, William Scherlis

Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA, 15213, USA
{george.fairbanks,david.garlan,william.scherlis}@cs.cmu.edu

Abstract

Object oriented frameworks impose additional burdens on programmers that libraries did not, such as requiring the programmer to understand the method callback sequence, respecting behavior constraints within these methods, and devising solutions within a constrained solution space. To overcome these burdens, we express the repeated patterns of engagement with the framework as a *design fragment.* We analyzed the 20 demo applets provided by Sun and created a representative catalog of design fragments of conventional best practice. By evaluating 36 applets pulled from the internet we show that these design fragments are common, many applets copied the structure of the Sun demos, and that creation of a catalog of design fragments is practical. Design fragments give programmers immediate benefit through tool-based conformance assurance and long-term benefit through expression of design intent.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Design, Documentation, Languages, Standardization

Keywords Object-oriented Programming, Frameworks, Patterns, Design Fragments

1. Introduction

Programmers use object oriented frameworks because they provide partially-complete and pre-debugged solutions to common problems, such as windowing systems and application servers. Popular examples of frameworks include Enterprise Java Beans[24], Microsoft .NET [4], and Java applets [23].

Frameworks differ from code libraries in that the framework, not the programmer, provides important parts of the architectural skeleton of the application [20] and, in doing so, places additional burdens on the programmer. The applet shown in Figure 1 is from the original Sun Java Development Kit (JDK) and it has a bug in it. No amount of code inspection can reveal the bug unless you also know how the applet framework will drive that code. The applet framework invokes the `start()` method then the `stop()` method on the programmer-provided code – perhaps this much could be guessed from the method names. Additionally, the runnable framework will invoke the `run()` method on the applet sometime after the `timer.start()` method is called. The bug is a race condition that can be reasoned about only if you know that the framework may call `start()` and `stop()` multiple times.

```
public class Clock2
  extends java.applet.Applet
  implements Runnable {
...
Thread timer = null;
...
public void start() {
  if (timer == null) {
      timer = new Thread(this);
      timer.start();
    }
}
public void stop() {
  timer = null;
}
public void run() {
  while (timer != null) {
    try {
      Thread.sleep(100);
    } catch (InterruptedException e){}
    repaint();
  }
  timer = null;
}
```

Figure 1. Threaded applet example

The programmer's intent during the `stop()` callback is to signal to the running thread that it should terminate. The signal is that the `timer` field is set to null. The race condition occurs when the framework invokes both `start()` and `stop()` before the thread checks the value of the `timer` field. When that happens, the old thread continues executing, since it missed the signal to terminate, and a new thread executes too, since it was started in the second call to `start()`. The bug was fixed in the next release of the JDK and the check for `timer != null` was replaced with `timer == Thread.currentThread()`. The burden placed on the programmer is to know when the framework will call his code.

Initially, it is not apparent why this applet is using a thread at all. Because this is a clock applet, it needs to regularly paint the correct time onto the screen. It would seem reasonable to do this

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

painting in the `start()` method. However, the framework has an implicit requirement that the programmer complete his processing within a few fractions of a second in the `start()` and `stop()` callbacks because the GUI of the applet is blocked until these methods complete. So, if the programmer were to just continuously paint the clock after the `start()` callback, the GUI would be unresponsive. The creation of a background thread is to satisfy both the framework's demands and the program's purpose. The burden placed on the programmer is to know what rules his code must follow.

If the programmer were freed from the framework's choice of what methods get called when, he might already know how to coordinate two threads to avoid a race condition. The programmer cannot rewrite the framework, so he must live within the framework's rules and some of his prior expertise cannot be used. In extreme cases, a framework provides just a single way to accomplish a task. For example, in the applet framework there is only one way to find out what the mouse is doing. The code must register as a mouse listener, implement the MouseListener interface and its required callback methods, and de-register. Additionally, this single path to success must be coordinated with the applet's callback sequence, so registering almost always happens in the `init()` callback and de-registering in `destroy()`. The burden placed on the programmer is to be more ingenious in finding a solution within a constrained solution space.

For other forms of reuse, such as libraries, programmers use method-level documentation and possibly method specifications. Neither of these is an appropriate place to document something like the example shown because the documentation would need to be spread across many methods. Furthermore, there are other things that can be accomplished with the `start()` and `stop()` methods besides coordinating a thread, so the documentation for these methods would be cluttered with many "if you want to do this, then ..." clauses.

Instead, leading framework authors [8] suggest that programmers follow the "Monkey see / monkey do" rule and copy code examples in which other programmers have used the framework. "When copying an example, you are looking for structure, much of which you'll typically delete because it doesn't apply, or at least not yet." When a programmer applies this rule, he must recognize that the code does what he wants (e.g., coordinate a background thread with an applet), cull the extra details, and reproduce that code elsewhere. However, finding such concrete patterns is difficult and time-consuming, separating out the relevant parts is error-prone, and preserving design intent is impossible. It is with good reason that this practice has been termed "rape and paste programming" [1].

Programmers want help that provides the advantages of "Monkey see / monkey do" but removes the drawbacks. Ideally, this help would encode the structure of the solution, reveal relevant details about the framework, avoid swamping them with information, and check conformance between their code and their intent.

This paper describes two contributions toward that ideal. First, we provide an improved technique to help programmers overcome the burdens of using frameworks. The technique, called design fragments, is based on the representation of structural patterns in the vein of JavaFrames [12] and OOram [28]. To the description of what the programmer must do, we add a minimally sufficient description of how the framework will act on the programmer's code so that the programmer can reason about the code's interaction with the framework. Tools inside the programmer's development environment can compare the programmer's stated intent with his source code and warn him when the two diverge. Such an approach has clear short-term benefit for the programmer.

It promises long-term benefits also because design intent has been expressed. For example, Sun fixed the Clock2 applet shown above between JDK 1.0 and 1.1, yet an internet search reveals that applets based on that 1.0 code example are still prevalent. With design intent expressed, the promise is that tools to check conformance would help with this and similar evolution problems.

In short, design fragments provide programmers with solutions, assure conformance with those solutions through code analysis, and, because design intent has been captured, provide an opportunity to catch errors, speed code comprehension, and improve code quality over the long-term.

The second contribution is a detailed case study on the applet framework that examines how fifty-six applets use design fragments. The case study allows us to evaluate the following hypotheses. 1. Source code that uses a framework is likely to follow structural patterns. 2. Programmers refer to examples to learn how to use frameworks. 3. The effort to create a catalog of design fragments tapers off. 4. The design fragments language can be used to express the patterns we discovered in the example code. Hypotheses like these are necessary in order for a structural pattern technique, such as design fragments or JavaFrames, to be a viable solution. Our results indicate that, for the most part, the hypotheses are true.

In this paper we describe the concepts behind design fragments in detail, the language used to express design fragments, the tooling we have built that allows conformance checking between the programmers code and the design fragment, and how we envision design fragments contributing to the software life cycle. We describe a case study involving the Java applet framework and analyze our hypotheses with respect to the collected data. We conclude by relating design fragments to related work and describe our future plans.

2. Design Fragments

A design fragment is a pattern that encodes a conventional solution to how a program interacts with a framework to accomplish a goal. It has two primary parts. The first is a description of what the programmer must build to accomplish the goal of this design fragment, including the classes, methods, and fields that must be present. This description also includes the behavior of these methods. The second part of the design fragment is a description of the relevant parts of the framework that interact with the programmer's code, including the callback methods that will be invoked, the service methods that are provided, and other framework classes that are used.

A design fragment provides a programmer with a "smart flashlight" to help him understand the framework. This smart flashlight illuminates only those parts of the framework he needs to understand for the task at hand. Without the smart flashlight, a programmer browsing the framework classes is swamped with private implementation details or unable to differentiate the relevant from irrelevant. Even simple parts of a framework have enormous complexity [17]: In the Swing user interface framework, the JButton class has 160 methods and fields, while JTree has 336.

Each framework will have its own catalog of design fragments that act like a handbook, collecting conventional solutions to problems. Programmers can see examples of the design fragment in use by navigating from the catalog to the code that implements a given design fragment. Discipline must be used when revising design fragments in the catalog, similar to the discipline used in revising code libraries with existing clients.

