COMP 3522

Object Oriented Programming in C++

Week 11 Day 1

Agenda

1. Design Patterns
    1. Singleton
    2. Abstract factory and dependency inversion
    3. Observer
    4. Decorator

COMP 3522

DESIGN PATTERNS

Design Patterns

- ***Design Patterns: Elements of Reusable Object-Oriented Software***
  - Written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
  - GoF (Gang of Four)
- Collection of generally **reusable** design solutions for common problems
- Formalized **best** practices
- We will look at multiple design patterns

Design Patterns

- Design patterns broken up to three main types
  - **Creational** - focus on different ways to create objects instead of direct instantiation
    - Singleton, Abstract Factory
  - **Structural** – focus on how to compose objects
    - Decorator
  - **Behavioral** – focus on communication between objects
    - Observer

SINGLETON

Design pattern: a really easy one!

- Sometimes we want to guarantee that only a **single instance** of a class will ever exist
- We want to prevent more than one copy from being constructed
- We must write code that enforces this rule
- We want to employ the **Singleton Design Pattern**

Singleton pattern

1. **Instantiates** the object on its first use
2. **Hides** a private constructor
3. **Reveals** a public getInstance function that returns a reference to a static instance of the class
4. **Disables** the copy constructor
5. **Disables** the assignment operator
6. **Provides** “global” access to a single object

Why/how do we use it?

Use the singleton pattern **when you need to have one and only one object of a type** in a system.

Singleton is a globally accessible class where we guarantee only a single instance is created

That's it.

Really, that's all there is to it.

Code sample (so easy!)

```
class singleton
{
    public:
        static singleton& get_instance()
        {
            static singleton instance;
            return instance; // Instantiated on first use.
        }
    private:
        int test_value;
        singleton() {}

    public:
        singleton(singleton const&)       = delete;
        void operator=(singleton const&)  = delete;
        int get_value() { return test_value++; }
};
```

