7a. Design Patterns – Creational Patterns

What is a design pattern?

- You will often find that new programming problems look a lot like old problems – and you have solved them before
- Good solutions are studied as “design patterns”
  - They are inherently reusable, flexible, and elegant
- Examples:
  - How do you write an object that can apparently change classes during run-time?
  - What should you do if you want the user to be able to restore a save state?
  - How do you reduce repetitive memory usage due to object creation?
- We will study around a dozen design patterns across three categories
- Popularized by “Gang of Four” textbook (Gamma, Helm, Johnson, Vlissides)

Starting principles (GoF)

- Interfaces are “types” – it only describes what requests it supports
- *Program to an interface, not an implementation*
  - Interfaces are “contracts” – fixed, small, and well-defined
  - Implementations can vary and can change
  - Minimize *instantiation* of concrete classes; commit to abstract classes
- Advantage: reduce dependencies, encourage polymorphism

Starting principles (GoF)

- Example code:

```
public void initAnimals() {
    rats = new ArrayList<Rat>();
    for (int i = 0; i < initRatNum; i++) {
        Rat rat = new Rat(RandomPoint());
        rats.add(rat);
    }
}
```

- Currently, Rat is a concrete class
  - If I want to add several other types of rat…
  - If I want to add other types of moving enemies…
- Currently, the Cat collision code only checks the list of rats

Starting principles (GoF)

- *Favor object composition over class inheritance*
- Inheritance is useful when we want to reuse functionality, however:
  - “Our experience is that designers overuse inheritance as a reuse technique”
  - Composition also allows you to reuse functionality
- Inheritance necessarily breaks encapsulation
- Composition can be thought of as “black-box reuse”

Starting principles (GoF)

- Inheritance: a BossRat is a Rat that has extra functions to fight back
- Composition: each Enemy has a DamagingType, DefendingType, and MovingType field
  - Use setters for each Enemy after creation
  - A Rat has no damage, no defense, and moves
  - A BossRat has damage, defense, and moves
  - A Door has no damage, defense, and does not move
  - A Trap has damage, no defense, and does not move
  - Now we can also change their behavior during gameplay

Three types of design patterns

- **Creational** patterns: How do we create objects?
- **Structural** patterns:
- **Behavioral** patterns:

Creational patterns

- As code gets larger, we move towards composition and away from inheritance to maintain coherence
  - Large inherited classes are unwieldy
- It becomes important to know when, where, and how we are instantiating objects
  - Composing a specific type of object becomes complicated
- Creational patterns help us solve these problems

Creational patterns

- Example: Create a Stage with a Cat and some Enemies
- We want to flexibly create many different types of stages during the game
- Do we want to:
  - startStage(numEnemies, typeEnemies, etc.) which calls Stage(numEnemies, typeEnemies, etc.)?
  - startStage() calling virtual functions e.g. Stage.createRats() to construct objects?
  - startStage(StageFactory) where we use the StageFactory to construct the stage?
  - Other patterns?

Factory Method

- Suppose now we have many different types of Enemies in our game
  - NormalRat, BossRat, Door, Trap, …
  - These inherit from Enemy with different rules and methods
- We want to unify the Enemy creation process with a single method
  - Stage setup will call this method to create enemies, depending on the difficulty and type of stage
- But we don’t know what to create (?)
  - Different types of Stages will have different Enemy properties and distribution

Factory Method: Example

- Solution: put a CreateEnemies() method (“Factory Method”) in the base Stage class; let inheritors define what to create

```
public class HardStage extends Stage {
   private EnemyList CreateEnemies() {
      EnemyList stageEnemies = new EnemyList();
      for (int i = 0; i < 5; i++) {
         BossRat bossRat = new BossRat();
         bossRat.HP *= 2;
         bossRat.ATK *= 2;
         stageEnemies.addEnemy(BossRat);
      }
   }
   //define other hard stage methods, such as changing collision
rules or giving enemies regenerating health points…
}
```