Design fragments have two immediate benefits for programmers. First, analysis tools can check conformance between the programmer's stated intent and his source code. Second, programmers who do not know a part of the framework can quickly find a solution in the catalog. We believe it is essential to provide programmers immediate value for their investment of effort.

Once programmers have started using design fragments, long-term benefits arise.

- Since programmers have expressed their intent (e.g., this code follows the Threaded Applet design fragment), it becomes possible to analyze the code with respect to that intent. Even if analysis tools are not available at the time the code is written, the expression of intent endures and can be checked by stronger tools available tomorrow. For example, our current tools cannot check for concurrency bugs today but in the future static analysis tools from the Fluid project [11] could ensure correct threading behavior.
- The use of design fragments allows other programmers to comprehend the code more quickly. At a glance it's possible to see, for example, that this is an applet that listens for mouse events, has a background task, and takes in parameters from HTML. Design fragments convey architectural information that is different from and complimentary to an Acme [10] architecture model.
- Evolution of code is easier. As was seen in the example from the introduction, the bug in the source code was detected and fixed but not before other code had cloned its structure. A design fragment could be marked as deprecated, causing programmers to examine their code and fix the bug. Note that in the example there is no single method or class that can be deprecated, only the collection of methods and classes that are used in a particular way.
- Unnecessary code diversity can be reduced. Instead of a task being implemented slightly differently by various programmers on the same team, they could standardize on a particular design fragment. This yields benefits in code comprehension and may reduce bugs. Framework authors could deliver both example applications, as they do now, as well as a catalog of design fragments. This catalog could act as a seed crystal so programmers would use the conventional solution unless they had a good reason to deviate.

3. Design Fragment Language

In describing the design fragment language, we will continue to use the code example from the introduction (despite its bugs), shown earlier in Figure 1. The intention of the language is to express:

- the structure of the programmer's code
- the behavioral requirements of the programmer's code
- the *relevant* structure of the framework code
- the *relevant* behavior of the framework.

One of our goals is to keep the language sufficiently simple that programmers can rapidly create and comprehend design fragments.

3.1 Structure

The structure for the threaded applet example is shown as a UML class diagram in Figure 2. The classes above the line are provided by the framework. Note that only methods that are relevant to this design fragment are shown on the framework classes. The classes below the line are roles in the design fragment and will be bound to the programmer's classes.

The design fragment language is expressed in XML. The language can refer to classes, interfaces, fields, methods, return values, and method parameters. Figure 3 shows how the RoleThread class is represented. The *provided="no"* clause indicates that RoleThread is not a framework-provided class.

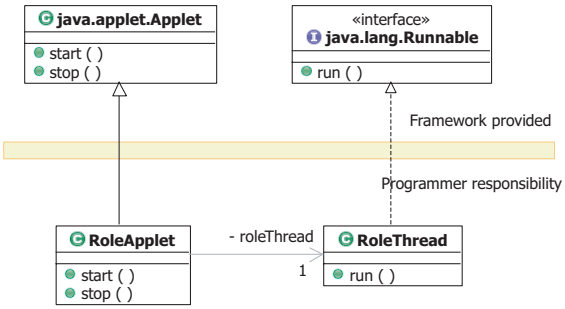

Figure 2. Structure of threaded applet

```
<class name="RoleThread" provided="no">
  <implementsinterface
        name="java.lang.Runnable" />
  <method name="run" returnvalue="void">
  </method>
</class>
```

Figure 3. RoleThread structure

3.2 Behavior

In addition to the code structure, the programmer needs to know how his code should behave. The required behavior in the threaded applet example is as follows. In the `start()` callback, the code should create a new RoleThread instance and assign it to the roleThread field. In the `stop()` callback, the code should set the roleThread field to null. We use the null field as a signal to the thread that it should terminate because calling `roleThread.stop()` is deprecated and unsafe. Then, in the `run()` callback, the code should loop repeatedly while checking that the roleThread field has not been set to null.

Most of this behavior can be expressed in the design fragments language. The required creation of a new instance is expressed like this: *requirednewinstance target="java.lang.thread.Thread" arguments=""*. A required method call is expressed like this: *requiredcallspec target="roleThread" method="start" arguments=""*. Additionally, freeform text can also be entered as a specification: *freeformspec text="By the end of this method, a new thread must be running and assigned to roleThread field*". However, note that the looping behavior required in `run()` cannot be expressed in the current design fragments language except as freeform text.

Inheritance and interfaces work the same way as Java, which includes single inheritance for classes but multiple implementations of interfaces. Constraints on behavior that are placed on superclasses or interfaces are inherited by subclasses.

New constraints can be added and, optionally, corresponding analyses for checking the constraints. The parser ignores any constraints it does not understand. So, for example, a method could be written with the following specification: *timingspec totaltime="200ms"* to indicate that the method should complete within 200ms. If desired, an analysis could be written to advise the programmer on the running time behavior of his method and the likelihood that it would violate the specification.

The behavior of the framework classes can be expressed with the specifications described above, but it can also use some new specifications that are specific to frameworks. In the threaded applet example, the programmer needs to know which framework methods are callback methods, the sequence of the callbacks, and how many times the callback can occur. The specification *invocationtype value="callback"* indicates that this framework method is

```
<class name="java.applet.Applet"
    provided="yes">
 <method name="start" returnvalue="void">
  <freeformspec text="Callback method;
    invoked when framework decides to
    initialize your applet." />
  <invocation-cardinality value="*" />
  <invocation-lifecycle value="yes" />
  <invocation-type value="callback" />
  <invocation-pair value="stop" />
  <invoked-before value="stop" />
 </method>
...
</class>
```

Figure 4. *start*() method in java.applet.Applet

a callback, not a service method. Callback methods are invoked by the framework on the programmer's code while service methods are provided by the framework for the programmer to invoke. Most callback methods are lifecycle methods but not all. On applets, the specification *invocation-lifecycle value="yes"* would be placed on `init()`, `start()`, `stop()`, and `destroy()`, but not on `paint()` or `getParameterInfo()`. Some callbacks occur in matched pairs, so the specification *invocation-pair value="stop"* would be placed on the `start()` method. Finally, placing the specification *invoked-before value="stop"* on the `start()` method indicates that it will be invoked before the `stop()` method. An example of these specifications is seen in the framework method `start()` on the `java.applet.Applet` class in Figure 4.

3.3 Bindings to Java Source Code

Each time a programmer wants to use a design fragment, he declares it using a Java 5 annotation. Figure 5 shows the declaration of a new instance named bc1 of the design fragment BackgroundContinuousV1. The Java 5 annotation, which applies to the demos.applets.clock package, is shown italicized. This annotation lives in the special Java file package-info.java.

```
@df.DFInstances({
 @df.DFInstance(
   df="BackgroundContinuousV1",
   inst="bc1")
}) package demos.applets.clock;
```

Figure 5. Declaration of a design fragment instance

Figure 6 shows some of the bindings between the roles in the design fragment and the Clock2 class. Again, the Java5 annotations are shown italicized. Note that here the Clock2 class plays both the RoleApplet and RoleThread roles from the design fragment, while in other applets, the RoleThread is sometimes a different class than the applet.

The advantages of using Java 5's annotations are that it is a standard mechanism and the annotations can be typechecked using the standard Java compiler. One disadvantage that can be seen above is that conceptually simple bindings become quite verbose.

4. Tool

In order to provide feedback on our ideas and in order to execute a large case study, we built tools to support the creation, binding, and evaluation of design fragments. The tool is an extension to the Java Development Tooling for the Eclipse integrated development environment (IDE) [8]. The tool has three parts that are visible to programmers. The first part is a new view in the IDE that displays a

```
@DFTypeBindings({
 @DFTypeBinding(inst="bc1",
   role="RoleApplet"),
 @DFTypeBinding(inst="bc1",
   role="RoleThread")
}) public class Clock2
     extends java.applet.Applet
     implements Runnable {
...
@DFFieldBindings({
 @DFFieldBinding(inst="bc1",
   role="roleThread")
}) Thread timer = null;
...
@DFMethodBindings({
 @DFMethodBinding(inst="bc1",
   role="start")
}) public void start() {
...
```

Figure 6. Binding of design fragment instance to Java code

catalog of design fragments, as shown in Figure 7. The second part is a new view that displays a list of the design fragments that have been bound to the source code, as shown in Figure 8. The third part is a new set of problem markers that appear in the standard Eclipse problem view, as shown in Figure 9. There are many components that run behind the scenes to keep these views updated, including a builder that re-parses the design fragment definitions when the source files change, a builder that re-evaluates the design fragment bindings when relevant Java source files change, and analysis tools that check conformance between the design fragments and the Java code.