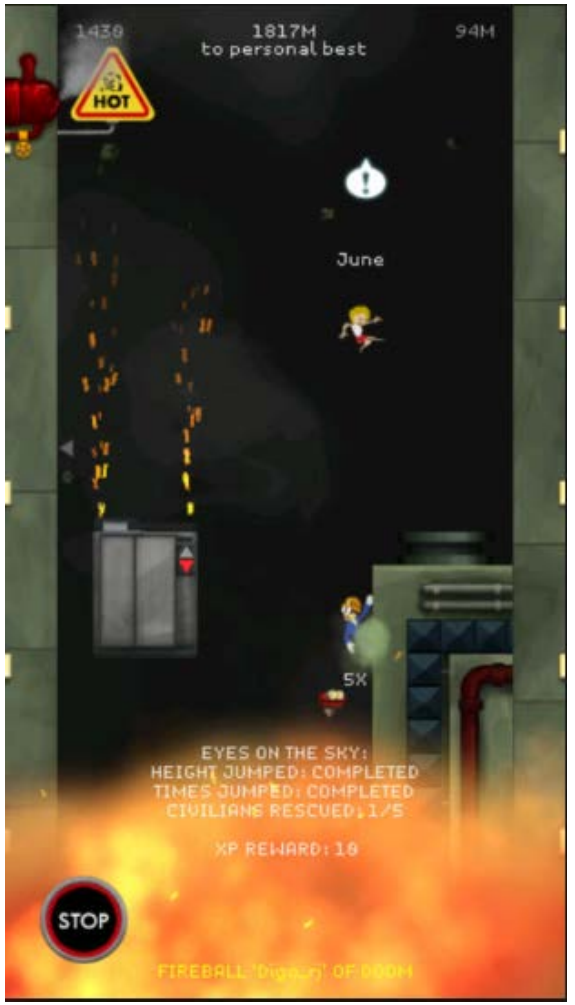
singleton.cpp

Application – Game screen management

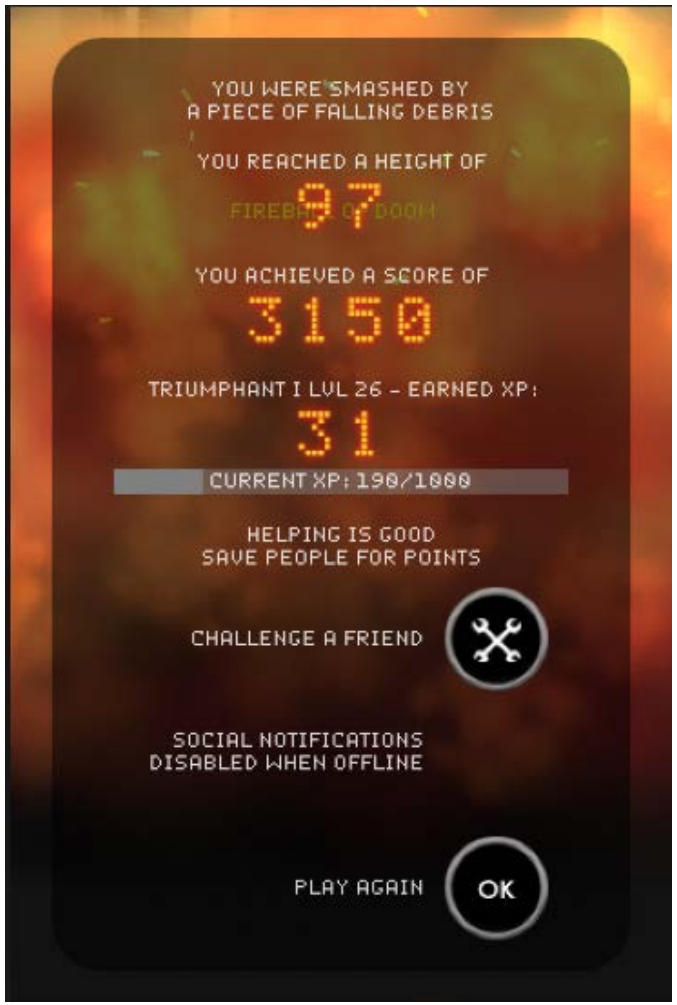
- Game has multiple screens
  - Start, gameplay UI, game over, store, etc
- Different screens must be able to be displayed at various places in the code
  - Store class wants to show store screens
  - Gameplay logic wants to show start/gameplay/game over
  - Settings logic wants to show store screen
- Need a central place to call to load specific screens on demand

Mechanic Panic – Singleton screens example

GameState: Main menu
ScreenManager::getInstance.show(MainMenu);



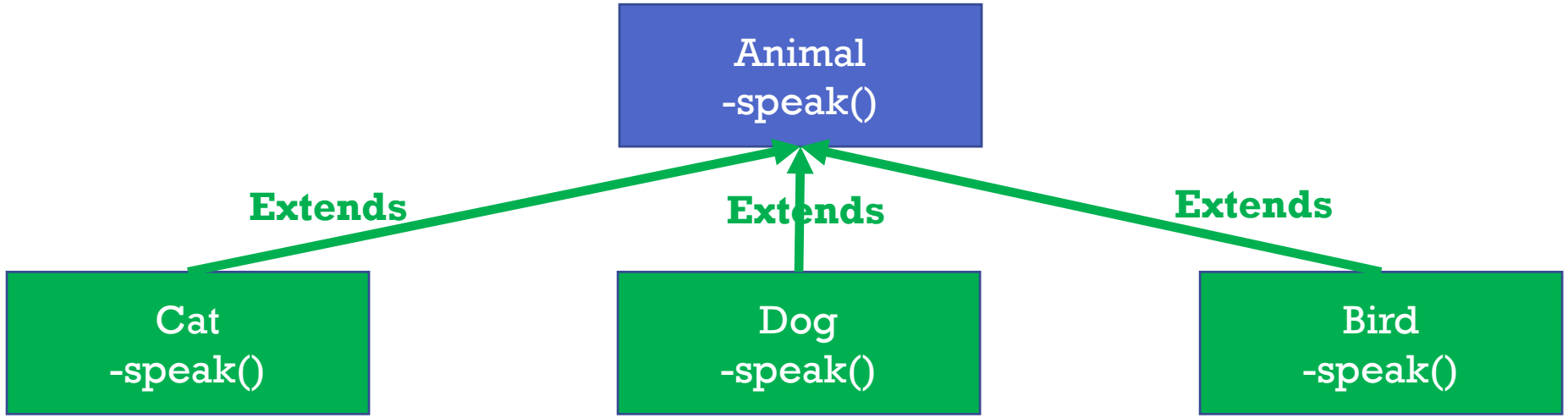
GameState: Gameplay
ScreenManager::getInstance.show(Gameplay);



