CSci 658: Software Language Engineering
Introduction to Patterns

H. Conrad Cunningham

17 February 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of February 2018 is a recent version of Firefox from Mozilla.

Slides: Introduction to Patterns (HTML slides)

Introduction to Patterns

These notes discuss "design patterns" mostly from the perspective of object-oriented programming using languages such as Java. Similar approaches may be used in functional languages such as Haskell, but often functional languages will use first-class and higher-order functions to express the patterns.

The classic works on design patterns are the "Gang of Four" [Gamma 1995] and the "Siemens" [Buschmann 1996] books. These notes use the terminology of the Siemens book.

What is a Pattern?

When experts need to solve a problem, they seldom invent a totally new solution. More often they will recall a similar problem they have solved previously and reuse the essential aspects of the old solution to solve the new problem. They tend to think in problem-solution pairs.

Identifying the essential aspects of specific problem-solution pairs leads to descriptions of problem-solving *patterns* that can be reused.

The concept of a pattern as used in software architecture is borrowed from the field of (building) architecture, in particular from the writings of architect Christopher Alexander.

Definition: "A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." [Buschmann 1996]

Where software architecture is concerned, the concept of a pattern described here is essentially the same concept as an *architectural style* or *architectural idiom* in the Shaw and Garlan book [Shaw 1996].

In general, patterns have the following characteristics [Buschmann 1996]:

- A pattern describes a solution to a recurring problem that arises in specific design situations.
- Patterns are not invented; they are distilled from practical experience.
- Patterns describe a group of components (e.g., classes or objects), how the components interact, and the responsibilities of each component. That is, they are higher level abstractions than classes or objects.
- Patterns provide a vocabulary for communication among designers. The choice of a name for a pattern is very important.

- Patterns help document the architectural vision of a design. If the vision is clearly understood, it will less likely be violated when the system is modified.
- Patterns provide a conceptual skeleton for a solution to a design problem and, hence, encourage the construction of software with well-defined properties.
- Patterns are building blocks for the construction of more complex designs.
- Patterns help designers manage the complexity of the software. When a recurring pattern is identified, the corresponding general solution can be implemented productively to provide a reliable software system.

Descriptions of Patterns

Various authors use different formats (i.e., "languages") for describing patterns. Typically a pattern will be described with a schema that includes at least the following three parts [Buschmann 1996]:

1. Context
2. Problem
3. Solution

Context

The *Context* section describes the situation in which the design problem arises.

Problem

The *Problem* section describes the problem that arises repeatedly in the context.

In particular, the description describes the set of *forces* repeatedly arising in the context. A force is some aspect of the problem that must be considered when attempting a solution. Example types of forces include:

- requirements the solution must satisfy (e.g., efficiency)
- constraints that must be considered (e.g., use of a certain algorithm or protocol)
- desirable properties of a solution (e.g., easy to modify)

Forces may complementary (i.e., can be achieved simultaneously) or contradictory (i.e., can only be balanced).

Solution

The *Solution* section describes a proven solution to the problem.

The solution specifies a configuration of elements to balance the forces associated with the problem.

- A pattern describes the static structure of the configuration, identifying the components and the connectors (i.e., the relationships among the components).
- A pattern also describes the dynamic runtime behavior of the configuration, identifying the control structure of the components and connectors.

Categories of Patterns

Patterns can be grouped into three categories according to their level of abstraction [Buschmann 1996]:

1. Architectural patterns
2. Design patterns
3. Idioms

Architectural patterns

Shaw and Garlan normally use the term *architectural style* for the architectural pattern concept described here.

Definition: "An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them." [Buschmann 1996]

An architectural pattern is a high-level abstraction. The choice of the architectural pattern to be used is a fundamental design decision in the development of a software system. It determines the system-wide structure and constrains the design choices available for the various subsystems. It is, in general, independent of the implementation language to be used.

An example of an architectural pattern is the Pipes and Filters pattern (slides). In Unix for instance, a filter is a program that reads a stream of bytes from its standard input and writes a transformed stream to its standard output. These programs can be chained together with the output of one filter becoming the input of the next filter in the sequence via the pipe mechanism. Larger systems can thus be constructed from simple components that otherwise operate independently of one another.

Other example architectural patterns are Layered systems, Blackboards, and the Model-View-Controller pattern for graphical user interfaces (such as in Web applications).