4.1 Design Fragment Catalog View

Figure 7 shows the design fragments catalog view for the Java applets framework. The Background Continuous Task V1 design fragment has been opened, showing the text of its goal, the parts provided by the framework, and the parts the programmer must build. The definition of the `run()` callback method has been opened, showing the specifications of when and how often this callback will be invoked by the framework.

The section on programmer responsibility shows the two class roles, RoleApplet and RoleThread. The class role RoleApplet must be a subclass of java.applet.Applet, must have a field named roleThread of type java.lang.Thread, and must implement the two callback methods `start()` and `stop()`. The `start()` callback method has been opened, showing the specifications of what the programmer is expected to do in order to fulfill framework obligations and this design fragment.

The design fragment catalog is represented as an Eclipse project. The files in the project are the XML design fragment definitions. One advantage of this representation is that Eclipse provides integration with source code control repositories, like CVS or Subversion, for files within projects, so programmers can stay updated with the latest design fragment catalog by synchronizing the project with the server. Following the Debian [3] example of maintaining stable, testing, and unstable builds of their Linux distribution, each design fragment catalog has folders for stable, testing, and unstable design fragments. It is expected that most programmers would use the stable design fragments, which have been vetted by the catalog maintainer, but less risk-averse programmers or ones on the cutting edge could use the testing or unstable folders. Each catalog contains design fragments for just one version of a framework. While the applet framework has changed relatively little over time, other frameworks can change significantly as they evolve.

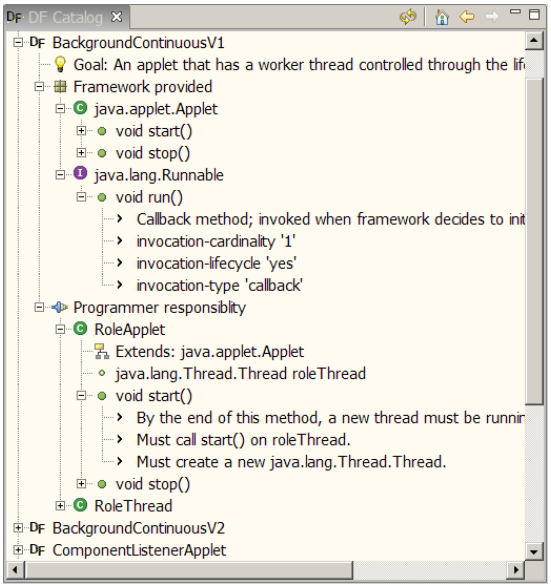

Figure 7. Design fragment catalog view

4.2 Design Fragment Instances View

Each use of a design fragment, which we call a design fragment instance, is given a name and declared in the package-info.java file. Usually a design fragment is used just once per class or package, but not always. For example, one of the applets we analyzed is a two-player Tetris game that used two background threads, so it used two instances of the Background Continuous Task V1 design fragment.

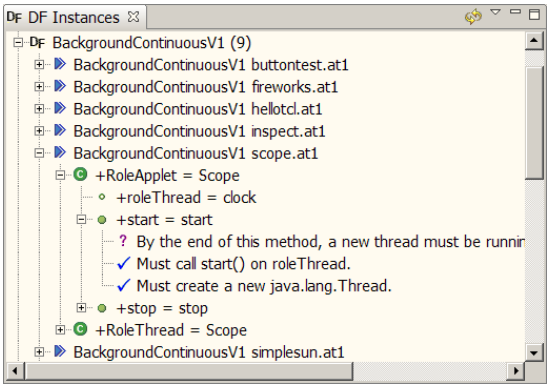

Figure 8. Design fragments instances view

Figure 8 shows some of the nine instances of the Background Continuous Task V1 design fragment. This view shows, for each role in the design fragment, where it is bound in the Java source code. For example, the scope.at1 instance has the RoleApplet and RoleThread roles bound to the Scope Java class, the roleThread role bound to the clock field. This view will indicate binding problems, such as when a class role is not bound to any Java class.

If there are any conformance analysis failures then they are shown in this view. The specifications for the `start()` method have been opened, revealing checkmarks. As currently implemented, the tool can detect when a method fails to invoke a required method and it does this through trivial analysis of the method body. The user interface distinguisheds three states: pass, fail, or no-analysis .

4.3 Eclipse Problems View

In the Eclipse IDE, a single view collects all problems found in the environment. The design fragments tool extends this view by adding new problem markers that appear within the view. As seen in Figure 9, when source code fails to meet the specifications defined in the design fragment, these problems are reported as warnings in the problems view. Clicking on the problem will navigate the programmer to the line in the source code where the problem was detected.

Most problems are reported as warnings, including conformance analysis failures and incomplete bindings. A few problems are reported as errors, including the declaration of design fragment instances where the design fragment is not found in the catalog.

4.4 Integration

The Eclipse IDE includes convenient features like incremental compilation upon the saving of source files so that the list of problems is always accurate. Programmers have grown accustomed to this style of interaction with their tools, so the design fragments tools operate in a consistent way. Changes to the source files that define design fragments cause those files to be re-parsed and presented in the catalog view. Any changes to the catalog will propagate to the design fragment instances view. Similarly, changes to the Java source code will trigger re-analysis of the bindings to design fragments and these will be displayed in the instances view. Any problems detected during these steps are reflected in the problems view.

5. Hypotheses

Our vision for design fragments relies upon some assumptions about how programmers interact with frameworks and example code. These assumptions are reasonable but it is best to state them explicitly as hypotheses so that data can be gathered to support or refute them.

Our first hypothesis is that *patterns exist in source code*, specifically source code that interacts with frameworks. If this is not true, then each programmer interacts with the framework in a slightly different way and it is senseless to try to encode patterns. A corollary to this is that *unnecessary diversity exists in source code*, meaning that sometimes programmers could choose to interact with the framework in a conventional way instead of being different. If unnecessary diversity exists then it might be possible to increase the frequency of pattern use by reducing that unnecessary diversity.

Our second hypothesis is that *programmers reference examples*, specifically source code examples that appear to do the same thing they want their program to do. Frameworks often come with example applications and it is possible that programmers look at these applications and clone their structure using the "Monkey see / monkey do" rule. If programmers do not already reference examples, then convincing them to reference a catalog of design fragments will be more difficult.

Our third hypothesis is that the *effort to create a catalog tapers*. As more applications are analyzed it should become less likely that more design fragments are found. Ideally, the discovery of new design fragments would follow an asymptotic curve. Analysis of a few applications would yield a great many design fragments but the rate of their discovery would slow down as more applications are analyzed. The earlier the catalog reaches 80% of its maximum size, or some other threshold of utility, then the more practical it will be to build it.

Our final hypothesis is that the *design fragments language can express patterns*, specifically the patterns that the authors have rec-

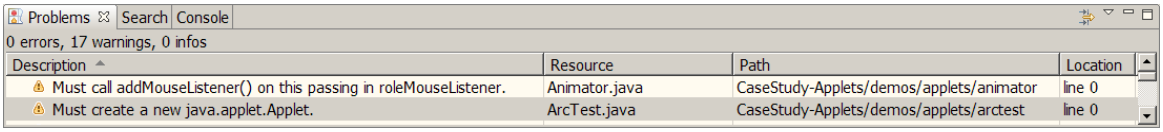

Figure 9. Problems view

ognized in the code. The evaluation for this hypothesis will be subjective and we will note the cases where we recognized a pattern but were unable to express it.