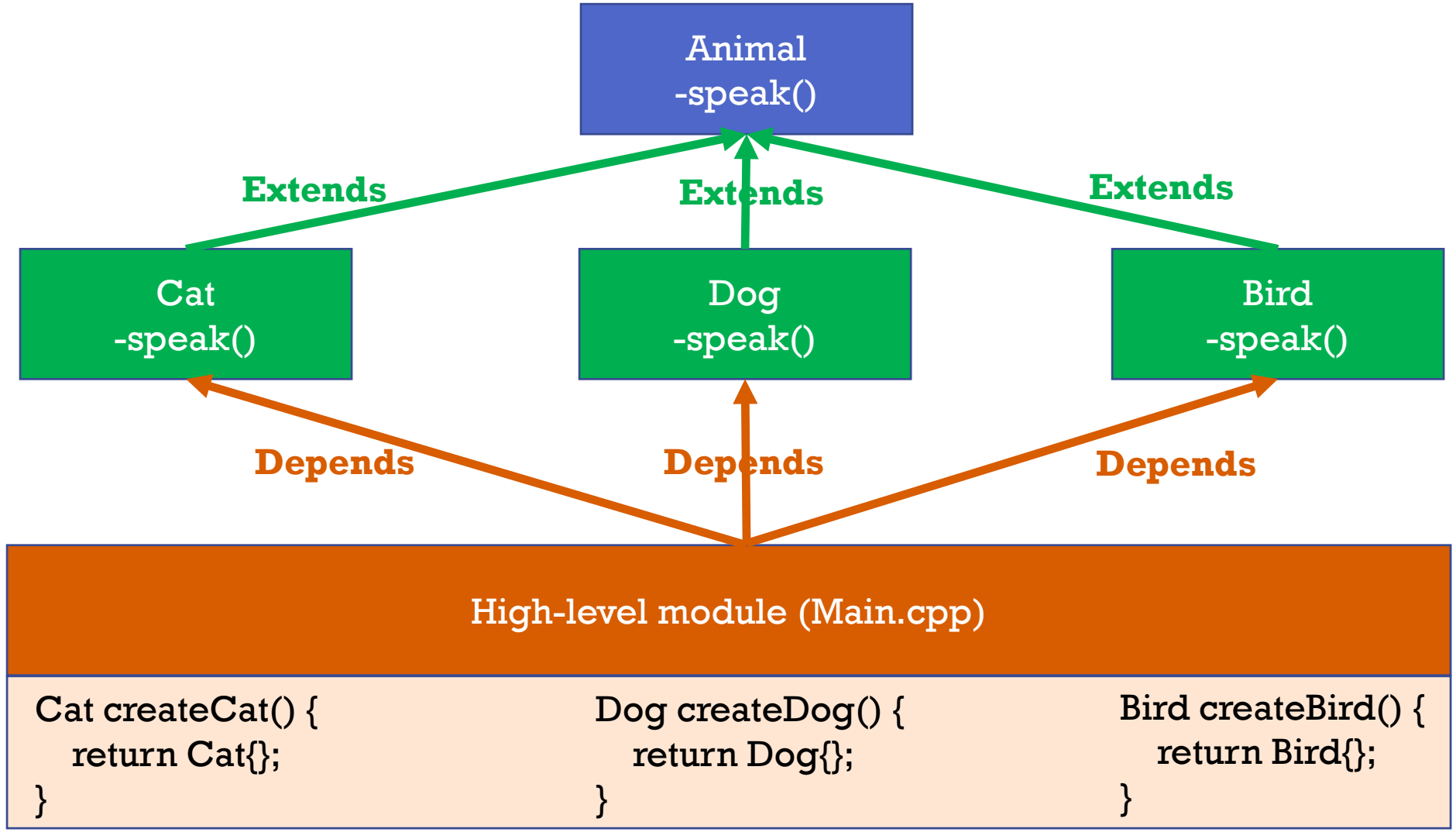
GameState: GameOver
ScreenManager::getInstance.show(Gameover);

