Patterns Generate Architectures

Kent Beck -- First Class Software, Inc.

Ralph Johnson -- University of Illinois at Urbana-Champaign

Abstract

We need ways to describe designs that communicate the reasons for our design decisions, not just the results. Design patterns have been proposed as ways of communicating design information. This paper shows that patterns can be used to derive an architecture from its problem statement. The resulting description makes it easier to understand the purpose of the various architectural features.

Introduction

Design is hard. One way to avoid the act of design is to reuse existing designs. But reusing designs requires learning them, or at least some parts of them, and communicating complex designs is hard, too. One reason for this is that existing design notations focus on communicating the "what" of designs, but almost completely ignore the "why". However, the "why" of a design is crucial for customizing it to a particular problem. We need ways of describing designs that communicate the reasons for our design decisions, not just the results.

One approach to improving design, currently receiving interest primarily outside the object community, is the idea of "architecture" [Garlan93]. An architecture is the way the parts work together to make the whole. The way architectures are notated, applied, and discovered are all topics of active research.

A closely related idea inside the object community is that of "framework" [Deutsch89] [Johnson88]. A framework is the reusable design of a system or a part of a system expressed as a set of abstract classes and the way instances of (subclasses of) those classes collaborate. Frameworks are a particular way of representing architectures, so there are architectures that can't be expressed as frameworks. Nevertheless, the two ideas overlap. Both are attempts to reuse design, and examples of one are sometimes used as examples of the other.

Another approach to design in the object community is "patterns". There were OOPSLA workshops in 1991 and 1992 on "an architecture handbook" [Anderson93] and ones in 1992 and 1993 on "patterns"[Coad93], with an overlap between the two groups, which shows a link between architectures and patterns. Much of the work on patterns focuses on

---

Authors' addresses: Kent Beck, First Class Software, Inc., P.O. Box 226, Boulder Creek CA 96006 e-mail: 70761.1216@compuserve.com, Ralph Johnson, Department of Computer Science, 1304 W. Springfield Ave., Urbana IL 61801 e-mail: johnson @cs.uiuc.edu

To be presented at ECOOP'94

patterns of relationships between objects as the building-blocks of larger architectures [Coad92] [Gamma93].

Our original interest in patterns[Kerth88] was sparked by the work of an architect, Christopher Alexander, whose patterns encode knowledge of the design and construction of communities and buildings [Alexander77] [Alexander79]. His use of the word "pattern" takes on more meaning than the usual dictionary definition. Alexander's patterns are both a description of a recurring pattern of architectural elements and a rule for how and when to create that pattern. They are the recurring decisions made by experts, written so that those less skilled can use them. They describe more of the "why" of design than a simple description of a set of relationships between objects.

We call patterns like Alexander's that describe when a pattern should be applied "generative patterns". Generative patterns share the many advantages of non generative patterns; they provide a language for designers that makes it easier to plan, talk about, and document designs. They have the added advantage of being easier for non-experts to use and providing a rationale for a design after the fact. This paper focuses on the last advantage.

This paper shows that patterns can be used to derive architectures, much as a mathematical theorem can be proved from a set of axioms. When patterns are used in this way, they illuminate and motivate architectures. Deriving an architecture from patterns records the design decisions that were made, and why they were made that way. This makes it easier to modify the architecture if circumstances change.

Patterns

Two kinds of patterns are needed to derive HotDraw: object-oriented design patterns and graphics patterns. This is probably typical of most architectures; some patterns will be generic and some will be specific to the application domain.

Each pattern follows this format:

- Preconditions–The patterns that must be satisfied before this one is valid. The sequence in which patterns are considered is one of the most important skills possessed by experts.
- Problem--A summary of the problem addressed by the pattern. The problem statement is used by the reader to decide if the pattern is applicable.
- Constraints–The constraints describe the conflicting (sometimes mutually exclusive) forces acting on any solution to the problem. Typical examples are tradeoffs between execution time and execution space, or development time and program complexity. Clearly stating priorities between constraints makes patterns easy to debate.
- Solution–A two or three sentence summary of the solution to the problem. The solution is often accompanied by a diagram illustrating the activity required to transform a system from one that doesn't satisfy the pattern to one that does.

The graphics patterns are Model-View-Controller, Collect Damage, and Update at User Speed. All of the other patterns except Editor are in the Design Pattern Catalog[Gamma94].