6. Applet Case Study

The applet framework allows Java code to run inside a web browser. Sun has bundled demonstration applets with the Java Development Kit (JDK) since its original version. The JDK today contains twenty demo applets and thousands more can be found on the internet with a simple search.

The applet framework defines several lifecycle callback methods that are invoked on the programmer's applet when the user starts the applet in the web browser. The applet first receives an `init()` callback, then at least one `start()` and `stop()` pair of callbacks, then a single `destroy()` callback. The framework also defines service methods that can be invoked by the programmer's code, such as `addMouseListener()`.

Many other frameworks are more complex than the applet framework but their complexity arises primarily from scale, not from a difference in their natures. All framework programmers are presented with the same challenges, including understanding which methods are callbacks or service methods, the sequence of callbacks, and the assumptions the framework makes about behavior within the callback methods. The applet framework is a suitable choice for research because all of these essential challenges are present, it is well known, and it is small. One exception is that some frameworks rely on declarative elements in addition to object oriented mechanics, either in external files or Java annotations, and we plan to examine this in our future work.

```
public class Simple
    extends java.applet.Applet {
  public Simple() {
    add(new java.awt.Label("Hello"));
  }
}
```

Figure 10. Simple AWT applet

Not all applets, however, meaningfully engage in the applet framework; for an example see the one shown in Figure 10. Since the applet framework is an extension to the Abstract Widget Toolkit (AWT), every legal AWT program is also a legal applet so long as it derives from java.applet.Applet. Since it is our desire to investigate framework use, we must exclude such applets from our study. We define "meaningfully engage in the applet framework" to mean that the code implements applet callbacks or invokes applet-specific service methods on the framework. Specifically, it cannot extend java.applet.Applet yet solely call AWT service methods.

6.1 Design Fragments from Sun Demos

We decided to start populating our catalog of design fragments by examining the demo applets provided in the Sun JDK. The intent of these demo applets was both to impress programmers with the applet framework's capabilities as well as instruct the next generation of applet writers, so it seemed a good place to start looking for canonical patterns of interaction with the framework.

Recognizing a design fragment is equivalent to defining a category. The design fragment author must examine source code and recognize that a subset of that code is repeated elsewhere. The author then encodes this pattern as a design fragment and binds it to the source code. Some of our initial attempts to define design fragments were overly broad (e.g., an applet that paints to the screen) and others overly narrow (e.g., an applet that has a background thread for running a control panel). The selection of an appropriate scope became easier after defining a dozen or so design fragments. However, it is natural that different authors would create different design fragments in same way that different authors would create different code libraries.

From the twenty Sun demo applets we found ten design fragments. A tabulation of these design fragments including a short description and how often they occurred is shown in Table 1. The first column lists the design fragments by category and name. A short description of the design fragment is in the second column, and the number of times the design fragment was found in the Sun demos and the internet are in the third and fourth columns. Note that the Background Continuous V1 and Focus Listener applets have a count of zero for the Sun column because they were not discovered until looking at applets from the internet. Also, the One-time Init Task and Timed Task design fragments were not found in the applets from the internet. All other design fragments were found in both.

In order to ensure that design fragments were consistently identified despite differences in the source code, we established the set of rules shown below to define the required matches and the allowed deviations. The rules were:

- *Background Continuous V2:* Must have a field holding reference to thread, Must create thread in `start()`, assigning field to thread, Must set field to null in `stop()`, Must implement `run()` and loop continuously until field is not the same as currently running thread.
- *One-time Init Task:* Must create thread in `init()`, Thread must execute `run()` to completion just once.
- *One-time On-demand Task:* May create thread at any time, Thread must execute `run()` to completion just once.
- *Timer:* Must create a new TimerTask in `start()` and call `schedule()` on it, Must call `cancel()` on the TimerTask in `stop()`.
- *Event Handling (all kinds):* Must register in `init()` using `addZZZListener()`, Must implement relevant interface, Must implement interface methods, May fail to de-register in `destroy()` using `removeZZZListener()`.
- *Parameterized Applet:* Must call `getParameter()`, perhaps not from `init()`, May fail to define `getParameterInfo()`, May fail to match `getParameter()` calls with `getParameterInfo()` data.
- *Manual Applet:* Must have `main()` method that calls `init()` and `start()` on applet

Design Fragment Name	Description	Instances from Sun demos	Instances from internet
Threading			
Background Continuous v1	A separate thread used to execute an ongoing task	0	9
Background Continuous v2	Same as above, but with a race condition removed	6	3
One-time Init Task	A separate thread used to run a task at startup, once	2	0
One-time On-Demand Task	A separate thread used to run a task at a domain-specific time, once	1	3
Timed Task	A task that should be repeated regularly	1	0
Event Handling			
Component Listener	Listening for component events	1	1
Focus Listener	Listening for when the applet gets focus in the GUI	0	1
Key Listener	Listening for keyboard events	1	2
Mouse Listener	Listening for simple mouse events	10	12
Mouse Motion Listener	Listening for complex mouse events	4	11
Other			
Parameterized Applet	An applet that reads parameters from a web page	13	17
Manual Applet	An applet that can be run from the command line because its main method manually invokes the applet lifecycle methods	5	5

Table 1. Design fragment frequency

6.2 Design Fragments from Internet

In order to evaluate our design fragments, we next collected a set of thirty-six applets from the internet. Our goal was to collect applets that had not been created by Sun so we used the search string *import java.applet.Applet -site:sun.com*. We revised our search strings to ensure that the applets meaningfully engaged in the applet framework, either by using one of the lifecycle methods or event handling interfaces. To find threaded applets, we added *java.lang.Thread* to the search string; to find mouse listener applets, we added *java.awt.MouseListener*; to find parameterized applets, we added *getParameter*. We collected the first ten applets that matched each search string. As a result of our targeted search process, we were sure to get applets that used the features we searched for but our collection of applets no longer represented a neutral sampling of applets on the internet.

A concern about our process is that our searches preferentially targeted specific kinds of applets, specifically those using threads, engaging in event listening, and reading parameters. Internet searches indicate that 27% of applets use threads, 3.5% use events, and 18% use parameters.

From the internet applets we found an additional two design fragments: Focus Listener Applet and Background Continuous V1. The Focus Listener Applet design fragment is structurally identical to the other listener design fragments except that it listens for user interface focus changes. Background Continuous V1 and V2 are the same in intent and nearly identical in structure, differing only in a single check that occurs in the `run()` method. V1 checks that the thread field is not null while V2 checks that the thread field is equal to the currently running thread. Similarly to how the design fragments were defined in the previous section, the rules for identifying the Background Continuous V1 design fragment are: Must have a field holding reference to thread, Must create thread in `start()`, assigning field to thread, Must set field to null in `stop()`, Must implement `run()` and loop continuously until field is found to be null.

Table 5 at the end of this paper is a compilation of all of the applets analyzed and the design fragments that were found within them.

We tolerated some deviations from the ideal in matching the design fragments to the code. The most common deviation was a failure to de-register for events, which occurred in about two-thirds of the listening applets. Also common was the reading of parameters via `getParameter()` but a failure to publish those parameters in `getParameterInfo()`, which occurred in about a third of the Parameterized Applets. Non-conformance is detailed in Table 2. We note these deviations not as compelling evidence that design fragments can reduce bugs in code, but rather as evidence that even in debugged, released code it is possible to find incorrect usage of framework interfaces because of the difficult, non-local nature of engaging with a framework.

	Sun Applets	Internet Applets
Failure to define getParameterInfo()	0 / 13	12 / 17
Failure to de-register for events	3 / 16	26 / 27

Table 2. Non-conformance to design fragment

The design fragments we discovered fall into three categories: threading, event handling, and other. The threading design fragments deal with how to coordinate threads with the pre-determined applet method callbacks. The event handling design fragments deal with how to obtain additional events from the applet (and also AWT) framework. In the other category, Parameterized Applet deals with how to obtain the textual parameters that can be passed into an applet and how to report to users what parameters can be passed in. Manual Applet deals with how to provide a `main()` method that simulates the callback structure of an applet so that the applet can be invoked from the command line.

The three categories of design fragments are discussed in detail in the following three subsections.