Factory Method: Example

- Solution: put a CreateEnemies() method (“Factory Method”) in the base Stage class; let inheritors define what to create

```
public class BonusStage extends Stage {
   private EnemyList CreateEnemies() {
      EnemyList stageEnemies = new EnemyList();
      for (int i = 0; i < 20; i++) {
         NormalRat rat = new NormalRat();
         rat.GoldReward *= 2;
         rat.Speed /= 2;
         stageEnemies.addEnemy(rat);
      }
   }
   //define other bonus stage methods, such as disabling escape…
}
```

Factory Method: Terminology

- CreateEnemies() is our *Factory Method*
- The base Stage class can have an implementation that's overridden, or it may have no implementation at all
- Stage is the *Creator* class
- HardStage, BonusStage are *ConcreteCreators*
- EnemyList is the *Product*

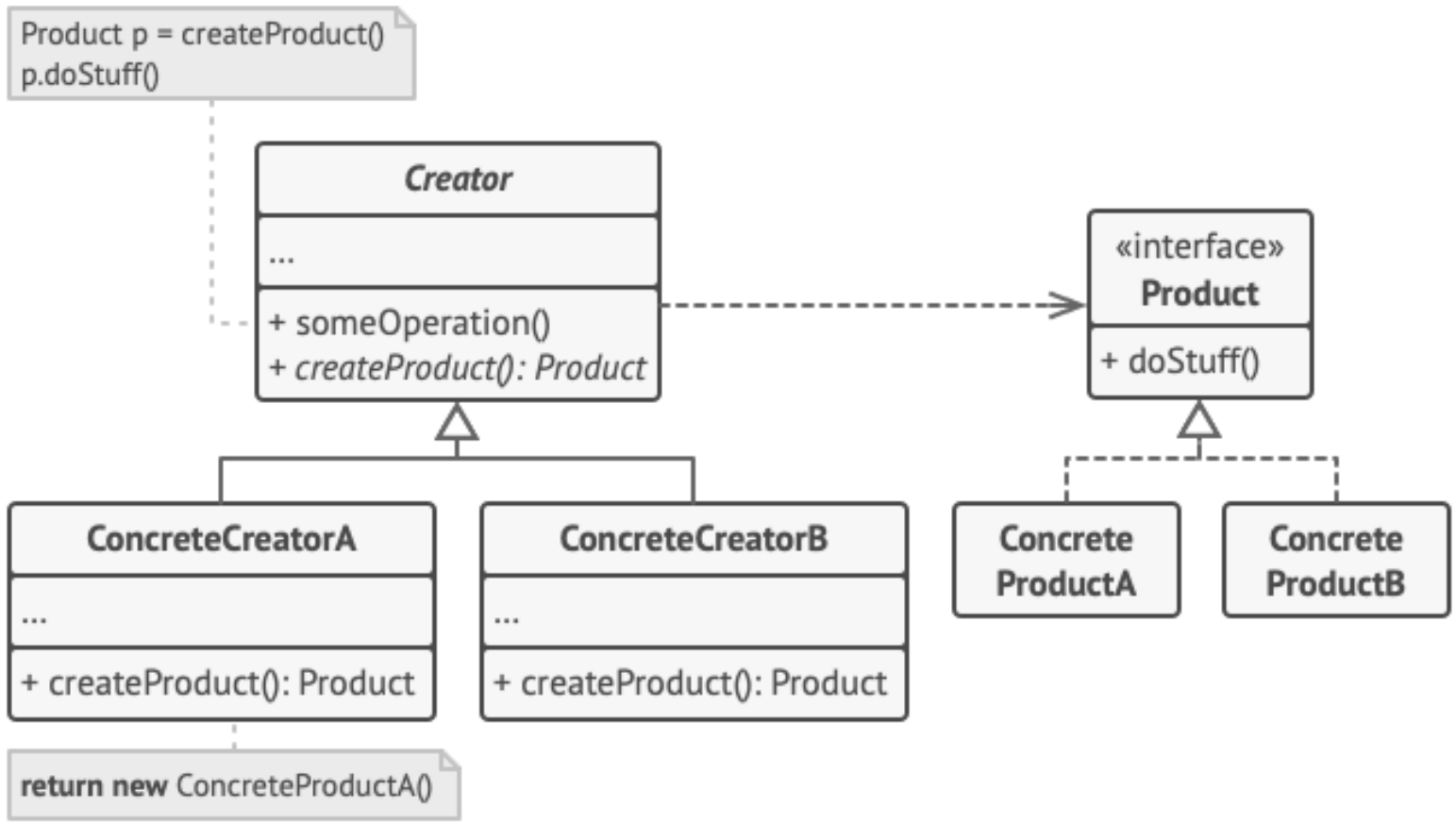
Factory Method: Advanced

- You can also have *ConcreteCreators* create *ConcreteProducts* that are implementations of Product
- Suppose now we have HardEnemyList and BonusEnemyList as separate implementations of the EnemyList interface
  - This may be helpful if, for example, EnemyList handles logic that decides how rats work together to escape or to attack you
- Parallelism is achieved: HardStage creates HardEnemyList, BonusStage creates BonusEnemyList, etc.

Factory Method: Why?

- What would we do if we didn't use a Factory Method?
- CreateHardStageEnemies(), CreateBonusStageEnemies()…
  - Bad: does not allow HardStage, BonusStage inheritance from Stage
- Stage takes care of CreateEnemies() for all cases…
  - Bad: Breaks dependency inversion
- When *not* to use a Factory Method?
  - If you don't need the subclasses, i.e. you would be creating subclasses just to inherit a Factory Method
  - In our example, if we want many variant normal stages with different rat distributions but no special rules?

Factory Method: Diagram



refactoring.guru

Factory Method: Example 2

refactoring.guru

- We want to develop a **cross-platform UI**
  - Specifically, a dialog box
- It should have different style buttons if viewed through web browser or run as a windows app
- WindowsDialog and WebDialog will be children of Dialog

Factory Method: Example 2 (Pseudocode)

```
class Dialog:
    abstract void createButton():Button

    void render():
        // Call the factory method to create a product object.
