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, lighthing, gardens, ...
  - a generative pattern language

Quality Without a Name...

… there is a central quality, which is the root criterion of life and spirit in a man, a town, a building, or a wilderness.
This quality is objective and precise, but it cannot be named.

… the search, which we make for this quality, in our own lives, is the central search of any person, and the crux of any individual person's story.
It is the search of ***those moments when we are most alive****.*

C.Alexander – Timeless Way of Building

What is a Pattern ?

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*

C. Alexander, "*The Timeless Way of Building*", 1979

Alexander's View of a Pattern

- Three part rule that expresses a relation between a certain context, a problem and a solution.
- ***Element of the world*** – a relationship between
  - a context
  - a system of forces that occur repeatedly in the context
  - a spatial configuration which allow forces to resolve themselves
- ***Element of language*** – an instruction
  - describes how the spatial configuration can be repeatedly used
  - to resolve the given system of forces
  - wherever the context makes it relevant
- The "*thing – process*" dualism
  - a thing that happens in the world
  - a process (rule) which will generate that thing

Alive and Dead Patterns

… the specific patterns out of which a building or a town is made
*may be **alive** or **dead**.*
*To the extent they are alive, they **let our inner forces loose**, and set us free;*
*but when they are dead, they keep us **locked in inner conflict**.*

… the more living patterns there are in a place
– a room, a building, or a town –
*the more it **comes to life as an entirety**,*
the more it glows,
the more it has that self-maintaining fire
which is the quality without a name.

C. Alexander, "*The Timeless Way of Building*", 1979

Pattern Language

… once we have understood how to discover individual patterns,
which are alive,
we may then make a language for ourselves for any building task we face.
The structure of the language is created by
*the **network of connections among individual patterns**:*
and the language lives, or not, as a totality,
to the degree these patterns form a whole.

C. Alexander, "*The Timeless Way of Building*", 1979

The Timeless Way: Leaving the Gate Behind…

… indeed, this ageless character has nothing, in the end, to do with languages.
The language and the process, which stem from it,
*merely **release the fundamental order**, which is native to us.*

They do not teach us,
they only remind us of what we know already,
and of what we shall discover time and time again,
when we give up our ideas and opinions,
and do exactly what emerges from ourselves.

Why Use Patterns ?

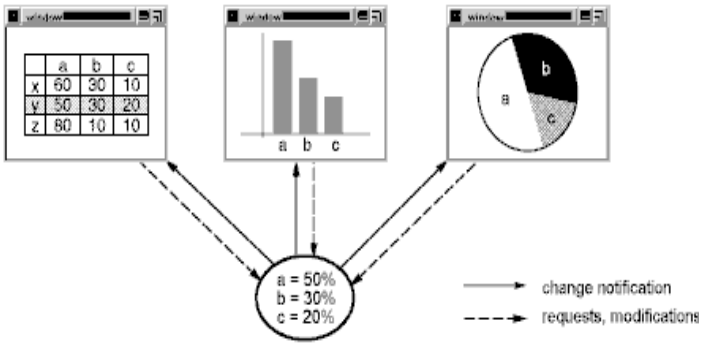
Patterns help you learn from other's successes, instead of your own failures
Mark Johnson (cited by B. Eckel)

- 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

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*"… ;)

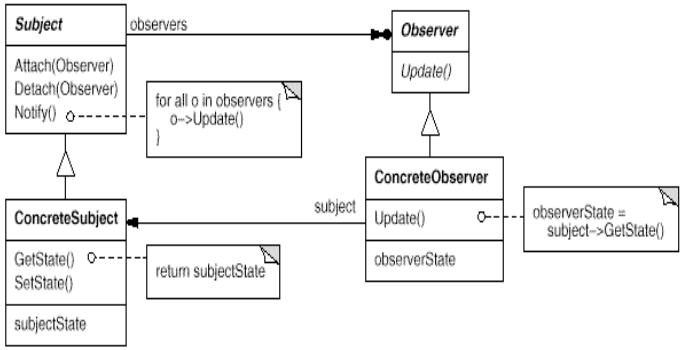
Example: Data-Views Consistency Problem



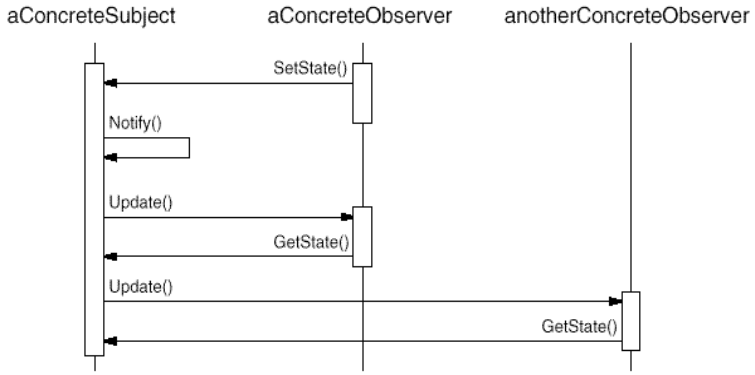
The Observer Pattern

- 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 Inheritance vs. Composition

- 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

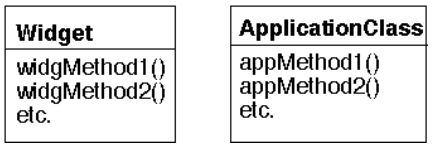
Favor composition over class inheritance

- 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

Creational Patterns

Stirring you up...

