Design Patterns

Overview

- The What and Why of patterns
- Pattern Categories
- Example of Design Patterns

Introduction

- Designing software for reuse is difficult as one must look for:
  - good problem decomposition, and the right software abstractions
  - flexibility, modularity and elegance
- Designs often emerge from an iterative process (trials and many errors)
- The good news is that successful designs do exist
  - in fact they exhibit some recurring characteristics
  - but they are almost never identical
- The engineering perspective: can designs be described, codified or standardized?
  - this would short circuit the trial and error phase
  - produce "better" software faster

Patterns in Perspective

"Talking about music is like dancing about architecture."
Steve Martin

- Architecture has some of the same problems as Software
  - each building is different with a different set of constraint
  - quality of a design is intangible, yet clearly exists. Some dimensions are
    - *does the building stand?*
    - *is it pleasant to live in?*
    - *other metrics such as comfort, harmony, freedom...*
- The concept of a "pattern" was first expressed in Christopher Alexander's work (A Pattern Language) in 1977 (2543 patterns)
- A *pattern* is a solution to a problem in a context
  - a pattern balances requirements, e.g. "designing a room" must take into account an entrance transition, an intimacy gradient, and ensure that there is light on two sides of every room.
- When multiple patterns are *connected*, we talk about a *Pattern Language*

Patterns in Perspective

For Software, patterns date from last century:

- 1987 Cunningham & Beck develop a pattern language for Smalltalk
- 1990 The Gang of Four (GoF) compile a catalog of design patterns
- 1991 Anderson gives first Patterns Workshop at OOPSLA
- 1993 Beck & Booch sponsor the first meeting of the Hillside Group
- 1994 First Pattern Languages of Programs (PLoP) conference
- 1995 "Design Patterns: Elements of Reusable Software" published

Software Design Patterns are...

- A pattern describes a recurring software structure and
  - abstract from concrete design elements such as problem domain, programming language (e.g. pattern book talks about Smalltalk and C++, but most things apply to Java)
  - identifies classes that play a role in the solution to a problem, describes their collaborations and responsibilities
  - lists implementation trade-offs
- Patterns are neither code, nor designs as they must be *instantiated* and applied. This process requires the engineer to:
  - evaluate trade-offs and impact of using a pattern in the system at hand
  - make design and implementation decision how best to apply the pattern, perhaps modify it slightly
  - implement the pattern in code and combine it with other patterns

A typical exam question: What sets apart a pattern language from a programming language?

Software Robustness to Change

- Change is intrinsic to software development, as requirements, use-cases, technologies and platforms evolve
- Robustness to change means that software can be modified locally without endangering overall structure. It is a quality that reflects ease of evolution and maintenance costs.
- Patterns address particular aspects of a system and allow for variation/evolution in those aspects.
  - for example: factories allow to change implementations ...

Benefits of using Patterns

- Patterns are a common design vocabulary
  - they allow engineers to abstract a problem and talk about that abstraction in isolation from its implementation.
  - they embody a culture, domain specific patterns increase design velocity
- Patterns capture design expertise and allow that expertise to be communicated
- Promote design reuse and avoid mistakes
- Improve documentation (less is need) and understandability (patterns are described well once)

GoF: *23 patterns in 3 categories*

- Creational Patterns
  abstract the object instantiation process
  - Factory Method   Abstract Factory   Builder
  - Prototype   Singleton
- Structural Patterns
  describe how classes and objects can be combined to form larger structures
  - Adapter   Bridge   Composite
  - Decorator   Facade   Flyweight
  - Proxy
- Behavioral Patterns
  concerned with communication between objects
  - Chain of Responsibility   Command   Interpreter
  - Iterator   Mediator   Observer
  - Template Method   State   Strategy
  - Visitor

A Template for Design Patterns

- Pattern Name and Classification
  - Concise name for the pattern and it's type (very important)
- Intent
  - Short statement about what the pattern does
- Also Known As
  - Other names for the pattern
- Motivation
  - A scenario that illustrates where the pattern would be useful
- Applicability Situations
  - Where the pattern can be used
- Structure
  - A graphical representation of the pattern
- Participants
  - The classes and objects participating in the pattern

A Template for Design Patterns

- Collaborations
  - How to do the participants interact to carry out their responsibilities?
- Consequences
  - What are the pros and cons of using the pattern?
- Implementation
  - Hints and techniques for implementing the pattern
- Sample Code
  - Code fragments for a sample implementation
- Known Uses
  - Examples of the pattern in real systems
- Related Patterns
  - Other patterns that are closely related to the pattern