ABSTRACT FACTORY

Introduction



Introduction



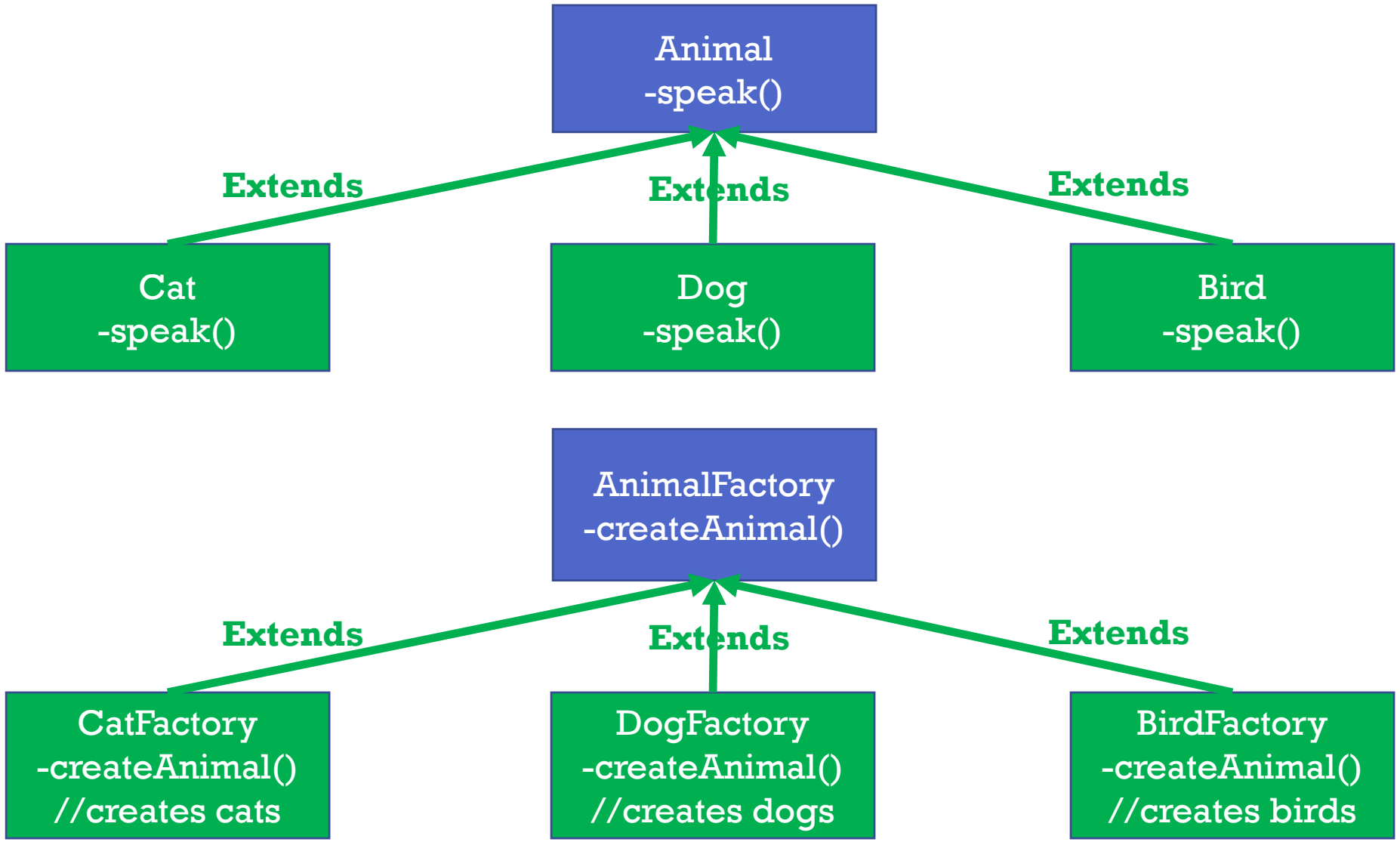
What is dependency inversion?

- The ***Dependency Inversion Principle*** (DIP) tells us that a system is more flexible when source code dependencies refer to abstractions
- Nothing concrete should be depended on
- In statically typed languages like Java and C++, this means that everything we use, import, and include should only refer to modules that contain abstractions like interfaces and abstract classes
- **We use polymorphism to gain control over every source code dependency**

