Today: **Design Patterns I**

The *patterns* ruckus

- We (software people) didn’t start the fire
- Christopher Alexander
  - *A Pattern Language*
  - *A Timeless Way of Building*

The *patterns* ruckus (more)

The *patterns* ruckus (even more)
Patterns in the world

- There are some problems that come up over and over again in a domain
  - Architecture: How to provide easy access to an enclosed space selectively?
  - Software: How to allow for open extension of classes without modifying their currently accepted code?

- Each time a similar problem comes up, designers will typically start with something that has worked before
  - But then add a wrinkle inspired by something that works better for their current context

- In the software field, an enthusiastic community has formed around the concept of design patterns
  - The “Gang of Four” (I hate that name) Design Patterns book really started it all in 1995
  - Architectures and designs are discussed using patterns terminology
  - People won’t treat you with respect as a designer anymore if you’re not familiar with at least some patterns

A pattern must...

- Solve a problem
  - That is, it must be useful

- Have a context
  - It must describe where the solution can be used

- Recur
  - It must be relevant in other situations

- Teach
  - It must provide sufficient understanding to tailor the solution

- Have a name
  - It must be referred to consistently

Patterns craze

- Beware!
  - It’s a sign of lack of expertise to think everything can be solved with patterns
  - Or to believe that the more patterns one can stick into one’s design, the better

- Not everything is a pattern

- Patterns do not lead to direct code reuse

- Patterns are deceptively simple

- Even experts can disagree
  - (ESPECIALLY experts can disagree)

- Designers’ key quality is good judgment
  - Good judgment comes from experience
  - Patterns are not a workaround to exercising good judgment

Design Pattern Descriptions

- Name and intent

- Problem and context
  - What is the problem and the context where we would use this pattern?
  - Under what specific conditions should this pattern be used?

- Solution
  - A description of the elements that make up the design pattern
  - Emphasizes their relationships, responsibilities, and collaborations
  - Not a concrete design or implementation; rather an abstract description

- Positive and negative consequences of use
  - The pros and cons of using the pattern
  - Includes impact on reusability, portability, and extensibility
Organizing Patterns

- Organizing by purpose: What a pattern does
  - Creational: Creating, initializing, and configuring classes and objects
  - Structural: Composition of classes and objects
  - Behavioral: Dynamic interactions among classes and objects

- Organizing by scope: What the pattern applies to
  - Class patterns
    - Focus on the relationships between classes and their subclasses
    - Involve inheritance reuse
  - Object patterns
    - Focus on relationships between objects
    - Involve composition reuse

- How would you classify the Template Method pattern?

Do I need to master them all?

- No – at least not now

- For the final, you’re expected to:
  - Superficially know of all of them
  - Master in depth a few of each kind

- The patterns catalog doesn’t end with the Gang of Four book
  - There are concurrency patterns, analysis patterns, etc.
  - Unreasonable to know of all of them if they’re out of your domain

Maze example

- Building a maze for a computer game
  - A maze is a set of rooms
  - A room knows its neighbours
    - Another room
    - A wall
    - A door
Maze classes

That wasn’t fun

- Fairly complex member just to create a maze with two rooms
- Obvious simplification
  - Room could initialize sides with four new Walls
  - That (kind of) just moves the code elsewhere
- Problem lies elsewhere: inflexibility
  - Hard-codes the maze creation
  - Changing the layout can only be done by rewriting, or overriding and rewriting
- Plan for evolution
  - Want to make the maze more flexible
    - Easy to change the components of the game
  - What to do about DoorNeedingSpell or EnchantedRoom?
    - How can you change createMaze() so that it creates mazes with these different kinds of classes?
    - Biggest obstacle is hard-coding of class names

Design pattern: Factory method

- Use when:
  - A class can’t anticipate the kind of objects to create
  - Hide the secret of which helper subclass is the current delegate