        Button okButton = createButton()
        // Now use the product.
        okButton.onClick(closeDialog)
        okButton.render()
```

Factory Method: Example 2 (Pseudocode)

```
class WindowsDialog extends Dialog:
    Button createButton():
        return new WindowsButton()

class WebDialog extends Dialog:
    Button createButton():
        return new HTMLButton()
```

```
class WindowsButton implements Button:
        void render(a, b):
         // Render a button in Windows style.
        void onClick(f):
         // Bind a native OS click event.

class HTMLButton implements Button:
        void render(a, b):
         // Return an HTML representation of a button.
        void onClick(f):
         // Bind a web browser click event.
```

Factory Method: Example 2 (Pseudocode)

(Main code that starts up dialog box)

```
 if (config.OS == "Windows"):
       Dialog dialog = new WindowsDialog()
else if (config.OS == "Web"):
       Dialog dialog = new WebDialog()
```

Abstract Factory

- An Abstract Factory allows us to create a **class** whose purpose is to produce a product
- When would we need a whole class instead of just a method?
  - When we need multiple methods to create a product
  - These methods change depending on the product
- For our example: suppose our now more complicated stage consists of not just enemies, but walls, exits, doors, and treasure
- We want several ways to set up these stages, depending on the stage's nature

Abstract Factory: Example

- Create a StageFactory (abstract base), then inherit from that:

```
public class HardStageFactory extends StageFactory {
   private EnemyList CreateEnemies() {
      HardEnemyList myEnemies = new HardEnemyList();
      //create a lot of enemies
      return HardEnemyList;
   }
   private WallList CreateWalls() {
      //create a few walls
   }
   private TrapList CreateTraps() {
      //create a lot of traps
   }
}
```

Abstract Factory: Example

- Create a StageFactory (abstract base), then inherit from that:

```
public class EasyStageFactory extends StageFactory {
   private EnemyList CreateEnemies() {
      EasyEnemyList myEnemies = new EasyEnemyList();
      //create a few easy enemies
      return EasyEnemyList;
   }
   private WallList CreateWalls() {
      //create a few walls
   }
   private TrapList CreateTraps() {
      //create no traps
   }
}
```