Let's start simple...

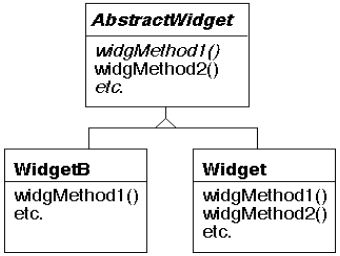


- We can modify the internal `Widget` code without modifying the `ApplicationClass`

Problems with Changes

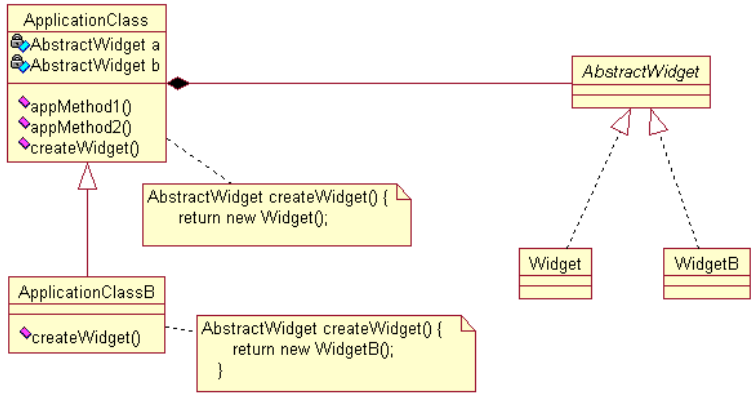
- What happens when we discover a new widget and would like to use in the **`ApplicationClass`**?
- Multiple coupling between **`Widget`** and **`ApplicationClass`**
  - **`ApplicationClass`** knows the interface of **`Widget`**
  - **`ApplicationClass`** explicitly uses the **`Widget`** type
    - hard to change because Widget is a concrete class
  - **`ApplicationClass`** explicitly creates new Widgets in many places
    - if we want to use the new **`Widget`** instead of the initial one, changes are spread all over the code

Apply "Program to an Interface"



- **`ApplicationClass`** depends now on an (abstract) interface
- But we still have hard coded which widget to create!
  - should I copy-paste? ;-)

Use a Factory Method

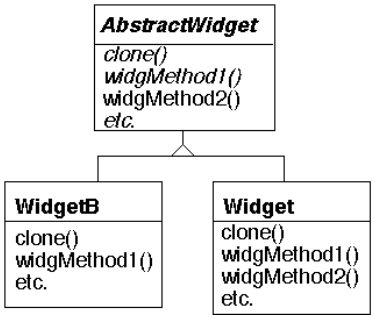


Evaluation of Factory Method Solution

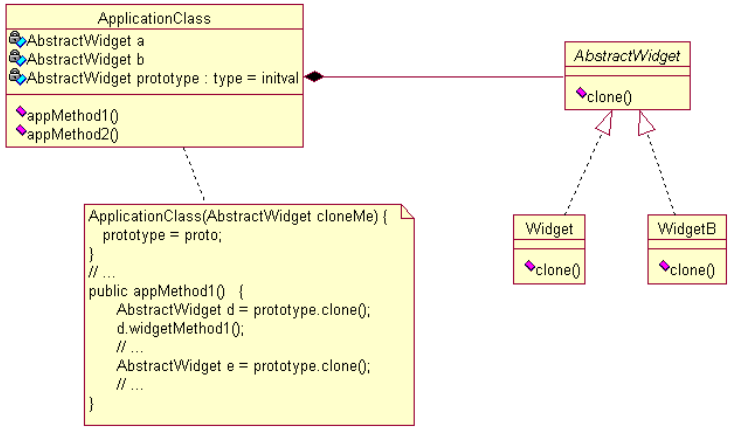
- Explicit creation of **`Widget`** objects is not anymore dispersed
  - easier to change
- Functional methods in **`ApplicationClass`** are decoupled from various concrete implementations of widgets
- Avoid ugly code duplication in **`ApplicationClassB`**
  - subclasses reuse the functional methods, just implementing the concrete Factory Method needed
- Disadvantages
  - create a subclass only to override the factory-method
  - can't change the **`Widget`** at run-time

Solution 2: Clone a Prototype

- Provide the **`Widgets`** with a clone method
  - make a copy of an existing Widget object



Using the Clone



Advantages

- Classes to instantiate may be specified dynamically
  - client can install and remove prototypes at run-time
- We avoided subclassing of **`ApplicationClass`**
  - Remember: Favor Composition over Inheritance!
- Totally hides concrete product classes from clients
  - Reduces implementation dependencies

More Changes

- What if ApplicationClass uses other "products" too...
  - e.g. Wheels, Cogs, etc.
- Each one of these stays for an object family
  - i.e. all of these have subclasses
- Assume that there are restrictions on what type of Widget can be used with which type of Wheel or Cog
- Factory Methods or Prototypes can handle each type of product but it get hard to insure the wrong types of items are not used together

Solution: Create an Abstract Factory