Maze creation

```java
public Maze createMaze() {
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door d = new Door(r1, r2);
    r1.setSide(Direction.North, new Wall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, new Wall());
    r1.setSide(Direction.South, new Wall());
    r2.setSide(Direction.North, new Wall());
    r2.setSide(Direction.East, d);
    r2.setSide(Direction.West, new Wall());
    r2.setSide(Direction.South, new Wall());
    Maze m = new Maze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```
Factory method (structure)

- **Product**
  - Defines the interface of objects the factory method creates
- **ConcreteProduct**
  - Implements the Product interface
- **Creator**
  - Declares the factory method which returns a Product type
  - Defines a default implementation
  - Calls the factory method itself
- **ConcreteCreator**
  - Overrides factory method: returns instance of ConcreteProduct

Sample code

```java
public class MazeGame {
    public static void main(String[] args) {
        Maze m = new MazeGame().createMaze();
    }
    private Maze makeMaze() { return new Maze(); }
    private Wall makeWall() { return new Wall(); }
    private Room makeRoom(int r) { return new Room(r); }
    private Door makeDoor(Room r1, Room r2) { return new Door(r1, r2); }
    public Maze createMaze() {
        Room r1 = makeRoom(1);
        Room r2 = makeRoom(2);
        Door d = makeDoor(r1, r2);
        r1.setSide(Direction.North, makeWall());
        r1.setSide(Direction.East, d);
        r1.setSide(Direction.West, makeWall());
        r1.setSide(Direction.South, makeWall());
        r2.setSide(Direction.North, makeWall());
        r2.setSide(Direction.East, d);
        r2.setSide(Direction.West, makeWall());
        r2.setSide(Direction.South, makeWall());
        Maze m = makeMaze();
        m.addRoom(r1);
        m.addRoom(r2);
        return m;
    }
}
```

Sample code (cont)

```java
public class BombedMazeGame extends MazeGame {
    private Wall makeWall() { return new BombedWall(); }
    private Room makeRoom(int r) { return new RoomWithABomb(r); }
    public Maze createMaze() {
        Room r1 = makeRoom(1);
        Room r2 = makeRoom(2);
        Door d = makeDoor(r1, r2);
        r1.setSide(Direction.North, makeWall());
        r1.setSide(Direction.East, d);
        r1.setSide(Direction.West, makeWall());
        r1.setSide(Direction.South, makeWall());
        r2.setSide(Direction.North, makeWall());
        r2.setSide(Direction.East, d);
        r2.setSide(Direction.West, makeWall());
        r2.setSide(Direction.South, makeWall());
        Maze m = makeMaze();
        m.addRoom(r1);
        m.addRoom(r2);
        return m;
    }
}
```

Sample code (cont 2)

```java
public class EnchantedMazeGame extends MazeGame {
    private Room makeRoom(int r) { return new EnchantedRoom(r, castSpell()); }
    private Door makeDoor(Room r1, Room r2) { return new DoorNeedingSpell(r1, r2); }
    private Spell castSpell() { return new Spell(); }
}
```
Sample code (cont 3)

```java
public static void main(String[] args) {
    Maze m = new EnchantedMazeGame().createMaze();
}
```

```
public static void main(String[] args) {
    Maze m = new BombedMazeGame().createMaze();
}
```

Consequences

- **Advantage**
  - Eliminates the need to bind specific implementation classes
  - Can work with any user-defined ConcreteProduct classes

- **Disadvantage**
  - Uses an inheritance dimension
  - Must subclass to define new ConcreteProduct objects
  - Interface consistency required

Other consequences

- Provides hooks for subclasses
  - Always more flexible than direct object creation

- Connects parallel class hierarchies
  - Localizes knowledge of which classes belong together

Implementation

- Two major varieties
  - Creator class is abstract
    - Requires subclass to implement
  - Creator class is concrete; provides a default implementation
    - Optionally allows subclass to re-implement

- Parameterized factory methods
  - Generic make() method takes class id as parameter

- Naming conventions
  - Use makeXXX() names

- Return object of class to be created
  - Or store as member variable