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."

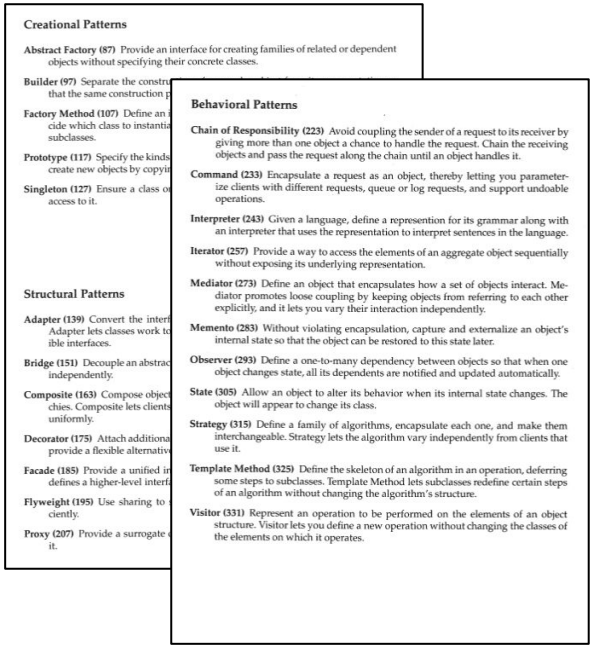
...while others will not.

"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.



Creational Patterns

Abstract Factory (87) Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder (97) Separate the constru… that the same construction p…

Factory Method (107) Define an … cide which class to instantia… subclasses.

Prototype (117) Specify the kinds… create new objects by copyi…

Singleton (127) Ensure a class o… access to it.

Structural Patterns

Adapter (139) Convert the interf… Adapter lets classes work to… ible interfaces.

Bridge (151) Decouple an abstrac… independently.

Composite (163) Compose object… chies. Composite lets clients… uniformly.

Decorator (175) Attach additiona… provide a flexible alternativ…

Facade (185) Provide a unified in… defines a higher-level interf…

Flyweight (195) Use sharing to … ciently.

Proxy (207) Provide a surrogate … it.

Behavioral Patterns

Chain of Responsibility (223) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command (233) Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter (243) Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator (257) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator (273) Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento (283) Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer (293) Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

You can see which patterns are in which categories at a glance by referring to the list on the inside front cover of the book.

Binding Time

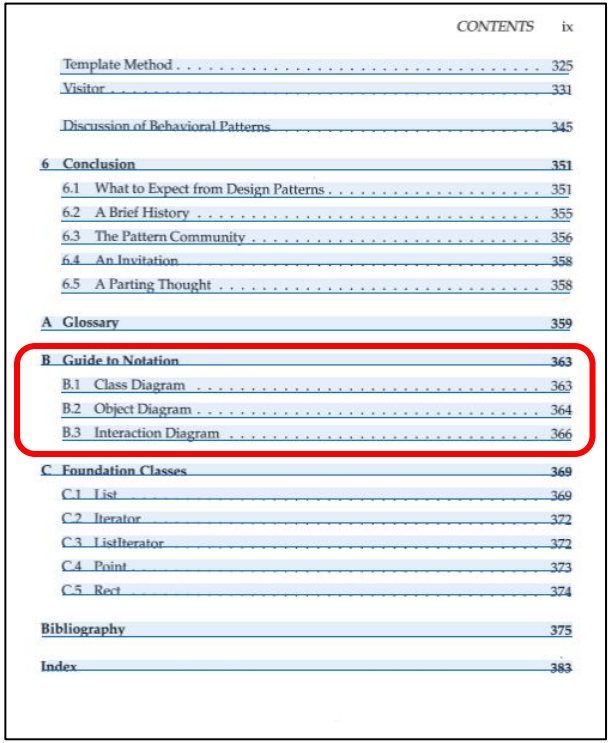
A second dimension of classification is *binding time*.

- 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.*

Pattern Application

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.

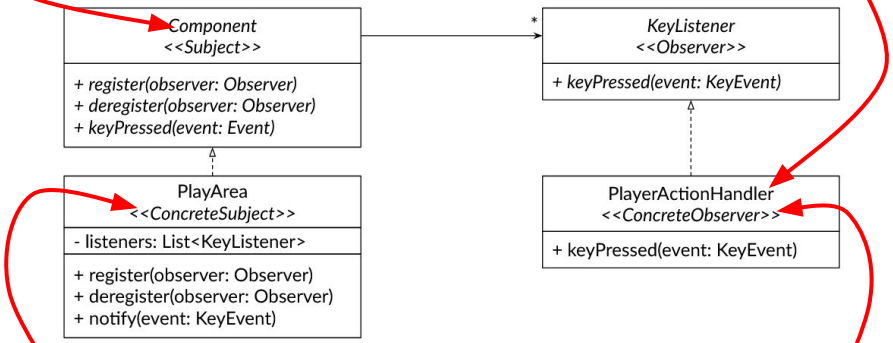
Being that it is a pure Object-Oriented language, implementing patterns in Java is straight forward.

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.*

Your pattern diagrams should use standard UML notation with class names that fit the ***context*** of your application.



But each class that is a participant in your pattern implementation should be clearly identified with the name of its role in *<<guillemets>>* below the class 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.

Name: *Image Receiver System*		GoF Pattern: *Observer*
Participants *(don't write anything here)*		
Class	Role in Pattern	Participant's Contribution in the context of the application
Image Receiver	*Subject, Concrete Subject*	*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.*
Image Processor	*Observer*	*Interface implemented by observers that wish to be notified when new images arrive.*
DCOM Image Processor	*Concrete Observer*	*Registers to be notified when images arrive. If the images are in the DICOM format it transfers the images and updates the database.*
ACR Image Processor	*Concrete Observer*	*Registers to be notified when images arrive. If the images are in ACR-NEMA format, the image is translated to the DICOM format. Extends DICOM Image Processor.*
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)*		

Let's take a detailed look at a GoF pattern card example...

GoF Pattern Cards

Name:		GoF Pattern:
Participants		
Class	Role in Pattern	Participant's Contribution in the context of the application
Deviations from the standard pattern:		
Requirements being covered:		

GoF Pattern Cards

The name of the subsystem in *your* architecture.

The name of the GoF pattern being implemented.

Name:	GoF Pattern:

Participants

Class	Role in Pattern	Participant's Contribution in the context of the application

The name of the role that each of *your* classes plays in the GoF pattern.

The name of **each class** in ***your subsystem*** that implements part of the pattern.

A **detailed** description (i.e. ***at least*** 2-3 sentences) of how each of *your* classes contributes to the pattern in **context**.

Any changes that you made to the standard pattern, and why.

Deviations from the standard pattern:

The requirements (name and number) that your implementation is satisfying.

Requirements being covered:

GoF Pattern Cards

Name: *Image Receiver System*		GoF Pattern: *Observer*
Participants *(don't write anything here)*		
Class	Role in Pattern	Participant's Contribution in the context of the application
Image Receiver	*Subject, Concrete Subject*	*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.*
Image Processor	*Observer*	*Interface implemented by observers that wish to be notified when new images arrive.*
DCOM Image Processor	*Concrete Observer*	*Registers to be notified when images arrive. If the images are in the DICOM format it transfers the images and updates the database.*
ACR Image Processor	*Concrete Observer*	*Registers to be notified when images arrive. If the images are in ACR-NEMA format, the image is translated to the DICOM format. Extends DICOM Image Processor.*
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)*		