Lecture 10: Chapter 12

- Pattern-Based Design

*Slide Set to accompany*

*Software Engineering: A Practitioner’s Approach, 7/e*

*by Roger S. Pressman*


For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.
Design Patterns

- Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution to for this?*
  - What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem?
- Design Patterns is the solution.
Design Patterns

- Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice.
  - Christopher Alexander, 1977
- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.
Basic Concepts

- **Context** allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.

- A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how
  - the problem can be interpreted within its context and
  - how the solution can be effectively applied.
Effective Patterns

- Coplien [Cop05] characterizes an effective design pattern in the following way:
  - *It solves a specific problem*: Patterns capture solutions, not just abstract principles or strategies.
  - *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
  - *The solution is not obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
  - *It describes a relationship*: Patterns do not just describe modules, but describe deeper system structures and mechanisms.
  - *The pattern minimizes human intervention*: All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.
Kinds of Patterns

- **Architectural patterns** describe broad-based design problems that are solved using a structural approach.
- **Data patterns** describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.
- **Component patterns** (also referred to as design patterns) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture.
- **Interface design patterns** describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end-users.
- **WebApp patterns** address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned.
Pattern Elements

- **Name**
  - A meaningful pattern identifier.

- **Problem description.**

- **Solution description.**
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.

- **Consequences**
  - The results and trade-offs of applying the pattern.
Example (component pattern): The Observer Pattern

- **Name**
  - Observer

- **Description**
  - Separates the display of object state from the object itself.

- **Problem description**
  - Used when multiple displays of state are needed.

- **Solution description**
  - See slide with UML description.

- **Consequences**
  - Optimisations to enhance display performance are impractical.
The Observer Pattern (1)

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.</td>
</tr>
<tr>
<td>Problem description</td>
<td>In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.</td>
</tr>
</tbody>
</table>
## The Observer Pattern (2)

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solution description</strong></td>
<td>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObserver, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.</td>
</tr>
<tr>
<td><strong>Consequences</strong></td>
<td>The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.</td>
</tr>
</tbody>
</table>
Multiple displays using the Observer pattern
A UML model of the Observer pattern

Subject
- Attach (Observer)
- Detach (Observer)
- Notify ()

ConcreteSubject
- GetState ()
- subjectState

Observer
- Update ()

ConcreteObserver
- Update ()
- observerState

observerState = subject -> GetState ()

for all o in observers
  o -> Update ()
Design problems

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.

- Tell several objects that the state of some other object has changed (Observer pattern).
- Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
- Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
- Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).
Pattern-Based Design
Design Tasks—I

- Examine the requirements model and develop a problem hierarchy.
- Determine if a reliable pattern language has been developed for the problem domain.
- Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
- Using the collaborations provided for the architectural pattern, examine subsystem or component level problems and search for appropriate patterns to address them.
- Repeat steps 2 through 5 until all broad problems have been addressed.
Design Tasks—II

- If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.
- Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.
- Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.
Common Design Mistakes

- Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence, you select a pattern that looks right, but is inappropriate for the solution required.
- Once the wrong pattern is selected, you refuse to see your error and force fit the pattern.
- In other cases, the problem has forces that are not considered by the pattern you’ve chosen, resulting in a poor or erroneous fit.
- Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.
Architectural Patterns

- Example: every house (and every architectural style for houses) employs a **Kitchen** pattern.
- The **Kitchen** pattern addresses problems associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room.
- In addition, the pattern might address problems associated with counter tops, lighting, wall switches, a central island, flooring, and so on.
- Obviously, there is more than a single design for a kitchen, often dictated by the context and system of forces. But every design can be conceived within the context of the ‘solution’ suggested by the **Kitchen** pattern.
Component-Level Patterns

- Component-level design patterns provide a proven solution that addresses one or more sub-problems extracted from the requirement model.
- In many cases, design patterns of this type focus on some functional element of a system.
User Interface (UI) Patterns

- **Whole UI.** Provide design guidance for top-level structure and navigation throughout the entire interface.

- **Page layout.** Address the general organization of pages (for Websites) or distinct screen displays (for interactive applications).

- **Forms and input.** Consider a variety of design techniques for completing form-level input.

- **Tables.** Provide design guidance for creating and manipulating tabular data of all kinds.

- **Direct data manipulation.** Address data editing, modification, and transformation.

- **Navigation.** Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.

- **Searching.** Enable content-specific searches through information maintained within a Web site or contained by persistent data stores that are accessible via an interactive application.

- **Page elements.** Implement specific elements of a Web page or display screen.

- **E-commerce.** Specific to Web sites, these patterns implement recurring elements of e-commerce applications.
WebApp Patterns

- **Information architecture patterns** relate to the overall structure of the information space, and the ways in which users will interact with the information.

- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.

- **Interaction patterns** contribute to the design of the user interface. Patterns in this category address how the interface informs the user of the consequences of a specific action; how a user expands content based on usage context and user desires; how to best describe the destination that is implied by a link; how to inform the user about the status of an on-going interaction, and interface related issues.

- **Presentation patterns** assist in the presentation of content as it is presented to the user via the interface. Patterns in this category address how to organize user interface control functions for better usability; how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.

- **Functional patterns** define the workflows, behaviors, processing, communications, and other algorithmic elements within a WebApp.
Patterns Repositories

- There are many sources for design patterns available on the Web.
- Some patterns can be obtained from individually published pattern languages, while others are available as part of a patterns portal or patterns repository.
- A list of patterns repositories is presented in the sidebar near Section 12.3