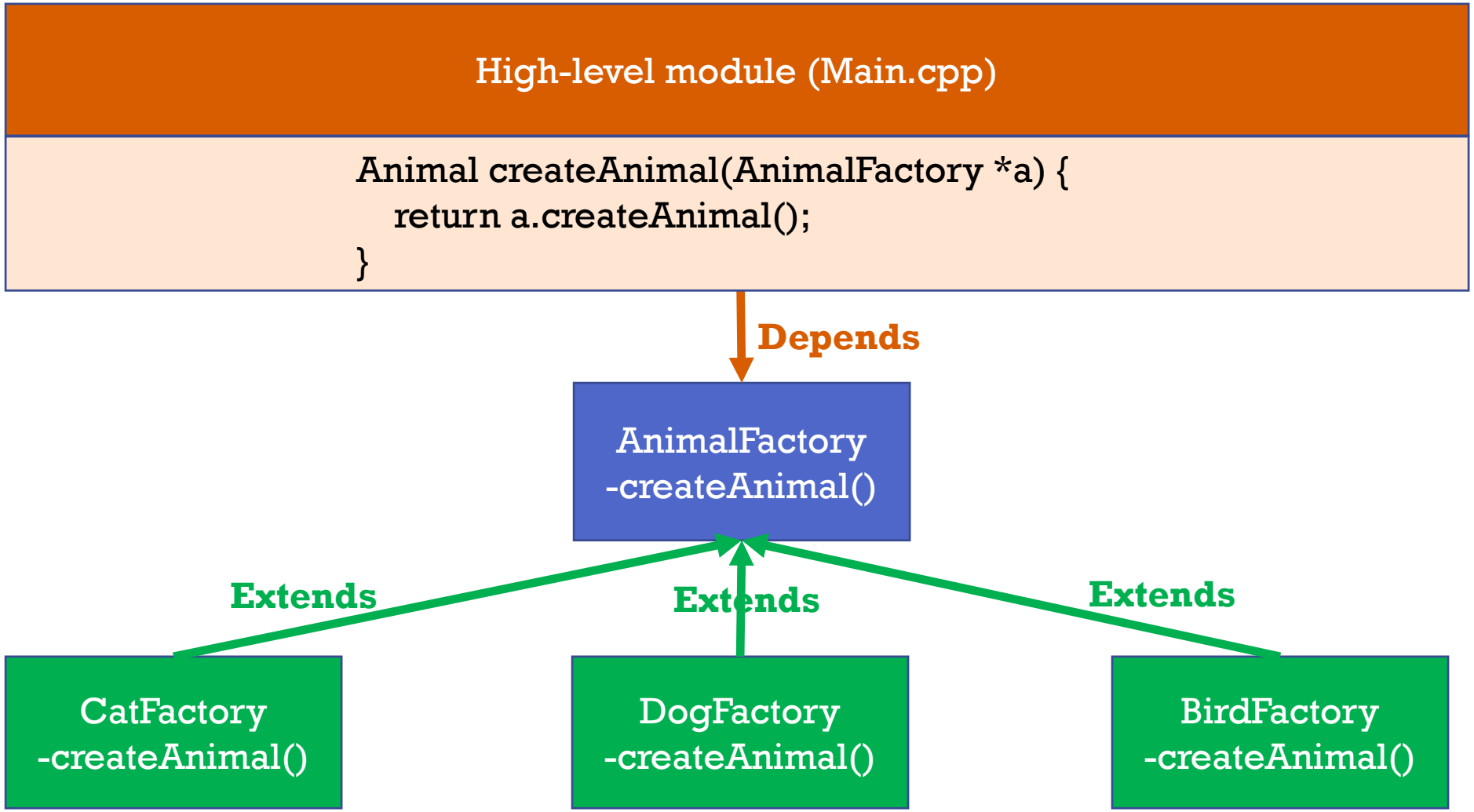
So what is it, again?

1. When source code dependencies point in the opposite direction compared to the flow of control
2. Classes and methods depend on abstractions, not concretions
3. High level modules do not depend on low level modules

Factory


Animal
-speak()
Extends
Extends
Extends
Cat
-speak()
Dog
-speak()
Bird
-speak()
AnimalFactory
-createAnimal()
Extends
Extends
Extends
CatFactory
-createAnimal()
//creates cats
DogFactory
-createAnimal()
//creates dogs
BirdFactory
-createAnimal()
//creates birds