Patterns by Example

- See the GoF Book.
- Consider *Creational Patterns*
  - their goal is to abstract away the process of creating an object from the client code; and thus enable implementations to be configure
  - the operations that instantiate new objects are hidden from the client
- As an example consider a Maze:

```
Maze create() {
   Maze maze = new Maze();
   Room r1   = new Room(1);
   Room r2   = new Room(2);
   Door door = new Door(r1, r2);
   maze.addRoom(r1); maze.addRoom(r2);
   r1.setSide(North, new Wall()); r1.setSide(East, door);
   r1.setSide(South, new Wall()); r1.setSide(West, new Wall());
```

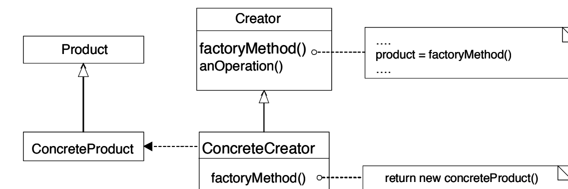
Patterns by Example (2)

- This design is not robust to change. Creational patterns can add some flexibility to the code. Choose one of:
  - Factory Method: uses methods instead of constructors
  - Abstract Factory: parameterize method with a creator object
  - Builder: parameterize method with an objec that construct a complete maze
  - Prototype: parameterize method with the prototypical objects for all components of a maze

```
Maze create() {
       Maze maze = new Maze();
       Room r1   = new Room(1);
       Room r2   = new Room(2);
       Door door = new Door(r1, r2);
       maze.addRoom(r1); maze.addRoom(r2);
       r1.setSide(North, new Wall()); r1.setSide(East, door);
       r1.setSide(South, new Wall()); r1.setSide(West, new Wall());
       r2.setSide(North, new Wall()); r2.setSide(East, new Wall());
       r2.setSide(South, new Wall()); r2.setSide(West, door);
       return maze;
}
```

Factory Method Pattern

- Intent
  - Provide an interface for creating an object, but choice of object's concrete class is delegated to subclasses.
- Motivation
  - frameworks often must instantiate classes but for each use of the framework different concrete classes may need to be created
- Applicability
  - Use the FM pattern if a class can't anticipate the objects it must create or a class wants its subclasses to specify the objects it creates.
- Structure



Factory Method Pattern

- Participants
  - Product
    Defines the interface for the type of objects the factory method creates
  - ConcreteProduct
    Implements the Product interface
  - Creator
    Declares the factory method, which returns an object of type Product
  - ConcreteCreator
    Overrides the factory method to return an instance of a ConcreteProduct
- Collaborations
  - Creator relies on its subclasses to implement the factory method so that it returns an instance of the appropriate ConcreteProduct

Factory Method Pattern

- Consequences
  - By avoiding to specify the class name of the concrete class and the details of its creation the client code has become more flexible
  - The client is only dependent on the interface
  - Construction of objects requires one additional class in some cases
- Implementation
  - There are two choices here
    - The creator class is abstract and does not implement creation methods (then it *must* be subclassed)
    - The creator class is concrete and provides a default implementation (then it *can* be subclassed)
    - Should a factory method be able to create different variants? If so the method must be equipped with a parameter.

Factory Method Pattern

- Implementation
  - An example of a parameterized factory method is

```
class Creator {
    public Product create(ProductID id) {
      if (id == MINE) return new MyProduct();
      if (id == YOURS) return new YourProduct();
      return null;
```

  - this can be extended to add more cases

```
class MyCreator extends Creator {
    public Product create(ProductID id) {
      if (id == YOURS) return new MyProduct();
      if (id == THEIRS) return new OurProduct();
      return super.create(id);
```

Factory Method Pattern

- Implementation
  - With generics the types used to describe products can be made more accurate

```
abstract class Creator<T extends Product> {
      public T create();
}

class MyCreator<T extends Product> {
     public T create() { return new T();}
}

MyCreator<YourProduct> factory = new MyCreator<YourProduct>();
```

Factory Method Pattern

- Revisiting the Maze example

```
class Maze {

   public Maze create();

   public Maze makeMaze() { return new Maze(); }

   public Maze makeRoom(int n) { return new Room(n); }

   public Maze makeWall() { return new Wall(); }

   public Maze makeDoor(Room r, Room r2) {return new Door(r, r2);}
}
```