The versions of the patterns in this paper differ from the versions in the Catalog in two ways. The most obvious difference is that the versions in this paper are much shorter, but that is out of necessity, not preference. A fully expressed pattern contains at least two or three pages of discussion of constraints and an illustration or example showing how it works. Another difference is that the versions in this paper are more generative that the versions in the Catalog. In other words, we emphasize the conditions under which the pattern applies, the transformation each causes in the design as it comes into existence. The patterns in the Catalog do not entirely ignore when they are applicable, since the *intent* is similar to the problem in our patterns. However, they often have a list of possible causes, and in general focus more on the solution and its variants than on when to use the solution. Although we have only rewritten a few of the patterns in the Catalog to be more generative, we have no reason to believe that all of them couldn't be.

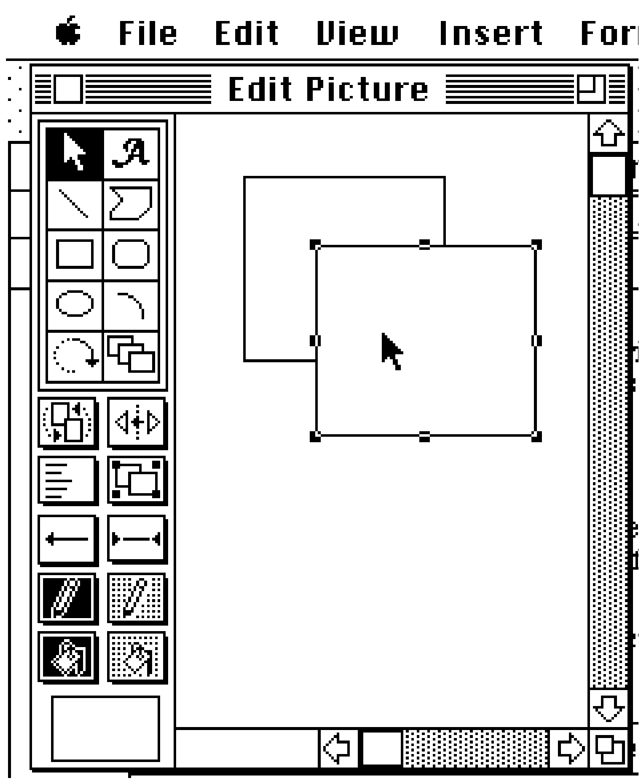
An example of a generative pattern for object-oriented software is "Objects for States", which says that you can transform an object which changes behavior depending on state into two objects, the first of which has the invariant parts of the original object, and a family of objects which it can refer to, one for each state in the original object.

HotDraw

The architecture that is derived is that of HotDraw, a framework for structured graphics editors. A previous paper presented a set of patterns for using HotDraw[Johnson92]. In contrast, this paper describes the patterns that create HotDraw. Patterns can be used at many levels, and what is derived at one level can be considered a basic pattern at another level.

HotDraw provides a reusable architecture for direct-manipulation, graphics-editor-like applications that is implemented in Smalltalk-80. (Versions have been implemented for other environments; since the only pattern that depends on the environment is the one that chooses a user-interface framework, and since there is a lot of commonality among user-interface frameworks, all the versions are similar.) It supports:

- many kinds of figures in the drawing
- a programmable palette of tools
- different handles for different figures
- smooth animation



A typical graphics editor with a palette, figures, and handles on figures.

This paper describes HotDraw from the top down, much like the proof of a theorem. It gives reasons for the design decisions in HotDraw, which, as Parnas says, are not always the original reasons for these design decisions [Parnas86]. Thus, the derivation of HotDraw is a rationalization of HotDraw, and is only partly related to the history of its design.

The purpose of this derivation, however, is not to show people how HotDraw was developed, but to let them understand it. Although people usually start to use a framework by modifying examples and recombining components in different ways, experts need to have a deeper understanding of the framework. We believe that the derivation reflects the understanding that an expert has.

Deriving Hot Draw

Describing an architecture with patterns is like the process of cell division and specialization that drives growth in biological organisms. The design starts as a fuzzy cloud representing the system to be realized. As patterns are applied to the cloud, parts of it come into focus. When no more patterns are applicable, the design is finished.

Each step of the following discussion will briefly describe what problem needs to be solved, the pattern that is used to solve it, and the effect the pattern has on the design.

User interface

The first problem to solve is getting the drawing on the screen so it can be displayed and manipulated. The following pattern tells us that we need to divide the responsibilities between three objects.

Model-View-Controller[Krasner88]

Preconditions A system is going to have a graphical user-interface.

