SWEN 262

Engineering of Software Subsystems
Anatomy of a Pattern

What are design patterns?

• A pattern is a general solution to a problem in context.
  ○ general - only provides an outline of the approach
  ○ problem - some recurring issue
  ○ context - the specific system being designed, and the expected design evolution

• Patterns are not code. They are generic recipes that may be followed to create a solution to a specific problem.

Each pattern describes a problem which 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 this solution a million times over, without ever doing it the same way twice.

Christopher Alexander is considered to be “the father of pattern language.”
Anatomy of a Pattern

Patterns allow us to gain from the experience and mistakes of others.

- Design for reuse is difficult.
- Experienced designers:
  - Rarely start from first principles.
  - Apply a working “handbook” of approaches.
- Patterns make this experiential knowledge available to all.
- Patterns also help evaluation of alternatives at a higher level of abstraction.

Patterns are not invented. They are discovered.

Over time, experienced engineers learn the “best practice” for solving a specific recurring problem.

This is a process of trial, error, and the testing of a set of alternatives.

Eventually a good solution is found and documented for others to use.

Sometimes a better solution is discovered later.
Pattern Intent

The most important piece of information about a pattern is **intent**.

- The intent provides a general indication of when a pattern is appropriate.
  - What is the nature of the problem or problems that the pattern is meant to solve?
  - In what kind of environment or system is it appropriate to use the pattern?

Some intentions may be familiar sounding...

“Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.”

...while others will not.

“Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”
Pattern Classification

The design patterns in the Gang-of-Four text are mainly classified according to the purpose of the pattern’s intent.

- **Creational** – The intent is mainly about creating objects.
- **Structural** – The intent is mainly about the structural relationship between objects.
- **Behavioral** – The intent is mainly about the interactions between objects.
Binding Time

A second dimension of classification is binding time.

- Using inheritance is compile-time (early) binding.
  - Class-based.
- Using delegation or composition is run-time (late) binding.
  - Object-based.

- Creational
  - class: defer creation to subclasses.
  - object: defer creation to another object (delegate).

- Structural
  - class: structure via inheritance
  - object: structure via composition

- Behavioral
  - class: algorithms/control via inheritance.
  - object: algorithms/control via object groups.
To apply a pattern you need to understand:

● **Structure**
  ○ The static class relationships between elements of the pattern.

● **Participants**
  ○ Each class/object in the pattern
  ○ The responsibilities of each class/object

● **Collaborations**
  ○ The general description of interactions between participants
  ○ The sequence diagram defining interactions

Note that GoF structure notation is OMT (pre-UML). See Appendix B for a guide on the notation.
**Consequences**

Every pattern has a set of associated *consequences* that describe the nuances of pattern usage including:

- How does the structure support the intent of the pattern?
- What are the trade-offs in pattern usage?
- Where are the variation points?

Consequences may include both benefits and potential drawbacks.

- Makes it easier to add new kinds of components.
- Can make the design overly general.

Beware of force fitting a pattern into a problem that it is not suited for.

Like any other tool, a pattern can become a *golden hammer*. 
Implementation Details

Implementation details for a specific pattern may vary from one language to the next.

Different languages may have language-specific:
- Pitfalls to avoid when implementing the pattern.
- Hints and techniques for applying the pattern.
- Design choices.

Being that it is a pure Object-Oriented language, implementing patterns in Java is straightforward.

In SWEN 262, your team is free to choose a framework in which to work, as long as the language in which you implement your patterns is full object-oriented.
Documenting Pattern Structure

- Every GoF Pattern has a class structure diagram.
  - Each participant in the pattern is documented as a class or an interface.
  - The diagrams in the GoF text are not standard UML.

- When you document your patterns, you will create a UML class diagram.
  - Use context-specific class names - names appropriate to the system that you are designing in context, i.e. drawn from your domain analysis.
  - Each class that plays a role in the pattern will have its pattern stereotype in <<guillemets>> beneath its name.
GoF Pattern Cards

As we’ve mentioned before, documentation is a very important part of your grade in this course.

