Introduction to Design Patterns

Origins of Patterns in Architecture

- C. Alexander: problem of objective quality
  - by making observations of buildings, towns, streets, gardens,
    - he discovered that high quality constructs had things in common
    - architectural structures differed from each others, even it they were of the same type solving the same problem. Yet different solutions were of high quality.
- Conclusion: structures could not be separated from the problem they are solving
  - ...so he looked at different structures yielding a high quality solution to same problem and extracted the core of the solution, i.e. the **patterns**.
- Alexanders patterns
  - solutions to a problem in a context
  - 253 patterns covering regions, towns, transportations, homes offices, rooms, lighting, gardens, ...
  - a generative pattern language

Design Patterns

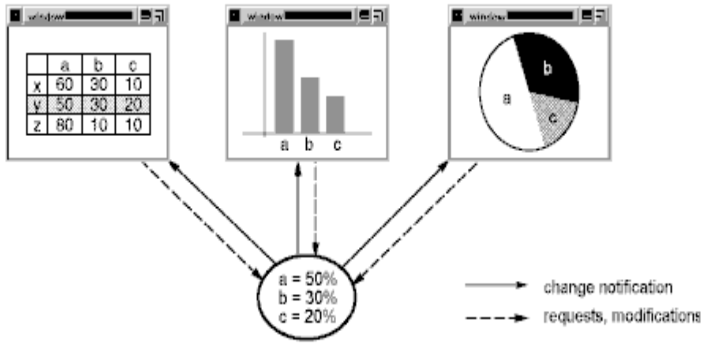
- Design patterns represent solutions to problems that arise when developing software within a particular context
  - Patterns = **Problem/Solution** pair in **Context**
- Capture static and dynamic structure and collaboration among key participants in software designs
  - key participant – major abstraction that occur in a design problem
  - useful for articulating the ***how*** and ***why*** to solve *non-functional forces*.
- Facilitate reuse of successful software architectures and design
  - i.e. the “*design of masters*”… ;)

Why Use Patterns ?

Patterns help you learn from other’s successes,
instead of your own failures
Mark Johnson (cited by B. Eckel)

- An additional layer of abstraction
  - *separate things that change from things that stay the same*
  - distilling out common factors between a family of similar problems
  - similar to design
- Insightful and clever way to solve a particular *class of problems*
  - most general and flexible solution

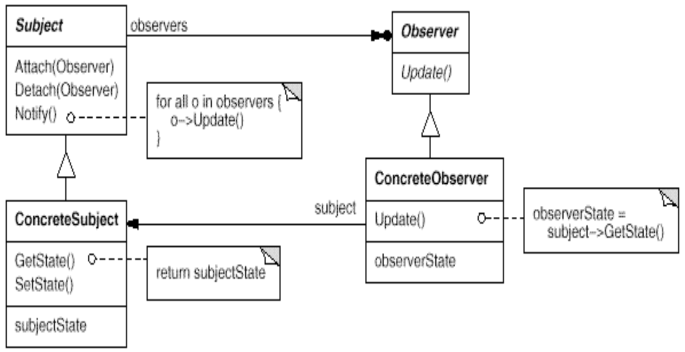
Example: Data-Views Consistency Problem



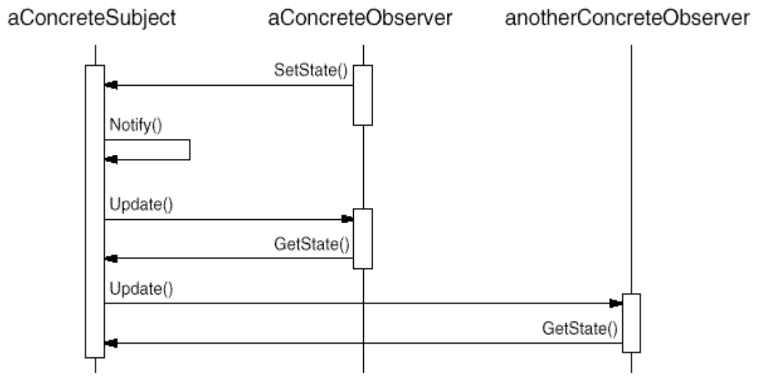
The Observer Pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically
- Forces
  - There may be many observers
  - Each observer may react differently to the same notification
  - The data-source (subject) should be as decoupled as possible from the observer
    - to allow observers to change independently of the subject

Structure of the Observer Pattern



Collaboration in the Observer Pattern



What Makes it a Pattern ?

