(GoF) Design Patterns

The Gamma, Helm, Johnson, and Vlissides (GoF) book is the “bible” of design patterns: it catalogues some of the most common and useful OO idioms

- This lecture will just be a quick refresher and warm-up
- We will cover some very fundamental patterns, just to have a common vocabulary
  - in later lectures I will freely say things like “this is done with an Abstract Factory” or “a Singleton pattern takes care of this”
- There are tons of material on the web—these slides are just quick notes
- `http://www.cs.wustl.edu/~schmidt/tutorials-patterns.html`
- `http://www.objenv.com/cetus/oo_patterns.html`

Overview

A design pattern is an abstract pattern occurring in the course of designing an OO system

Why catalogue patterns?

- not having to reinvent them
- common vocabulary => better communication, documentation

Ok, but what makes something a “pattern”? Why isn’t every single idiom a pattern?

- resistance to change!
- design patterns are design “fixpoints”
  - they represent the state of a mature system, after requirements have changed many times and the system architecture has reached a steady state

Catalogue

The catalogue has a strict format

- Pattern name (e.g., Observer) and classification (creational/structural/ behavioral)
- Intent
- Alternate names
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Creational Patterns: Abstract Factory

- For creating families of related objects without specifying which
- Example:
  - Window with subclasses PMWindow, MotifWindow
  - Scrollbar with subclasses PMScrollbar, MotifScrollbar
  - client is oblivious to actual window used, employs an abstract factory to “CreateScrollBar”, “CreateWindow”
  - concrete factories are subclasses of the abstract factory

Factory Method

- For defining an interface for object creation, but letting subclasses decide which class to instantiate. Example:
  - a generic library of abstract classes (an *application framework*) is used as the basis of client-server applications
  - applications create documents (`Document` is an abstract class in the framework), the application class (also an abstract class in the framework) has an abstract `CreateDocument` method
  - other application code in the framework refers to this method and manipulates the documents it returns
  - concrete applications define the `CreateDocument` method and create the right concrete `Document` (instance of a subclass of `Document`)

Singleton

- For ensuring that a class only has one instance
- Example implementation:
  - the class cannot be instantiated externally (e.g., protected constructor)
  - a static method (“class method”) is used to access the single instance of the class and create it the first time
- Advantages over a class with only static members and methods
  - singleton means “at most one”, not “exactly one”
  - object identity (can use the object as a key, for instance)
  - avoid issues of order of static initialization
  - able to inherit methods from a non-singleton class, or implement interface (i.e., able to do dynamic dispatch)

Structural Patterns: Bridge (aka Envelope-Letter)

- For decoupling an abstraction from its implementation (so they can be extended independently)
- Example:
  - A window may be subclassed across two different axes of variability: windowing toolkit (e.g., XWindows, PMWindows) and implementation (e.g., TextWindow, GraphicalWindow)
  - Avoid the combinatorial blowup by separating the `Window` abstract class from the `WindowImpl` abstract class
  - A window holds a reference to a `WindowImpl`

Behavioral Patterns: Command

- For encoding actions as objects so they can be recorded, logged, modified at runtime (e.g., context sensitive menus) etc.

- Example:
  - A graphical application may have several different commands in a menu
  - Instead of calling a method for each command, register a “command” object (instance of a subclass of an abstract `Command` class)
  - Each command object supports an “execute” method

- An OO way to do *callbacks*

Observer (aka Publish-Subscribe)

- For registering objects and notifying them when events occur
- Example:
  - A class `Model` models data to be displayed graphically. The data may be presented in multiple views. Each view is a subclass of a `View` abstract class, which defines an `update` method
  - `Model` has methods to dynamically `register` and `unregister` views
  - When the data change, all registered views are notified (their `update` is called)
- Very common pattern for GUIs (MFC, Smalltalk MVC)

Visitor

- For encoding operations as independent entities, without distributing them throughout the classes they are applicable on
  - useful for adding functionality without editing classes
- Typically, in OO designs if an “operation” is applicable to multiple types (classes), it is defined as a method in all the corresponding classes. With visitor, the “operation” can be in a class by itself
- Example:
  - A compiler has `TypeCheck`, `CodeGen`, etc. operations with different implementations for declarations, statements, etc.
  - It is easier to organize code with these operations as separate entities

- Also easier to add new operations
- Each class (`Declaration`, `Statement`, etc.) defines an `Accept` method that takes a visitor and passes `this` as a parameter to the visitor's `Visit` method (or `VisitDeclaration`, `VisitAssignment`, etc., method if no overloading)
- The visitor defines the operation (e.g., typecheck) and is an instance of a subclass of the `Visitor` abstract class (which defines `Visit`, or `VisitAssignment`, etc.)

- Visitor is a way to structure code by operation instead of by type of data
- "Programming functionally in an OO language"

Comments

"*There is no problem in computer programming that cannot be solved with one extra level of indirection*" (Anonymous)

"*There are run-time inefficiencies, but the human inefficiencies are more important in the long run*" (GoF)

- There are other correct designs, but they may not withstand change so well
- Design patterns add indirection to help anticipate change in the system