Problem Graphical user-interfaces can be hard to build. Users demand programs that are easy to use, easy to learn, and powerful, and a good user interface is necessary to achieve these goals. How should the responsibilities of implementing a user interface be divided between objects?

Constraints Modern graphical user interfaces have a small number of recurring visual elements. Because user-interfaces need to be consistent, we depend on a few interaction techniques, such as menus, buttons, scroll-bars, and lists. The effort that someone puts into learning how to use one program, or one part of one program, should apply to other programs and other parts of the same program.

In implementing a user interface, we must strike a balance between a design that uses many objects but is difficult to learn and one that uses few objects and sacrifices flexibility. One important axis of flexibility is the information that is displayed. A second axis of flexibility, independent of display, is interpreting user gestures and mapping them into state changes. A third degree of freedom is the ability to put multiple user interfaces on the same information.

Therefore:

Solution Divide your system into three objects: a Model, View and Controller. The Model is responsible for maintaining state and surfacing the behavior necessary to support the user interface. The View is responsible for displaying an up-to-date version of the Model. The Controller is responsible for mapping user gestures to changes of state in the model.

Applying Model-View-Controller to our system we derive three objects: Drawing, to hold the drawing, DrawingView, to display it, and DrawingController to parse user gestures into changes to the drawing.

Many elements in a drawing

Next we notice that Drawings need to contain many figures. This leads us to the Composite pattern.

Composite

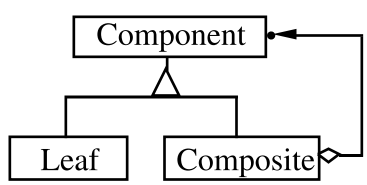
Preconditions A composite object is an object with many components; a folder consists of a set of documents, a book consists of a set of chapters, a company consists of a set of departments.

Problem A naive modeling of composite objects leads to many similar classes specialized primarily by the types of their components.

Constraints Consider a text editing system with "chapter", "section", "subsection", "paragraph", "word", "item" etc. We could have a large number of classes, each of which was really just a collection of its components. Some have titles, some always start a new page, but in general they are quite similar. On the one hand, the composite object is "obviously" different from its components. On the other hand, it is easier to reuse and understand a design that minimizes the number of classes by making its classes be polymorphic and composable in any combination. We don't want to make every "text-element" have a list of components; some will be atomic. But we would like a scheme that lets us avoid having to duplicate code and to design a lot of classes.

Therefore:

Solution Make an abstract class that represents both the composite and its components. Then make the classes of the composite and the components be its concrete subclasses. The subclasses have to implement all the operations defined in the superclass. Some operations are implemented in every use of the Composite pattern, such as "iterate over components", which will do nothing for leaf classes, but will have a more complicated implementation for true composite classes. But most operations are application specific.



The Three Classes in Composite

Applying Composite to Drawing creates two new classes: Figure, for the generic thing in a drawing, and PrimitiveFigure for figures that contain no sub-parts. Drawing takes the role of the branching part of the composite.

Notice that using Composite here implies that we can nest Drawings within Drawings.

Interpreting input

Which tool is selected in the palette changes how input is parsed. Applying Objects for States to DrawingController solves this problem.

Objects for States

Preconditions An object whose behavior depends on its state.

Problem It is common for an object whose behavior depends on its state to explicitly check the value of its variables using *if* or *case* statements. Although this technique can work well when there are only few states, it produces programs that are hard to understand and extend. It also introduces a multiple update problem in maintenance, where all cases must be updated in parallel to preserve correct behavior.

Constraints On the one hand, encoding dependence on state by *if* statements makes a design compact; there are fewer objects to understand and fewer procedures to track down. On the other hand, it can be hard to tell which states are important, and hard to tell when a state transition occurs, since an object's states are determined by the set of values some of its variables can take on, and a state transition is an assignment statement. Moreover, adding a new state might require changing all of the object's methods. If an object has only two or three states, and it is unlikely to need a new state, then it is probably better to encode state dependency directly in methods. Otherwise, states should be represented as objects.

Therefore:

Solution Implement an object whose behavior depends on its state and that has more than a couple of states, or when the set of possible states is likely to change in the future, by representing the states by objects and having the object act as a "multiplexor", delegating messages to its current state.

A design that uses "encode state in object" can be transformed into one that uses "represent states as objects" by the following 5 steps:

1) Enumerate the possible states of the object.

2) Make a class for each state.

3) Give object an instance variable that contains its current state.

4) If a method of the object depends on its state, move it to the state-classes, and replace it with a method that simply delegates the operation to the current state. Add the object as an argument to the message that it delegates so that the state-classes can access its instance variables.