Factory Method Pattern

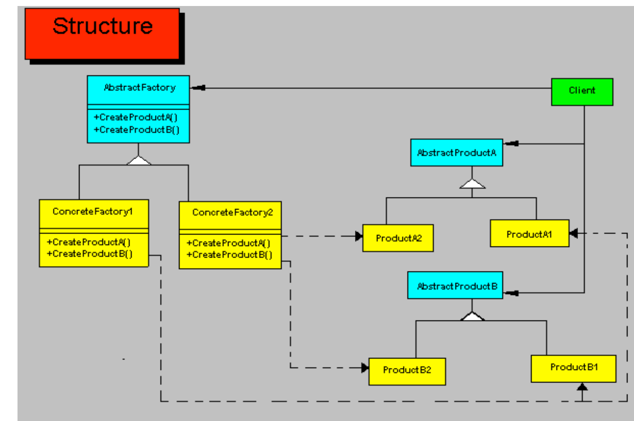
- Revisiting the Maze example

```
Maze create() {
    Maze maze = makeMaze();
    Room r1   = makeRoom(1);
    Room r2   = makeRoom(2);
    Door door = makeDoor(r1, r2);
    maze.addRoom(r1);   maze.addRoom(r2);
    r1.setSide(North, makeWall();   r1.setSide(East, door);
    r1.setSide(South, makeWall());   r1.setSide(West, makeWall());
    r2.setSide(North, makeWall();    r2.setSide(East, makeWall());
    r2.setSide(South, makeWall());   r2.setSide(West, door);
    return maze; }

    class MazeBombsGame extends Maze {
            public Maze makeRoom(int n) { return new RoomWithABomb(n); }
            public Maze makeWall() { return new BombedWall();  }

    }
```

Abstract Factory Pattern

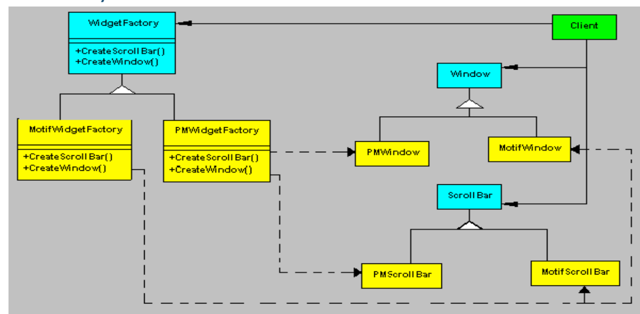
- INTENT
  - interface for creating families of related objects without revealing their concrete classes
- MOTIVATION
  - user interface toolkit that supports multiple look-and-feel standards (e.g. Motif and Presentation Manager)
  - applications should be portable across window managers
- APPLICABILITY
  - a system should be independent of how its products are created, composed and presented
  - a system should be configured with one of multiple families of products
  - a family of products is designed to be used together, and you need to enforce this constraint

Abstract Factory Pattern



Abstract Factory Pattern

- CONSEQUENCES
  - isolates concrete classes. clients only manipulate abstract interfaces
  - it makes changing product families easy.
  - it promotes consistency among products. forbids arbitrary mix and match across families
  - difficult to add new products, requires extending the interface of all factory classes



Abstract Factory Pattern

- IMPLEMENTATION
  - factories as singletons as only one instance of the class is needed (usually)
  - creating the products:
    - one factory method per kind of product (override the factory method to specify the actual objects to create)
    - when there can be a large number of families (i.e. requiring many concrete factory classes) use the prototype pattern to have a single concrete factory
- Sample Code

```
class MazeFactory {
  public Maze makeMaze() { return new Maze(); }

  public Maze makeRoom(int n) { return new Room(n); }
  public Maze makeWall() { return new Wall(); }
  public Maze makeDoor(Room r1, Room r2) { return new Door(r1, r2);}
}
```

Abstract Factory Pattern

```
Maze create(MazeFactory factory) {

     Maze maze = factory.makeMaze();
     Room r1   = factory.makeRoom(1);
     Room r2   = factory.makeRoom(2);
     Door door = factory.makeDoor(r1, r2);
     maze.addRoom(r1); maze.addRoom(r2);
     r1.setSide(North, factory.makeWall()); r1.setSide(East, door);
     r1.setSide(South, factory.makeWall());  r1.setSide(West, factory.makeWall());
     r2.setSide(North, factory.makeWall()); r2.setSide(East, factory.makeWall());
     r2.setSide(South, factory.makeWall());  r2.setSide(West, door);
     return maze;
}
```

Abstract Factory Pattern

```
class EnchantedMazeFactory extends MazeFactory {
     public EnchantedMazeFactory();
     Room makeRoom(int n) { return new EnchantedRoom(n); }
     Door makeDoor(Room r1, Room r2){ return new MagicDoor(r1,r2);}


class BombedMazeFacotry extends MazeFactory {
     public BombedMazeFactory();
     Room makeRoom(int n) { return new RoomWithABomb(n); }
     Door makeWall(){ return new BombedWall(r1,r2);}
```

can a RoomWithABomb use the special features of its walls?

References

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995
- *Pattern Oriented Software Architecture: A System of Patterns*, Frank Buschmann(Editor), Wiley, 1996
- Doug Schmidt for Distributed patterns.
- Links:
  - Design Patterns Home Page: http://hillside.net/patterns/patterns.html
  - Portland Pattern Repository: http://c2.com/ppr