6.3 Threaded Applets

Five of the design fragments had the purpose of coordinating a separate thread. Two of these are the Background Continuous variants that have been discussed as a running example through this paper, as in Figure 2. These threads are intended to run for a long time, usually the duration of the applet.

The One-Time Init Task design fragment uses a thread to perform some time-consuming startup task, such as establishing a connection with a server. This task is done in a background thread so as to keep the GUI responsive. This task is started in the `init()`

callback method. For the One-Time On-Demand Task design fragment, the only difference is that the background task can be started at any time during the running of the applet, such as when the user presses a key. It is possible to consider the former as a special case of the latter.

The Timed Task design fragment was only found in the Sun demo applets but could have been applied in many places where Background Continuous was used. Timed Task uses a Java Timer instead of a thread and the Timer can be set to run every so many milliseconds. The Clock2 applet from Figure 1 could have been written more simply and with less risk of a race condition had it used the Timed Task instead.

The applet framework never explicitly requires programmers to create new threads yet 27% of applets on the internet do. The applet framework constrains the solution space for programmers and they in turn have solved their problem using threads. It is interesting to note that no traditional technique for documenting interfaces, in this case framework interfaces, would instruct programmers to use threads.

6.4 Event Handling Applets

All of the event handling design fragments followed the structure shown in Figure 11, which shows the Mouse Listener design fragment. The programmer's code must implement the appropriate listener interface, in this case MouseEvent, and provide implementations for each of the required callback methods defined in that interface. In the `init()` callback method of the applet, the programmer's code calls the framework service method to register for callback events of this type, in this case `addMouseListener()`. A corresponding `removeMouseListener()` service method is provided for the applet to de-register for events and it should be called in the `destroy()` callback method on the applet.

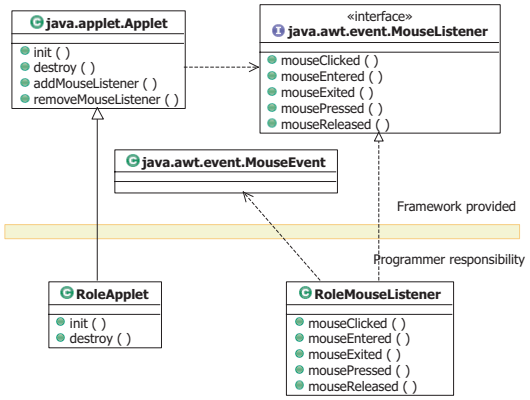

Figure 11. Mouse event handling structure

Many of the applets from the internet did not de-register for events. Since most applets respond to events their whole life, there is little harm in this since destruction of the applet happens just before the entire program terminates. It is present in the design fragment because the Sun applets, with few exceptions, de-registered for events and because it is probably good practice.

6.5 Parameterized Applets

Java applets are often run from web pages. It is possible to pass parameters into the applet via the HTML text, like: `<applet code=ArcTest.class width=400 height=400>`. In this case, the parameters `width` and `height` are passed in as strings with values of "400". The applet can read these parameters with the framework service method `getParameter(String name)`. It is also possible for an applet to let its users know what parameters they can pass in, and this is done with the non-lifecycle callback method `getParameterInfo()`, which returns an array containing the parameters and their expected types.

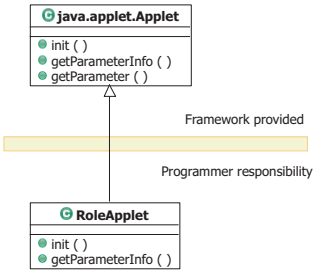

Figure 12. Parameter handling structure

Ideally, every applet that reads parameters would publish the fact that it reads them. Furthermore, the published list should match exactly the calls to `getParameter()` made by the applet. In practice it appears easy to break this non-local constraint since more than half of the applets from the internet had a mismatch between the parameters they queried and the parameters they published.

6.6 Similarities Between Sun and Internet Applets

Some of the applets from the internet shared structural features with the Sun applets. Specifically, some of the field names were identical. The threaded Sun applets used the following field names to hold a reference to the thread: *engine*, *kicker*, *killme, runner* (twice), and timer. The internet applets used: *engine*, *kicker*, *killme*, *runner* (four times), aniThread, artist, clock, _helloThread, loader, marcher, my_thread, Tetris1, and Tetris2. Note that the highlighted field names engine, runner, kicker, and killme are found in both the demo applets and the internet applets.

7. Analysis

The results of the case study support our hypotheses. Programmers did engage the applet framework in roughly the same way. Based on looking at their applets, it appears that programmers copied the structure of the Sun demos. We were able to create a catalog quickly. And, with a few exceptions, the design fragment language was able to express the patterns we found.

7.1 Hypothesis: Patterns Exist in Code

Our first hypothesis was that patterns exist in source code, specifically source code that interacts with frameworks. As seen in Table 5, across the 56 applets we found 108 design fragment instances. In order to conform to a design fragment, the program must adhere to the rules previously described.

Conformance was universal for the bureaucratic parts of design fragments dealing with event handling, such as implementing interfaces and providing method implementations. Universal conformance is not surprising because this bureaucracy is checked by the compiler and the code will never function correctly without it. Conformance was also universal for registering for events, but about 2 in 3 applets failed to de-register, including essentially all of the non-Sun applets. In most applets, no fault can be detected by a user because the applet and the rest of the Java environment are terminated at the same time. Note too that registering and de-registering must be implemented in two different callback methods.

In the Parameterized Applet design fragment, programmers were asked to keep two parts of their code consistent: asking for parameters and publishing which parameters they ask for. More than a third of the applets, and more than two thirds of the non-Sun ones, failed to publish their parameters. Parameters can be passed in to the applet even when those parameters are not published, so a user may not detect a fault. Note that checking for parameters is implemented in a different method than publishing the parameters.

We note that type-checked bureaucracy is correlated with high conformance. Low conformance is correlated with difficulty to observe faults through testing and a lack of tool-based conformance checking (such as type checking from the compiler).

A corollary to our hypothesis was that unnecessary diversity exists in source code. We found some evidence for this in the existence of two versions of the Background Continuous design fragment, but these versions were substantially similar. It is possible that our process of searching the internet for specific kinds of applets led to less diverse code, or that the applet framework is so simple that consistency is high.

7.2 Hypothesis: Programmers Reference Examples

Our second hypothesis was that programmers reference examples, specifically source code examples that appear to do the same thing they want their program to do. Two features of the applets from the internet support this hypothesis, both from the threaded applets category.

First, the replication of the threading bug in nine of the twelve applets that implemented the Background Continuous design fragment variants supports the notion that this bug was copied from the original demo applet bug.

Second, identical names of the thread fields strongly suggests that code was copied. The original Sun applets used names like *engine* and *runner* that could plausibly be independently re-created by other programmers. However, names like *kicker* and *killme* are also seen in both Sun and internet applets seem unlikely to have arisen independently by chance.

If the next generation of frameworks were to distribute its example applications with design fragments bound to the source code, then there is a good chance that this expression of design intent would also be copied by programmers and bug fixes like the race condition could be propagated to copied code.

7.3 Hypothesis: Effort to Create a Catalog Tapers

Our third hypothesis was that the effort to create a catalog tapers. After examining fifty-six applets we had found twelve design fragments for our catalog. Ten of these twelve were found in the initial twenty demo applets from Sun. One of the remaining design fragments was the buggy version of Background continuous and the other was the Focus Listener, which is structurally equivalent to the other event listening design fragments.

Our catalog was built by examining the applets alphabetically starting with the Sun demo applets. The growth of the catalog is plotted in Figure 13. We would like to see the rate of discovery of new design fragments slow down as more applets are analyzed, and this is what is shown in the chart. This result is not a quirk of the alphabetical order of evaluation since a glance at Table 5 reveals that there are paths that populate the catalog either more quickly or more slowly.

7.4 Hypothesis: Design Fragment Language Can Express Patterns

Our fourth hypothesis was that the design fragments language can express patterns, specifically the patterns that the authors have recognized in the code. The design fragment language did a good job of encoding the structure of the patterns with a few exceptions.