Abstract Factory: Example

- Create a StageFactory (abstract base), then inherit from that:

```
public class PuzzleStageFactory extends StageFactory {
   private EnemyList CreateEnemies() {
      //create no enemies
   }
   private WallList CreateWalls() {
      //create lots of walls
   }
   private TrapList CreateTraps() {
      //create lots of traps
   }
}
```

Abstract Factory: Example

- Now, our client will call the code as follows:

```
public class GamePanel extends Panel {
   StageFactory factory;
   Stage stage;
   void SetupStage() {
      //set factory to the correct type, then…
      stage.walls = factory.CreateWalls();
      stage.enemies = factory.CreateEnemies();
      stage.traps = factory.CreateTraps();
   }
   void respawnEnemies() {
      stage.enemies = factory.CreateEnemies();
   }
}
```

Abstract Factory: Why?

- No chance of accidentally creating nonsensical Stage
  - In other words, all Stages we create will be carefully designed
- Outward-facing client code is simple
- Could we do this with Factory Methods?
  - Yes: HardStage itself would have CreateEnemies(), CreateTraps(), etc. but also many other functionalities about the Stage it wants to implement
  - It may be preferable to separate out the object creation methods into an AbstractFactory for the Single-Responsibility Principle
- When *not* to use Abstract Factory?
  - It necessarily calls for the creation of more classes, which may increase complexity

Abstract Factory: Example 2

refactoring.guru

- Continuation of Factory Method: Example 2
- We also want an HTML Checkbox
- It would make no sense to create something with a Windows Button and an HTML Checkbox

Abstract Factory: Example 2 (Pseudocode)

```
interface GUIFactory:
    Button createButton()
    Checkbox createCheckbox()
```

```
class WinFactory implements GUIFactory:
    Button createButton():
        return new WinButton()
    Checkbox createCheckbox():
        return new WinCheckbox()
```

```
class HTMLFactory implements GUIFactory:
    Button createButton():
        return new HTMLButton()
    Checkbox createCheckbox():
        return new HTMLCheckbox()
```

Abstract Factory: Example 2 (Pseudocode)

```
class Application:
    private field factory: GUIFactory
    private field button: Button
    private field checkbox: Checkbox
    Application(GUIFactory Factory):
        this.factory = factory
    void createUI():
        this.button = factory.createButton()
        this.checkbox = factory.createCheckbox()
    void paint():
        button.paint()
        checkbox.paint()
```

- Now, we can create a Windows application by constructing

  Application(new WinFactory())

Abstract Factory: Terminology

- Abstract Factory: GUIFactory
- Concrete Factory: WinFactory, HTMLFactory
- Abstract Product: Button, Checkbox
- Concrete Product: WinButton, WinCheckbox, …

Prototype

- A way to copy an object
- Could we just do:

```
Rat Rat2 = new Rat()
Rat2 = Rat1;
```

- No, that's not a new object
- Could we do:

```
Rat Rat2 = new Rat()
Rat2.Health = Rat1.Health;
Rat2.ATK = Rat1.ATK;
```

Prototype

- If we need to clone the Rat object, we should give it a clone method

```
public class Rat {
   Rat clone() {
      Rat newRat = new Rat();
      newRat.HP = this.HP;
      newRat.MaxHP = this.maxHP;
      newRat.ATK = this.ATK;
      newRat.setLocation(this.location);
      return new Rat;
   }
}
```

Prototype: Advanced

- The better use case of Prototype is when you don't exactly know which subtype of object you will be creating
- I want the Cat to be able to press a button that creates new Enemies on the stage
- This Button is an object, but there may be *several types of buttons* – for creating easy enemies, hard enemies, a mix…
- How would other **design patterns** solve this problem?