Dependency Inversion



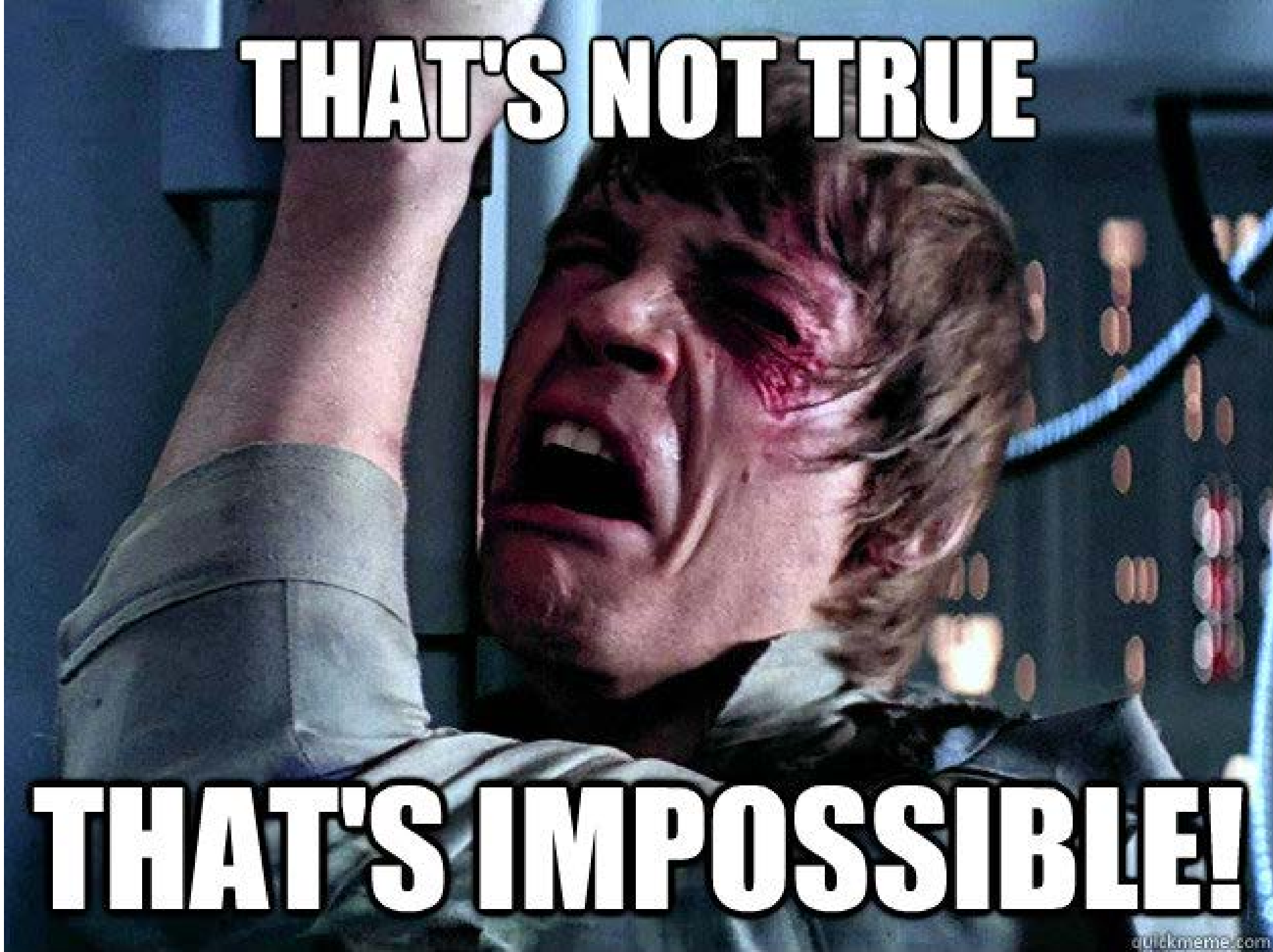
Why is this so?

- Changing an abstraction means changing all its concrete implementations
- Changing a concrete implementation doesn't usually require changing the interface(s) it implements
- We say that interfaces are less volatile than implementations
- Stable software architectures
  - Avoid depending on volatile concrete classes
  - Favor the use of stable abstract interfaces.

How do we do this?

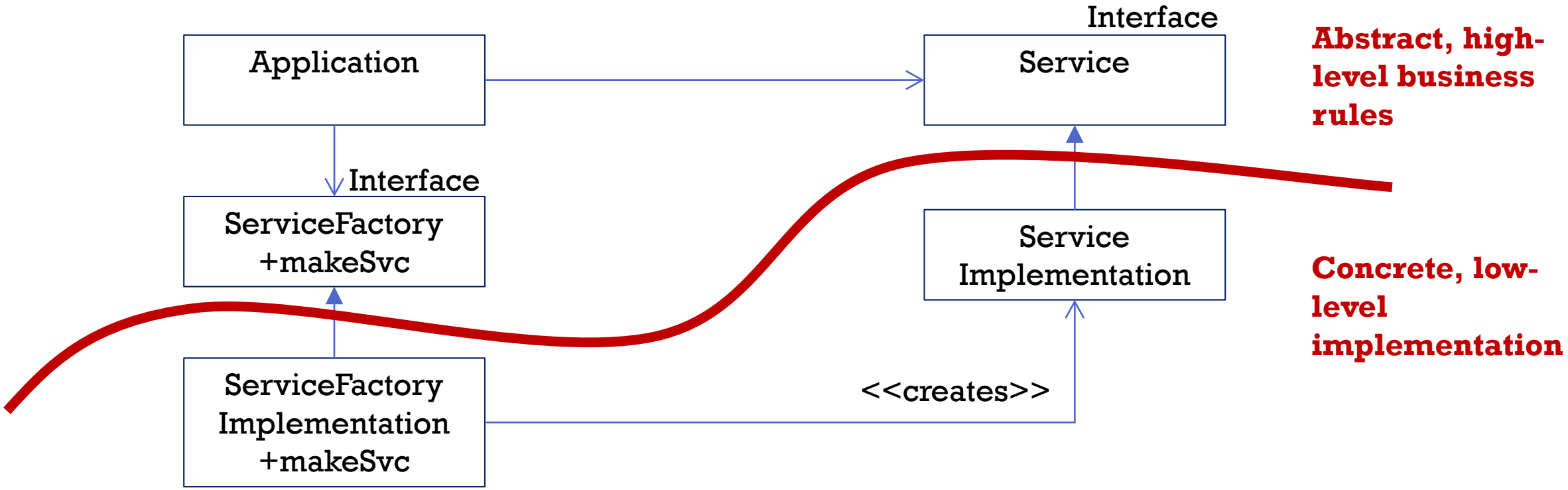
1. Don’t refer to volatile concrete classes, refer to abstract interfaces instead
2. Don’t derive from volatile concrete classes
3. Don’t override concrete functions, make the functions abstract and create multiple implementations
4. Never mention the name of anything concrete and volatile.