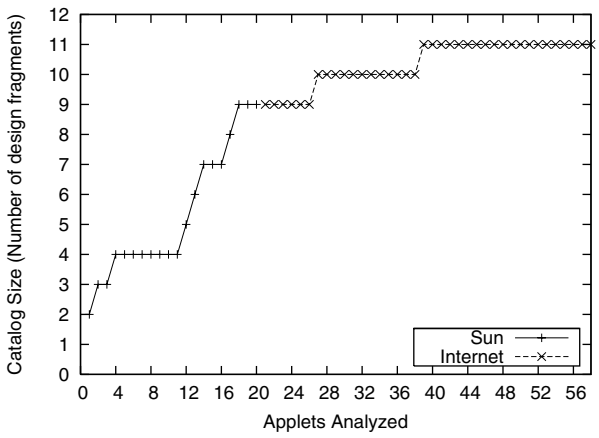

Figure 13. Catalog size

In the Parameterized Applet design fragment, the programmer should return an array of an array of string to publish the parameters that his applet checks. The strings are the parameters that are checked elsewhere in the program via calls to the framework service method `getParameter()`. Since the design fragment language cannot refer to the strings used as actual parameters to method invocations, nor can it refer to the strings in the array, it cannot express the constraint that these strings be paired.

The design fragment language also cannot express when the programmer may change the name of a role (e.g., class, method, or field) and when he should not. We have adopted the convention that role names beginning with "role" (e.g., `RoleApplet`) can be changed but other cannot, (e.g., `start()`).

The behavioral requirements of a pattern can only be minimally expressed in the design fragment language. The differences between a design fragment where the thread runs continuously versus one where the thread runs just until its task are completed must be expressed in natural language that cannot be checked by tools. Additionally, the expression of the order of framework method callbacks is only partial.

Despite these restrictions in expressiveness, we found that the interaction between the programmer's code and the framework could largely be expressed through the design fragments language.

8. Eclipse Framework

We created a catalog of design fragments for the Eclipse framework opportunistically, since we were creating a tool that uses the Eclipse framework. Unlike the applet case study, the design fragments we created have not been tested on a large number of example applications.

The Eclipse Java development environment allows programmers to build arbitrary Java programs, but also provides special tools to help them build extensions to the Eclipse environment itself. One of these special tools is a wizard that creates an example application. The programmer can choose features that he would like in the example application from a short list and the wizard generates a suitable application.

We created our design fragments from the generated example application since it, like the Sun demo applets, is intended to show programmers how to use the framework. We created a catalog containing the design fragments shown in Table 3. These design fragments were more complex than the applet design fragments. For example, a UML diagram of the Dynamic Right Click Menu design fragment structure is shown in Figure 14.

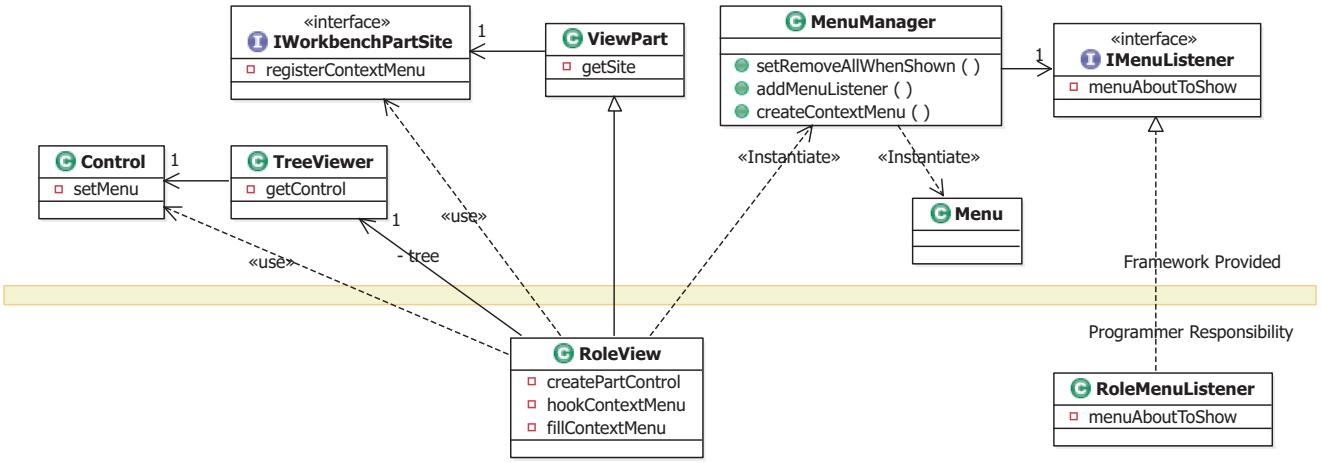

Figure 14. Eclipse dynamic right click menu structure

While using the Eclipse framework, we noticed that even after we had used one part of the framework extensively, for example the part for creating the user interface, this knowledge did not make us experts in other parts of the framework, for example the parts for incremental compilation. While the applet framework is sufficiently small that a programmer can comprehend it within a few hours, the Eclipse framework is far larger and more complex. With a large framework like Eclipse, knowing the structure of common solutions could speed up development time.

Design Fragment Name	Description
Content Provider	Data presented in a view
Label Provider	The labels for data in a view
Initializing Actions	Initial setup of the action objects
Resource Listener	Responding to resource changes
Dynamic Right Click Menu	A popup context menu
Toolbar Action	Ad an action to view's toolbar
Tree View Initialization	Initial setup of a tree view
Tree View Selection	User selecting item in tree view
View Sorter	A view that stays sorted

Table 3. Design Fragments from Eclipse

9. Related Work

This work builds upon previous work in a number of areas. Researchers in the 1980s and 1990s documented the use of software frameworks that they observed being used in industrial and academic settings. Frameworks were described as a new reuse mechanism that differed from class libraries. Ralph Johnson suggested natural language design patterns as a way for programmers to understand frameworks. Most design patterns are based on the ideas of role modeling, where a given class can play various roles and its responsibilities are the superset of the responsibilities of its roles. Research into design patterns led to tools that could model design patterns precisely and compare them with source code. The precise modeling of design patterns was modified to describe the programmers burden and instead of how the framework was implemented. Cookbooks and recipes followed a similar path starting from unstructured text through a precise representation.

9.1 Categorization of the Research

Table 4 provides a categorization of the work on framework documentation. It divides the work along two dimensions describing the style of documentation. The first dimension is whether the documentation provides examples or specifications. The second dimension is whether the documentation prescribes how clients should use the framework or describes the implementation of the framework.

	Prescribes how clients should use the framework	Describes the implementation of the framework
Specification-based	Helm Contracts FCL Constraints UML-F Profile Decl. Metaprogramming OOram Riehle frameworks JavaDoc	JavaDoc OOram Riehle fwks
Example-based	Decl. Metaprogramming Design Patterns Hooks JavaFrames Cookbooks JavaDoc *Design Fragments*	Design Patterns JavaFrames Utrecht Tool JavaDoc

Table 4. Categorization of the research

Relatively few projects have set out to describe the internals of frameworks and most of these projects were older, when frameworks were themselves just beginning to become popular. Most research has focused on documenting how to use the framework rather than documenting how the framework is designed. Techniques such as JavaDoc and Design Patterns can be applied to almost any documentation problem, and have been used to describe how frameworks work internally.

Techniques in the Example-based dimension do not claim to document every possible and correct use of the framework but

instead provide known-good examples. This simplifies the task both for the documentation author and for the reader.

Conversely, techniques in the Specification-based dimension claim to cover all correct use in much the same way that functional specifications for methods should cover all cases of inputs and outputs. A Microsoft Foundations Classes example from FCL Constraints constrains all subclasses of CWnd such that each must call one of three window creation methods defined in the framework.