5) Move a method to the state-classes by copying it to every class, changing every instance variable access to use the original object (which has been passed as an argument to the message), and deleting all the parts of the method that are for other states.

The result of this transformation is that there will be a new class for each state of the original object, and much of the code in the class of the original object will have been moved to the classes of the states. You can add a new state by adding a new class, but the code for the object is now spread among several classes.

You can often use Factor a Superclass (not described here) to reduce the size of the code after applying Objects for States.

Each state object will be a Tool. The DrawingController will have a new attribute to hold the current tool.

Create the object to help DrawingController

Where are Tools created? DrawingController's responsibility is only to manage which Tool is current and delegate input parsing to the currentTool. However, different kinds of drawings will need different sets of Tools.

Editor

Preconditions You have a collection of dissimilar objects, probably after applying Composite.

Problem Often an object in a collection will have a specialized operation that you want to invoke, but most objects in the collection will not support that operation. Should we have an abstract superclass that supports all operations and let specialized operations produce errors for most subclasses? Should we provide a way for clients to determine whether an object has a particular operation?

For example, consider a collection of vehicles. They will all support "turn left" and "turn right" operations, but only a few (e.g. submarines, helicopters) will have "go up" and "go down" operations. Should we have the superclass Vehicle support al operations and let "go up" and "go down" produce errors for most subclasses? Should we provide a way for clients to determine whether a vehicle is a submarine and hence whether it supports "go up"?

Constraints Systems are hard to understand if they have many classes, each with a specialized interface. A good design should minimize the number of different interfaces that you have to learn. On the other hand, the problem we are working on often requires that different objects support different operations, so we are forced to have different interfaces. We can either try to design a very general interface and force-fit many classes to it, try to hide some of the interfaces(reducing the apparent complexity, if not the actual complexity), or just live with a complicated system.

If each class has one client that uses its specialized interface, then the Editor pattern can be used to give all of them the same interface.

Solution Hide the specialized interfaces by making an object (an editor) that represents the client of the object, and making each object responsible for producing its editor. Instead of using a specialized interface directly, you ask the object for its editor and then invoke the editor. The editor will be the only object that knows the private interface of the specialized object. The name "Editor" comes from the use of this pattern in a hypermedia system, where following a link to an object will always invoke an editor for that object.

It is important that there is a standard interface to the editor. Often the editor is a user interface object, and it will have a standard user interface object interface.

Example: Given a set of vehicles, where some can go up and down, but others can only go right or left, give each vehicle a "controls" operation that returns a set of controls, such as steering wheels, knobs, and dials. The user then has operations like "turn right" and "push" that operate on these controls, which will in turn control the direction of the vehicle. The only controls for a vehicle like a car will be for turning left and right and for controlling velocity, but a submarine will also have controls for going up and down.

We have an object (DrawingController) that manipulates an object (Drawing) through objects with standard protocol (Tools). This is half the Editor pattern. We can finish the pattern by giving Drawing the responsibility for returning a standard set of Tools.

Update the display when a Figure changes appearance

How can we be sure that changes to the drawing are reflected on the screen?

Observer

Preconditions Two objects must be synchronized. Changes in one object must be reflected in the other.

Problem If changing one object requires changing another then there is a constraint between them. But we don't want every client to have to know about this constraint. Moreover, we aren't even sure how many objects need to be changed when the first object is changed. Different instances of the same class might have different numbers of dependents, or the dependents might change over time.

Constraints To make objects as reusable as possible, we do not want to hard-code constraints into them. Objects should be responsible only for their own state, not for the state of other objects. On the other hand, if there is a constraint between the states of two objects, then that constraint must be recorded somewhere. In some way, changes to the first object must be translated into changes to the second.

Therefore:

Solution Have an object involved in a constraint keep a list of dependents. Each time it changes, it notifies all its dependents. When an object is notified that something it depends on has changed, it takes appropriate action. In general, an object can have many dependents, and a change to the object will require only some of its dependents to change. But each dependent will be able to determine whether the change to the object was significant and can ignore those that are not.

Appling Observer to Figure, we add each Figure's enclosing Drawing as a dependent, and the DrawingView as a dependent of the top-most Drawing.

Notice that we could have made the relationship between a Figure and its Drawing one-to-one (calling the Drawing the Figure's parent, for instance). Having done this, though, if we ever wanted to keep the Figure and some other object synchronized, we would have to duplicate code to update both the parent and another object.