Prototype: Advanced

- Factory Method: Button has a PressButton() factory method that creates enemies, subclasses will override it
  - EasyButton would create EasyEnemy, HardButton would create HardEnemy
- Abstract Factory: extract code to create several subclasses, one for each type of button
  - EasyButtonFactory, HardButtonFactory, MixButtonFactory…
- Both require more subclasses…
- But Prototype can achieve this without extra subclasses

Prototype: Advanced

```
public class Button {
   List<Prototype> prototypes;
   int curEnemyInd = 0;
   Enemy pressButton() {
      if (curEnemyInd >= prototypes.length()) return null;
      curEnemyInd += 1;
      return prototypes[i].clone();
   }
}
```

- This Button is highly flexible: you can put a list of whatever enemies you want
- Stage is responsible for creating the right enemies

Prototype: Advanced

```
public class HardStageFactory extends StageFactory {
   Button createButton() {
      Button button = new Button();
      button.prototypes.add(getEnemyPrototype(“Hard”));
      button.prototypes.add(getEnemyPrototype(“Easy”));
      button.prototypes.add(getEnemyPrototype(“Hard”));
   }
}
```

- getEnemyPrototype() can call a Prototype Manager that keeps a prototype of all enemies
  - For example it could scale enemies by stage

Prototype: Terminology

- Prototype: Prototype (used in Button)
- Prototype Manager: can keep multiple pre-built Prototypes for use in different parts of the code
- Client: pressButton() code that asks the Button to clone

Prototype: Why?

- Two needs:
  1. You need to copy an object
  2. Your code needs to ignore what specific implementation of the object you're copying; maybe this is decided during run-time
- You can solve this with more subclassing, but this would increase code complexity and decrease flexibility
- When *not* to use Prototype?
  - When you have to clone a highly complex object with circular references
  - Shallow copy vs deep copy?
  - Note that clone()'s signature needs to be fixed

Builder

- A Builder is a class for creating complex multi-step objects
- Didn’t we already have a pattern with a class for creating complex products?
  - Abstract Factory is not concerned with steps
- Different ConcreteBuilders inherit from the Builder class to produce different products
- A Director can guide any builder

Builder

- Our Abstract Factory example is awkward because it should've been solved with the Builder pattern instead
- Here is what we might do to create a properly challenging Stage…

1. Set up several rooms, create walls for these rooms.
2. Create appropriate traps in rooms.
3. Create enemies for each room. Enemies will not spawn on walls, doors or traps.