9.2 Role Modeling

Object Oriented Role Analysis Modeling (OOram) is a software engineering method developed by Trygve Reenskaug that focuses on collaborating objects (role models) instead of classes [28]. Each role model consists of a number of roles with assigned behavior. Classes are created by composing these roles. A tool for the Smalltalk language was created for authoring and composing these role models. Reenskaug recalls Brad Cox's metaphor [2] of the surface area of components, that is, the things that must be understood about the component for a client to use it correctly, and applies it to frameworks. He notes that the surface area of a framework should be kept as small as possible, can be described with role models, and should not be changed for fear of breaking existing applications. Reenskaug is credited with the creation of the Model-View-Controller pattern, whose implementation in Smalltalk may be considered the earliest framework [20].

In his thesis [30], Dirk Riehle extends the role modeling concepts from OOram to treat frameworks as first-class concepts, calling it "role modeling for framework design." Role models describe the interface between the framework and the programmer's code; "free roles" represent the roles the programmer can implement to use the framework. Programmers should find frameworks with associated role models easier to comprehend since the complexity of the class models has been explained in terms of cross-cutting role models.

9.3 Precise Design Patterns, Code Ties

The Utrecht University design pattern tool tool [5], implemented in Smalltalk, allowed the creation of prototype-based design patterns and binding of these design patterns to source code. Conformance checking between the pattern and source code could be performed and predefined fixes could be used to repair non-conformance. Modeling focused on design patterns and application of the tool to frameworks was not specifically explored. Conformance checking was limited to static program structure while our analysis additionally supports behavior checking.

9.4 Frameworks

Confronting the challenge of communicating how to use the Model-View-Controller framework in Smalltalk-80, Krasner and Pope [22] constructed an 18 page cookbook that explained the purpose, structure, and implementation of the MVC framework. The cookbook begins with text but increasingly weaves in detailed code examples to explain how the framework could be used to solve problems. This cookbook was designed to be read from beginning to end by programmers and could also be used as a reference but every recipe did not follow a consistent structure nor was it suitable for parsing by automatic tools.

Ralph Johnson appears to have been the first to suggest documenting frameworks using patterns. He notes that the typical user of framework documentation wants to use the framework to solve typical problems [18] but also that cookbooks do not help the most advanced users [19]. Patterns can be used both to describe a framework's design as well as how it is commonly used. He argues that the framework documentation should describe the purpose of the framework, how to use the framework, and the detailed design of the framework. After presenting some graduate students with his initial set of patterns for HotDraw, he realized that a pattern isolated from examples is hard to comprehend.

Froehlich et al.'s Hooks focus on documenting the way a framework is used, not the design of the framework [7]. They are similar in intent to cookbook recipes but are more structured in their natural language. The elements listed are: name, requirement, type, area, uses, participants, changes, constraints, and comments. The instructions for framework users (the changes section) read a bit like pseudo code but are natural language and do not appear to be parsable by tools. Cookbook recipes, hooks, and design fragments are similar in that they all provide example-based descriptions of how to use a framework. Hooks added structure to recipes but were still natural language; design fragments regularize hooks to make them tool-readable and enable tool-based assurance.

Design patterns themselves can be decomposed into more primitive elements [27]. Pree calls these primitive elements metapatterns and catalogs several of them with example usage. He proposes a simple process for developing frameworks where identified points of variability are implemented with an appropriate metapattern, enabling the framework user to provide an appropriate implementation.

The declarative metaprogramming group from Vrije University [33, 32] uses Pree's metapatterns [27] to document framework hotspots and defines transformations for each framework and design pattern. Framework instances (plugins) can be evolved (or created) by application of the transformations. The tool uses SOUL, a prolog-like logic language. The validation was done on the HotDraw framework by specifying the metapatterns, patterns and transformations needed. The validation uncovered design flaws in HotDraw, despite its widespread use, along with some false positives. The declarative metaprogramming approach to modeling framework hotspots appears to have significant up-front investment before payoff in order to provide its guarantees about correct use of the framework. It may additionally assume a higher level of accuracy or correctness in frameworks than will commonly be found in practice.

In [33], the authors comment that their approach specifically avoids design patterns in favor of metapatterns because there could be many design patterns. While this makes their technique generally applicable and composable, it will be difficult to add pattern-specific semantics and behavior checking to their approach.

A UML profile is a restricted set of UML markup along with new notations and semantics [6]. The UML-F profile provides UML stereotypes and tags for annotating UML diagrams to encode framework constraints. Methods and attributes in both framework and user code can me marked up with boxes (grey, white, half-and-half, and a diagonal slash) that indicate the method/attribute's participation in superclass-defined template patterns. A grey box indicates newly defined or completely overridden superclass method, a white box indicates inherited and not redefined, a half-and-half indicates redefined but call to super(), and a slashed box indicates an abstract superclass method.

The Fixed, Adapt-static, and Adapt-dyn tags annotate the framework and constrain how users can subclass. Template and Hook tags annotate framework and user code to document template methods. Stereotypes for Pree's metapatterns (like unification and separation variants) are present, as are predefined tags for the Gang of Four [9] patterns. Recipes for framework use are presented in a format very similar to that of design patterns but there is no explicit representation of the solution versus the framework. The recipe encodes a list of steps for programmer to perform.

The FRamework EDitor / JavaFrames project [13, 12, 14] is a result of collaboration between The University of Tampere, the

University of Helsinki, and commercial partners starting in 1997. They have developed a language for modeling design patterns and tools that act as smarter cookbooks, guiding programmers step-by-step to use a framework. With the 2.0 release of JavaFrames, many of these tools work within the Eclipse IDE. Their language allows expression of structural constraints and the tool can check conformance with the structural constraints. Code can be generated that conforms to the pattern definition, optionally including default implementations of method bodies. Specific patterns can be related to general patterns; for example a specific use of the Observer pattern in a particular framework can be connected to a general definition of the Observer pattern.

The Framework Constraint Language (FCL) [16] applies the ideas from Richard Helms object oriented contracts [15] to frameworks. Like Riehle's role models, FCLs specify the interface between the framework and the user code such that the specification describes all legal uses of the framework. The researchers raise the metaphor of FCL as framework-specific typing rules and validate their approach by applying it to Microsoft Foundation Classes, historically one of the most widely used frameworks. The language has a number of built-in predicates and logical operators. It is designed to operate on the parse tree of the users code. Though targeted at plugin points, this language appears to be compatible with design fragments and could provide the basis of a richer constraint language for design fragments.

9.5 Aspects

Aspect oriented programming seeks to improve the modularity of source code by localizing programmer-chosen concerns into their own input files [31, 21, 25]. For example, the parts of a program that deal with logging could be extracted to a new source file so they do not clutter up the main code. Design fragments and aspects share a similar desire to decompose a program into smaller chunks. While design fragments are specifications, aspects are implementations. It may be possible to use aspects to provide default implementations for design fragments.

9.6 General Programming Assistance

The complexity of programming has long been recognized and attempts to help programmers manage that complexity have been researched. The Inscape Environment [26] focused on the challenges of evolution and scale in procedural programs. It addresses these challenges in part through specification of interfaces, much like design fragments. The specification language was deliberately impoverished in order to avoid the tar pit of verification, again much like the desire of design fragments to maintain simplicity in its language to encourage adoption.

The Programmers Apprentice [29] was an attempt to apply artificial intelligence to the problem of programming by providing an intelligent agent to support the programmer. Cohesive collections of program elements are bound together into a cliché, similar to a design fragment based on syntactic code structure, encoding roles and constraints. These clichés are used by the tool to aid the programmer.

9.7 Comparison of Design Fragments with Related Work

Design fragments are similar to JavaFrames in that both encode structural patterns that programmers use to engage with a framework. JavaFrames has been influenced by the cookbook approaches and provides automated tool support to ensure the recipe has been followed. Design fragments add a description of the relevant parts of the framework. This description begins to answer why the recipe or pattern works, which enables two things.

First, analysis tools that check for errors, such as the race condition described in this paper, can take advantage of the descriptions of the framework. Direct analysis of the framework code may be impossible because its source is unavailable or because it is too large.

Second, knowing why a recipe works enables the programmer to go beyond the recipe. Functionality demands from his problem domain will cause him to push on the limits of the pattern and he must be given an understanding of how this can be done. With exposure to many design fragments, the programmer will build up a mental model of how the framework acts on his code.