Design patterns

Definition: "A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context." [Buschmann 1996]

A design pattern is a mid-level abstraction. The choice of a design pattern does not affect the fundamental structure of the software system, but it does affect the structure of a subsystem. Like the architectural pattern, the design pattern tends to be independent of the implementation language to be used.

Examples of design patterns include the following:

- *Adapter* (or Wrapper) pattern.

  This pattern adapts the interface of one existing type of object to have the same interface as a different existing type of object.

  For example, a class might adapt the built-in Java `Vector` class to provide the operations of a "stack" class and hide the non-stack features of the `Vector`.

  The paper "Creating Applications from Components: A Manufacturing Framework Design" [Schmid 1997] also gives an example of how the Adapter pattern can be used in an application framework. The paper shows that a portal robot machine class can be adapted for use as a transport service class.

- *Iterator* pattern.

  This pattern defines mechanisms for stepping through container data structures element by element. Iterator objects for standard collections are now common in most programming language libraries. Programmers can implement iterators for their own custom collections.

- *Strategy* (or Policy) pattern (slides).

  The goal of this pattern is to allow any one of a family of related algorithms to be easily substituted in a system.

  We also see this pattern used in the Schmid article [Schmid 1996]; the third transformation involved breaking up the application logic class `ProcessingControl` into several subclasses of a new `ProcessingStrategy` class. The specific processing strategy could then be selected dynamically based on the specific part-processing task.

Idioms

Definition: "An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language." [Buschmann 1996]

An idiom is a low-level abstraction. It is usually a language-specific pattern that deals with some aspects of both design and implementation.

In some sense, use of a consistent program coding and formatting style can be considered an idiom for the language being used. Such a style would provide guidelines for naming variables, laying out declarations, indenting control structures, ordering the features of a class, determining how values are returned, and so forth. A good style that is used consistently makes a program easier to understand than otherwise would be the case.

In Java, the language-specific iterator defined to implement the `Iterator` interface can be considered an idiom. It is a language-specific instance of the more general Iterator design pattern.

Another example of an idiom is the use of the Counted Pointer (or Counted Body or Reference Counting) technique for storage management of shared objects in C++. In this idiom, we control access to a shared object through two classes, a *Body* (representation) class and a *Handle* (access) class.

An object of the *Body* class holds the shared object and a count of the number of references to the object.

An object of a *Handle* class holds a direct reference to a body object; all other parts of the program must access the body indirectly through handle class methods. The handle methods can increment the reference count when a new reference is created and decrement the count when a reference is freed. When a reference count goes to zero, the shared object and its body can be deleted. Often the programmer using this pattern will want to override the `operator->` of the handle class to give more transparent access to the shared object.

A variant of the Counted Pointer idiom can be used to implement a "copy on write" mechanism. That is, the body is shared as long as only "read" access is needed, but a copy is created whenever one of the holders makes a change to the state of the object.

Acknowledgements

In Spring 2017, I adapted these notes from my previous notes on the topic. I wrote the first version during Spring 1998 for my Software Architecture course based, in part, on:

- Chapter 1 of the book *Pattern -Oriented Software Architecture : A System of Patterns* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal (Wiley, 1996).
- Chapter 2 of the book *Software Architecture: Perspectives on an Emerging Discipline* by Mary Shaw and David Garlan (Prentice-Hall, 1996).

In Spring 2018 I revised the notes slightly to fit in with the other documents for the CSci 658 course.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

References

[**Buschmann 1996**] – **"Siemens" book** Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern -Oriented Software Architecture : A System of Patterns*, Wiley, 1996.

[**Gamma 1995**] – **"Gang of Four" (GoF) book** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995. (A percursor paper is "Design Patterns: Abstraction and Reuse of Object-Oriented Design".)

[**Grand 1998**] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, Volume 1, Wiley, 1998.

[**Schmid 1996**] Hans Albrecht Schmid. Creating Applications from Components: A Manufacturing Framework Design, *IEEE Software*, Nov. 1996.

[**Schmid 1997**] Hans Albrecht Schmid. Systematic Framework Design by Generalization, *Communications of the ACM*, Vol. 40, No. 10, pp. 48-51, 1997.

[**Shaw 1996**] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

Concepts

TODO:

Pattern (for software architecture), architectural pattern (or architecural style), design pattern, idiom. Pattern context, problem, and solution. Forces. Pipes and Filters, Blackboard, Model-View-Controller architectural patterns. Adapter, Iterator, Strategy design patterns. Counted Pointer (or Reference Counting, Handle-Bokdy) idiom.