Properly documenting your design, including UML class and sequence diagrams is important.

Just as important is specifying your rationale for major design decisions: why you made the choices that you made.

- What were the benefits?
- What were the trade offs? Why were they acceptable?

Frequently these design decisions include implementing a pattern in your architecture.

GoF patterns should be documented using a GoF Pattern card.

<table>
<thead>
<tr>
<th>Name: Image Receiver System</th>
<th>GoF Pattern: Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participants (don’t write anything here)</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Role in Pattern</td>
</tr>
<tr>
<td>Image Receiver</td>
<td>Subject, Concrete Subject</td>
</tr>
<tr>
<td>Image Processor</td>
<td>Observer</td>
</tr>
<tr>
<td>DICOM Image Processor</td>
<td>Concrete Observer</td>
</tr>
<tr>
<td>ACR Image Processor</td>
<td>Concrete Observer</td>
</tr>
</tbody>
</table>

Deviations from the standard pattern: The notify method on each concrete observer returns a boolean. Observers are called in the order in which they are registered. If no observer returns true, indicating that the image was handled, no additional observers are notified (the notification process is short circuited).

Requirements being covered: 1 (interface with medical imaging devices, support multiple image formats), 2 (accept images from imaging devices), 3 (store images in DICOM format).

Let’s take a detailed look at a GoF pattern card example...
# GoF Pattern Cards

<table>
<thead>
<tr>
<th>Name:</th>
<th>GoF Pattern:</th>
</tr>
</thead>
</table>

## Participants

<table>
<thead>
<tr>
<th>Class</th>
<th>Role in Pattern</th>
<th>Participant’s Contribution in the context of the application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Deviations from the standard pattern:

## Requirements being covered:
GoF Pattern Cards

Name: [ ] GoF Pattern: [ ]

<table>
<thead>
<tr>
<th>Class</th>
<th>Role in Pattern</th>
<th>Participant’s Contribution in the context of the application</th>
</tr>
</thead>
</table>

Deviations from the standard pattern:

Requirements being covered:

- The name of the subsystem in your architecture.
- The name of your GoF pattern being implemented.
- The name of each class in your subsystem that implements part of the pattern.
- The name of each role that each of your classes plays in the GoF pattern.
- Any changes that you made to the standard pattern, and why.
- The requirements (name and number) that your implementation is satisfying.

* A detailed description (i.e. at least 2-3 sentences) of how each of your classes contributes to the pattern in context.
<table>
<thead>
<tr>
<th>Class</th>
<th>Role in Pattern</th>
<th>Participant’s Contribution in the context of the application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Receiver</td>
<td>Subject, Concrete</td>
<td>A service that provides a network API used by external imaging devices to upload newly captured images into the system. Notifies observers upon receipt of new images.</td>
</tr>
<tr>
<td>Image Processor</td>
<td>Observer</td>
<td>Interface implemented by observers that wish to be notified when new images arrive.</td>
</tr>
<tr>
<td>DCOM Image Processor</td>
<td>Concrete Observer</td>
<td>Registers to be notified when images arrive. If the images are in the DICOM format it transfers the images and updates the database.</td>
</tr>
<tr>
<td>ACR Image Processor</td>
<td>Concrete Observer</td>
<td>Registers to be notified when images arrive. If the images are in ACR-NEMA format, the image is translated to the DICOM format. Extends DCOM Image Processor.</td>
</tr>
</tbody>
</table>

**Deviations from the standard pattern:** The notify method on each concrete observer returns a boolean. Observers are called in the order in which they are registered. If an observer returns “true,” indicating that the image was handled, no additional observers are notified (the notification process is short circuited).

**Requirements being covered:** 1 (interface with medical imaging devices, support multiple image formats), 2 (accept images from imaging devices), 3 (store images in DICOM format)