Only redisplay changed part of the drawing

If several parts of the drawing change simultaneously (from the user's perspective), then we would like those parts of the drawing to update as a unit.

Collect Damage

Preconditions A program has an internal representation of an picture. It periodically changes part of the picture, and then must update the display to correspond to the internal representation.

Problem To be efficient, a program should display as little of the picture as possible. How do you display just the part of the picture that changed?

Constraints Each time you change part of the picture, you could display the part that changed. This would be the simplest solution. However, there is a large constant overhead for redisplaying, as potentially the entire representation of the picture must be traversed. For efficiency, you would like to only redisplay once, even if you several parts of the picture changed.

Another issue is that the user expects a single action (for example, changing the color of a certain kind of Figure) to result in a single redisplay. Updating a little at a time gives the impression that the system is slow.

Therefore:

Solution Have the graphics system keep track of the part of the image that has changed. Associate a *damaged region* with the image. Every time a part of the image changes, add its area to the damaged region. Adding, deleting, or changing the color of a part of the image will add its area to the damaged region. Moving a part of the image will add both its old and new areas to the damaged region. This can usually be done automatically by the graphics system, assuming that the graphical elements that make up the image are considered part of the graphics system, and not the application program.

Note that we haven't said how redisplay is initiated, just that it will take place only in the damaged region.

Applying Collect Damage to DrawingView adds an attribute damagedRegion, and applying it to Figure causes it to broadcast a message before and after all appearance changes.

In a language like Smalltalk with anonymous functions, you can embed the "damaging" idiom in a method "damageAround: aBlock".

Initiate the redisplay

Now that we are collecting damaged regions, we need some way to make sure they get redisplayed.

Update at User Speed

Preconditions You are writing a program that is animating a visual display in real-time, probably in response to user input. You are Collecting Damage.

Problem When should you update the display?

Constraints If you update the display too often, your application will spend all its time performing low-level graphics operations, and it may not be able to keep up with the animation. If you don't update the display often enough, the animation will be jerky.

A possible solution to this problem is to spawn an independent thread of control to update the display. By varying the rate at which it cycles, you can tradeoff the amount of system resources required to keep the display consistent and the smoothness of updates. However, this solution requires introducing and maintaining a monitor on the top-most Drawing, to avoid redisplaying it while it is in an inconsistent state.

Solution Update the drawing at the same speed that the user makes gestures. This implies that you redisplay the drawing once for each user event, which implies that there should be a loop in the program that reads an event, handles the event, and then updates the drawing (by repairing damage). If events come in faster than your program can handle them then you should not redisplay between them.

Add responsibility to DrawingController to tell the DrawingView to redisplay the damaged region after routing each event to the current Tool.

This pattern is different than the convention of many editors, which drag outlines around (which as a Pattern might be called Intermediate Changes to Outlines). It gives a more immediate feel to the application.

Create handles

The next problem to solve is where to create handles. Different Figures will require different handles, but the enclosing Drawing should be insulated from these differences.

The Editor pattern solves this problem. Applying Editor to Figure creates a Handle object. Each Figure must be responsible for returning a set of Handles that modify it.

Since it must be displayable in the Drawing, Handle can be simply implemented by making it a subclass of Figure.

Manage handles

Once we have Handles, we need to manage them. They need to be managed together, since selecting a different Figure causes all of the previous Handles to be discarded. They exist in a limbo between the Drawing and the DrawingView. We cannot manage them in the Drawing, since Drawings nest and we would not know at which level to store them. On the other hand, the DrawingView already is busy collecting damage and displaying the Drawing. We need somewhere new to put the Handles.

Wrapper

Preconditions We have a set of classes that can be composed in different ways to make an application. We probably use Composite and Observer so that we can make many kinds of systems out of a fairly static set of parts.

Problem We want to add a responsibility to an object, but we don't want to have to make a lot more classes or make it more complicated.

Constraints Adding the responsibility in a subclass would ensure that the old class was not made more complicated. However, we might want to reuse that responsibility for another object. Multiple inheritance would let us statically assign it to another object, but multiple inheritance can lead to an explosion of classes, and many languages do not support it. Moreover, multiple inheritance does not support dynamically reassigning a responsibility to another object.

Solution One way to add a responsibility to an object is with a wrapper. A T-wrapper "wraps" an instance of T and supports the T protocol by forwarding operations to the T, handling directly only those operations that relate to the responsibility it is designed to fulfill.

Applying Wrapper to Drawing creates a new object, SelectionDrawing. SelectionDrawing has protocol to hide and show sets of handles. All other messages it passes through to its Drawing.