A pattern must...

- ...solve a problem
  - i.e. it must be useful
- ...have a context
  - it must describe where the solution can be used
- ...recur
  - must be relevant in other situations
- ... teach
  - provide sufficient understanding to tailor the solution
- ... have a name
  - referred consistently

GoF Form of a Design Pattern

Pattern name and classification

Intent
   what does pattern do

Also known as
   other known names of pattern (if any)

Motivation
   the design problem

Applicability
   situations where pattern can be applied

Structure
   a graphical representation of classes in the pattern

Participants
   the classes/objects participating and their responsibilities

Collaborations
   of the participants to carry out responsibilities

GoF Form of a Design Pattern (contd.)

Consequences
   trade-offs, concerns

Implementation
   hints, techniques

Sample code
   code fragment showing possible implementation

Known uses
   patterns found in real systems

Related patterns
   closely related patterns

Classification of Design Patterns

- Creational Patterns
  - deal with initializing and configuring classes and objects
  - *how am I going to create my objects?*
- Structural Patterns
  - deal with decoupling the interface and implementation of classes and objects
  - *how classes and objects are composed to build larger structures*
- Behavioral Patterns
  - deal with dynamic interactions among societies of classes and objects
  - *how to manage complex control flows (communications)*

Design Pattern Catalog - GoF

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Vistor

Benefits of Design Patterns

- Inspiration
  - *patterns don't provide solutions, they **inspire** solutions*
  - Patterns **explicitly** capture expert knowledge and design tradeoffs and make this expertise widely available
  - ease the transition to object-oriented technology
- Patterns improve developer communication
  - pattern names form a **vocabulary**
- Help document the architecture of a system
  - enhance understanding
- Design patterns enable large-scale reuse of software architectures

Drawbacks of Design Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from patterns overload
- Integrating patterns into a software development process is a human-intensive activity

Key Mechanisms in Design Patterns

Class vs. Interface Inheritance

- Class – defines an implementation
- Type – defines only the interface
  - the set of requests that an object can respond to
- Relation between Class and Type
  - the class implies the type

On class, many types. Many classes, same type

- Class Inheritance = one implementation in terms of another
- Type Inheritance = when an object can be used in place of another

GoF Design Principle no. 1

Program to an interface, not an implementation

- Use interfaces to define common interfaces
  - and/or abstract classes in C++
- Declare variables to be instances of the abstract class
  - not instances of particular classes
- Use *Creational patterns*
  - to associate interfaces with implementations

Benefits

- Greatly reduces the implementation dependencies
- Client objects remain unaware of the classes that implement the objects they use.
- Clients know only about the abstract classes (or interfaces) that define the interface.

Class vs. Object Patterns

- Mechanisms of reuse
  - White-box vs. Black-box
- Class Inheritance
  - easy to use; easy to modify
    - implementation being reused;
  - language-supported
  - static bound ⇒ can't change at run-time;
  - mixture of physical data representation ⇒ breaks encapsulation
    - change in parent ⇒ change in subclass
- Object Composition
  - objects are accessed solely through their interfaces
    - no break of encapsulation
  - any object can be replaced by another at runtime
    - as long as they are the same type

Design Principle no. 2

Favor composition over class inheritance

- Keeps classes focused on one task
- Inheritance and Composition Work Together!
  - ideally no need to create new components to achieve reuse
  - this is rarely the case!
  - reuse by inheritance makes it easier to make new components
    - modifying old components
- Tendency to overuse inheritance as code-reuse technique
- Designs – more reusable by depending more on object composition