- **You're right, Luke!**
- DIP violations cannot be entirely removed
- But we can gather them and keep them separate
- We **try** to put all the concrete things in main



Introducing the Abstract Factory

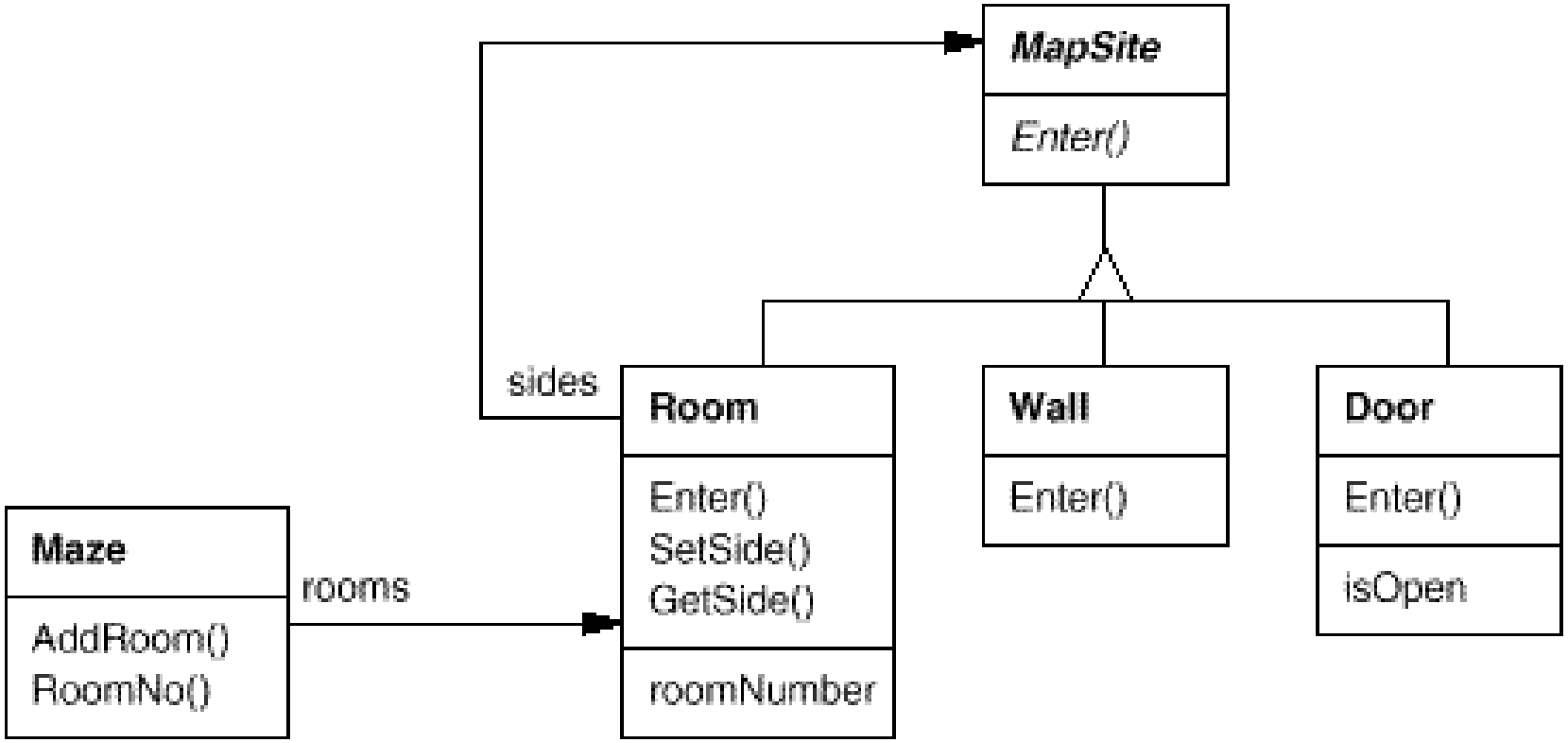
- We can use an Abstract Factory to manage this undesirable dependency