SelectionDrawing can be implemented simply by making it a subclass of Drawing that has two Figures. The first is a Drawing that will contain all of the Handles. The second is the Drawing to be wrapped.

Conclusion

The HotDraw architecture is not magic, but is the logical result of a set of design patterns. In the past, we have explained the architecture as "Drawing, Figure, Tool, and Handle". The pattern-based derivation puts each of these classes in perspective. It explains exactly why each was created and what problem it solves. Presented this way, HotDraw becomes much easier to re-implement, or to modify should circumstances so warrant. This is a completely different approach to describing the design of a framework than more formal approaches like Contracts[Helm90] The more formal results only explain what the design is, but a pattern-based derivation explains why.

We didn't choose these patterns by chance. We had been talking about patterns for a long time, but usually tried to defend an action we were taking in terms of some pattern we knew. This was the first time we had tried to explain an existing design in terms of the sequence of design decisions that led to it. It was not hard to come up with the general sequence, though it took work to make the pattern descriptions clear even though they are short, and we did not completely succeed.

When we attempted to derive HotDraw from the patterns described in the Design Pattern Catalog, we immediately realized that the Catalog had none of the graphics patterns that HotDraw would need. It turned out that it did not have the Editor pattern, either, so the derivation of HotDraw showed us that we needed to describe a new object-oriented design pattern. This is similar to the proof process in mathematics, where the presentation of a proof hides most of its history, and where advances in mathematics are often caused by break-downs in proofs. Catalogs of design patterns will mature as people try to explain designs in terms of patterns, and find patterns that are missing from the catalogs.

A pattern-based derivation of an architecture is like the presentation of a mathematical theorem. Beginning with a problem to solve, you use well-known, independent steps to incrementally refine the problem to a solution. The result is that you not only understand the final system, you understand the reasoning that led to it. This should make it easier for programmers to use and extend systems documented with pattern-based derivations.

Bibliography

[Alexander77] Christopher Alexander, Sara Ishikawa and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[Alexander79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[Anderson93] Bruce Anderson. Workshop Report--Towards an Architecture Handbook. OOPSLA'92: Addendum to the Proceedings, printed as OOPSLA Messenger, 4(2): 109-114, April 1993

[Coad92] Peter Coad, "Object-Oriented Patterns", *Communications of the ACM,* 35(9):153-159, 1992.

[Coad93] Peter Coad and Mark Mayfield. Workshop Report--Patterns. OOPSLA'92: Addendum to the Proceedings, printed as OOPSLA Messenger, 4(2): 93-95, April 1993

[Deutsch89] L. Peter Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 Programming System", pp 55-71, *Software Reusability, Vol II*, ed. Ted J. Biggerstaff and Alan J.Perlis, ACM Press, 1989.

[Gamma93] Erich Gamma, Richard Helm, Ralph Johnson,and John Vlissides, "Design patterns: Abstraction and reuse of object-oriented design". In *European Conference on Object-oriented Programming*, Kaiserlauten, German, July 1993. Published as Lecture notes in Computer Science #707, pp. 406-431, Springer-Verlag.

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson,and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture* Addison-Wesley, 1994.

[Garlan93] David Garlan and Mary Shaw, "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering* Volume I, World Scientific Publishing Company, 1993.

[Johnson88] Ralph E. Johnson and Brian Foote,"Designing Reusable Classes" *Journal of Object-Oriented Programming,* 1(2):22-25, 1988.

[Johnson92] Ralph E. Johnson, "Documenting Frameworks with Patterns" *OOPSLA '92 Proceedings*, SIGPLAN Notices, 27(10): 63-76,Vancouver BC, October 1992.

[Kerth88] Norman Kerth, John Hogg, Lynn Stein, and Harry Porter, "Summary of Discussions from OOPSLA-87's Methodology and OOP Workshop", OOPSLA'87: Addendum to the Proceedings, printed as SIGPLAN Notices, 23(5), pp. 9-16, 1988.

[Krasner88] Glenn E. Krasner and Stephen T. Pope,"A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming,* 1(3):26-49, 1988.

[Helm90] Richard Helm and Ian M. Holland and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", *OOPSLA '90 Proceedings*, SIGPLAN Notices, 25(10), pp.169-180,Vancouver BC, October 1990.

[Parnas86] David L. Parnas and P.C. Clements "A Rational Design Process: How and Why to Fake It", *IEEE Transactions on Software Engineering, SE-12:2 February 1986.*