- These steps must be taken in order to avoid awkward generation

Builder: Director

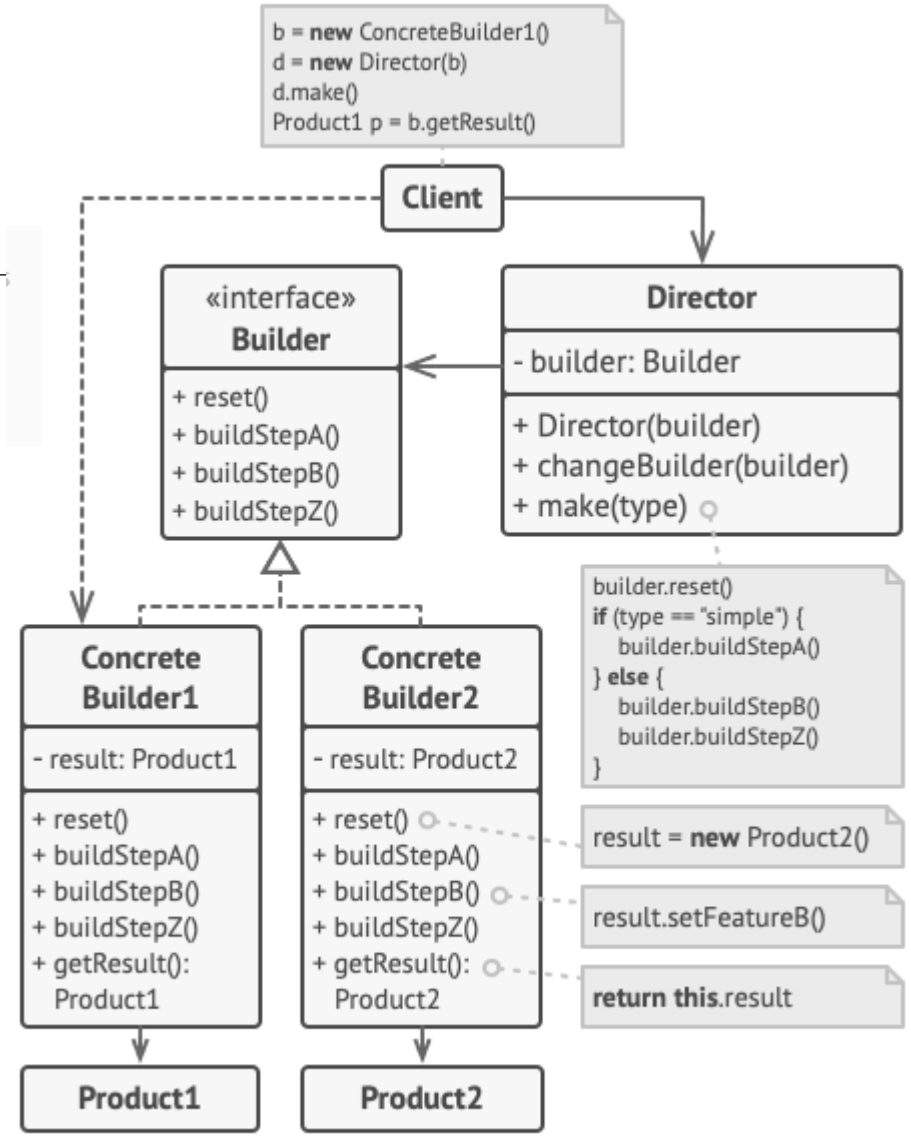
```
public class StageDirector {
      public Stage createStage() {
         builder.createRooms();
         builder.createWalls();
         if (gameSetting.Traps == true) {
            builder.createTraps();
         }
         builder.createEnemies();
         return builder.getStage();
      }
}
```

Builder: ConcreteBuilder

```
public class EasyStageBuilder extends Builder {
      Stage stage;
      public Stage createRooms() {
         //stage.Rooms…
      }
      public Stage createEnemies() {
         for (Room room: Rooms) {
            //room.Enemies… create few, easy enemies
         }
      }
      …
}
```

Builder: Terminology

- Director: StageDirector
- Builder: EasyStageBuilder
- Client: The code that calls createStage() on a Director



Builder: Why?

- Similar motivation to Abstract Factory, but…
  - Abstract Factory builds several related objects; Builder builds one big object in several steps
  - The objects in Abstract Factory are not directly communicating to each other; in Builder, they are part of one object and can rely on each other
  - The steps in Builder may be done separately – perhaps even based on user action or other runtime factors
- Why not use a Builder?
  - Builder is necessarily more complicated than Abstract Factory to allow these interactions, driven by a Director class

Singleton

- Conceptually, some Classes must have one and only one instance
- Furthermore, it would be convenient to be able to access this instance everywhere
- Examples:
  - In a Maze game…
  - In our Cat game…
- Trying to create a new instance should automatically return the old instance
  - Constructors won't do this automatically

Singleton

```
class GamePanel {
   private static GamePanel gamePanel;
   private GamePanel() {
      //private constructor; cannot be called outside
      //set up mouse listeners, init animals…
   }
   public static GamePanel getInstance() {
      if (gamePanel == null) {gamePanel = new GamePanel();}
      return gamePanel;
   }
}
```

Singleton

- Singleton is an OOP implementation of global variables. Differences:
  - A Singleton can control access; it cannot be suddenly changed by a client
  - Java does not have global variables
- Why *not* use a Singleton?
  - Singletons are inherently not encapsulated; anyone can call them anywhere
  - Any code that depends on a Singleton requires understanding the state and behavior of the Singleton
    - It can be argued that global variables are inherently anti-OOP
  - Needs care in multithreading
  - Producing unit tests for Singleton and objects that depend on it is harder