At first glance design fragments appear very similar to Riehle's role models. Following the categorization from section 9.1, role models are specification-based, while design fragments are example-based. As such, a role model strives to describe the complete and abstract protocol that every set of classes conforming to it must follow, while design fragments describe just a single legal use of that protocol. In addition to this fundamental difference, there are differences of focus and intent. First, design fragments often span multiple framework role models, such as in the Mouse Listener design fragment where the Applet callback methods are coordinated to invoke service methods to register for mouse events. Second, design fragments often encode actions outside of the framework, such as in the thread coordination design fragments. Third, design fragments are asymmetric, so they define only what the programmer must do and only provide a programmer-centric view of what the framework roles are doing. The intent of both techniques is to aid programmers in using frameworks and in practice their strengths are complimentary.

10. Conclusions and Future Work

Frameworks impose burdens on programmers and existing techniques do not provide adequate help to programmers. Design fragments are patterns that encode conventional solutions to how a program interacts with a framework to accomplish a goal. Since it is necessary to understand how the framework influences the programmer's code, design fragments encode the relevant parts of the framework. By declaring that his program uses a design fragment, a programmer is expressing enduring design intent that can be checked by analysis tools now and in the future.

Through a case study with fifty six applets, this paper has supported the hypotheses that code interacting with frameworks follows patterns, programmers copy example code, the effort to create a design fragment catalog is mostly up-front, and that the design fragment language is effective at expressing most of the patterns we observed.

In the future, we plan to evaluate design fragments on larger frameworks and involve more users. One area of particular interest is the evolution of design fragments as frameworks evolve, for example the evolution of the Eclipse framework through its 3.0, 3.1, and 3.2 versions. We plan to strengthen our ability to check conformance of the behavior specifications and broaden their scope.

Frameworks are increasingly relying upon declarative elements in addition to the object oriented mechanisms discussed here. Web frameworks such as Apache Struts define objects in Java but declare their relationship to the website flow in an XML file. Enterprise frameworks such as J2EE are increasingly relying upon Java annotations to encode characteristics of classes. We plan to identify what kinds of burdens these place on programmers and determine if design fragments are an appropriate technique to assist the programmer.

One issue we will work to overcome is the programmer's cost of expressing design intent versus the short-term benefit. It is possible that an improved user interface could reduce the expression cost. For example, it seems reasonable to instantiate design fragments with help from a wizard.

11. Acknowledgements

This work and paper were greatly improved through helpful comments from Bradley Schmerl, the members of the SSSG seminar, and the anonymous reviewers. This research was sponsored by the US Army Research Office (ARO) under grant number DAAD19-01-1-0485, the National Science Foundation under grant CCR-0113810, and a research contract from Lockheed Martin ATL. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the ARO, the U.S. government, Lockheed Martin ATL or any other entity.

References

[1] Kent Beck and Donald G. Firesmith. *Kent Beck's Guide to Better Smalltalk : A Sorted Collection (SIGS Reference Library)*. Cambridge University Press, 1998.

[2] Brad Cox. *Object-Oriented Programming, An Evolutionary Approach*. Addison Wesley, New York, 1987.

[3] The Debian Linux Distribution. http://www.debian.org.

[4] D. Fay. An architecture for distributed applications on the internet: Overview of microsoft's .NET platform. *IEEE International Parallel and Distributed Processing Symposium*, April 2003.

[5] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495, Jyvaskyla, Finland, June 1997.

[6] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley Professional, 2001.

[7] Gary Froehlich, H. James Hoover, Ling Liu, and Paul Sorenson. Hooking into object-oriented application frameworks. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 491–501, New York, NY, USA, 1997.

[8] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Professional, 2003.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 1995.

[10] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, 2000.

[11] Aaron Greenhouse, T.J. Halloran, and William L. Scherlis. Observations on the assured evolution of concurrent java programs. *Science of Computer Programming*, 58:384–411, March 2005.

[12] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Annotating reusable software architectures with specialization patterns. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 171, Washington, DC, USA, 2001.

[13] Imed Hammouda and Kai Koskimies. A pattern-based j2ee application development environment. *Nordic Journal of Computing*, 9(3):248–260, 2002.

[14] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002.

[15] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 169–180, New York, NY, USA, 1990. ACM Press.

[16] Daqing Hou and H. James Hoover. Towards specifying constraints for object-oriented frameworks. In *CASCON '01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 2001.

[17] Daqing Hou, Kenny Wong, and H. James Hoover. What can programmer questions tell us about frameworks? In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 87–96, Washington, DC, USA, 2005.

[18] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76, New York, NY, USA, 1992.

[19] Ralph E. Johnson. Components, frameworks, patterns. *SIGSOFT Softw. Eng. Notes*, 22(3):10–17, 1997.

[20] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001.

[22] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[23] Sun Microsystems. Java applets. http://java.sun.com/applets/.

[24] Richard Monson-Haefel. *Enterprise JavaBeans (3rd Edition)*. O'Reilly, 2001.

[25] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proc. 7th IBM Conf. Object-Oriented Technology*, July 1994.

[26] Dewayne E. Perry. The inscape environment. In *ICSE '89: Proceedings of the 11th International Conference on Software Engineering*, pages 2–11, New York, NY, USA, 1989.

[27] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley Longman, 1994.

[28] Trygve Reenskaug, P. Wold, O. A. Lehne, and Manning. *Working With Objects: The Ooram Software Engineering Method*. Manning Pubns Co, 1995.

[29] Charles Rich and Richard. C. Waters. The programmer's apprentice: A research overview. In D. Partridge, editor, *Artificial Intelligence & Software Engineering*, pages 155–182. Norwood, NJ, 1991.

[30] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2000.

[31] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999.

[32] Tom Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 2002.

[33] Tom Tourwé and Tom Mens. Automated support for framework-based software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 148, Washington, DC, USA, 2003.

Source	DF Name	AppletWithThreadV0	AppletWithThreadV1	AppletWithThreadV2	AppletWithThreadV3	AppletWithTimer	ComponentListener	FocusListener	KeyListener	ManualApplet	MouseListener	MouseMotion	ParameterizedApple
Sun demo	animator		1								1		1
Sun demo	arctest									1			
Sun demo	barchart												1
Sun demo	blink					1							1
Sun demo	cardtest									1			
Sun demo	clock		1										1
Sun demo	dithertest		1							1			1
Sun demo	drawtest									1			
Sun demo	fractal		1								1		1
Sun demo	graphicstest									1			
Sun demo	graphlayout												1
Sun demo	imagemap		1								1	1	1
Sun demo	jumpingbox						1				1	1	
Sun demo	moleculeviewer			1							1	1	1
Sun demo	nervoustext		1								1		1
Sun demo	simplegraph												
Sun demo	sortdemo				1						1		1
Sun demo	spreadsheet								1		1		1
Sun demo	tictactoe										1		
Sun demo	wireframe			1							1	1	1
Internet	anbutton		1										1
Internet	antacross		1								1	1	
Internet	antmarch		1								1	1	
Internet	blinkhello				1								
Internet	brokeredchat				1					1			1
Internet	bsom												1
Internet	buttontest	1								1			
Internet	cte												
Internet	demographics										1	1	
Internet	dotproduct						1				1	1	
Internet	envelope												
Internet	fireworks	1											1
Internet	gammabutton												1
Internet	geometry												1
Internet	hellotcl	1								1			1
Internet	hyperbolic										1	1	
Internet	iagtager											1	
Internet	inspect	1											
Internet	kbdfocus							1	1		1		
Internet	lagttager												
Internet	linprog										1	1	
Internet	mousedemo										1	1	
Internet	myapplet												
Internet	myapplet2									1	1	1	1
Internet	nickcam												1
Internet	notprolog									1			
Internet	scatter												1
Internet	scope	1											1
Internet	simplepong												1
Internet	simplesun	1											1
Internet	smtp												1
Internet	superapplet										1	1	
Internet	tetris	2											
Internet	ungrateful												1
Internet	urccalendar												1
Internet	urlexample				1								1
Internet	webstart	1							1		1	1	
Internet	ympyra										1		
	Total	9	9	2	4	1	2	1	3	10	22	15	30

Table 5. All applets with origin and counts of design fragments