Example: making a maze

- Classic example from Gang of Four – authors of Design Patterns book
- Suppose we want to build a maze
- A maze is a set of rooms
- A room knows its neighbours, i.e., room, door, wall
- Ignore players, movements, etc.

First draft


MapSite
Enter()
sides
Room
Enter()
SetSide()
GetSide()
roomNumber
Wall
Enter()
Door
Enter()
isOpen
Maze
AddRoom()
RoomNo()
rooms

Task

Implement a function called

create_maze( )

to design a maze with 2 rooms connected by a door

```
public Maze create_maze()
{
        Maze maze;
        Room room;
        Room room2;
        Door door;
        maze.add_room(room);
        maze.add_room(room2);
        maze.add_door(door);
        return maze;
}
```

Problem

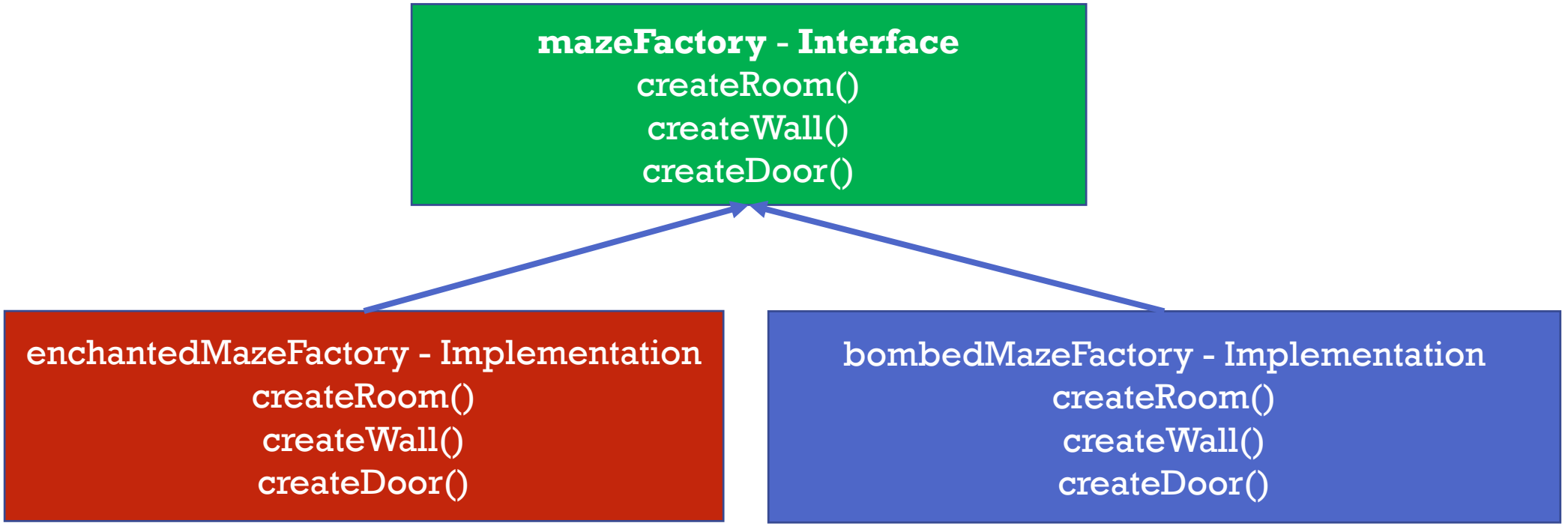
- We can only use this to create one kind of maze
- What if we want to create a different kind of maze, like an enchanted forest maze, or a haunted house maze, etc.?

- Do we need to create a different kind of create_maze function?

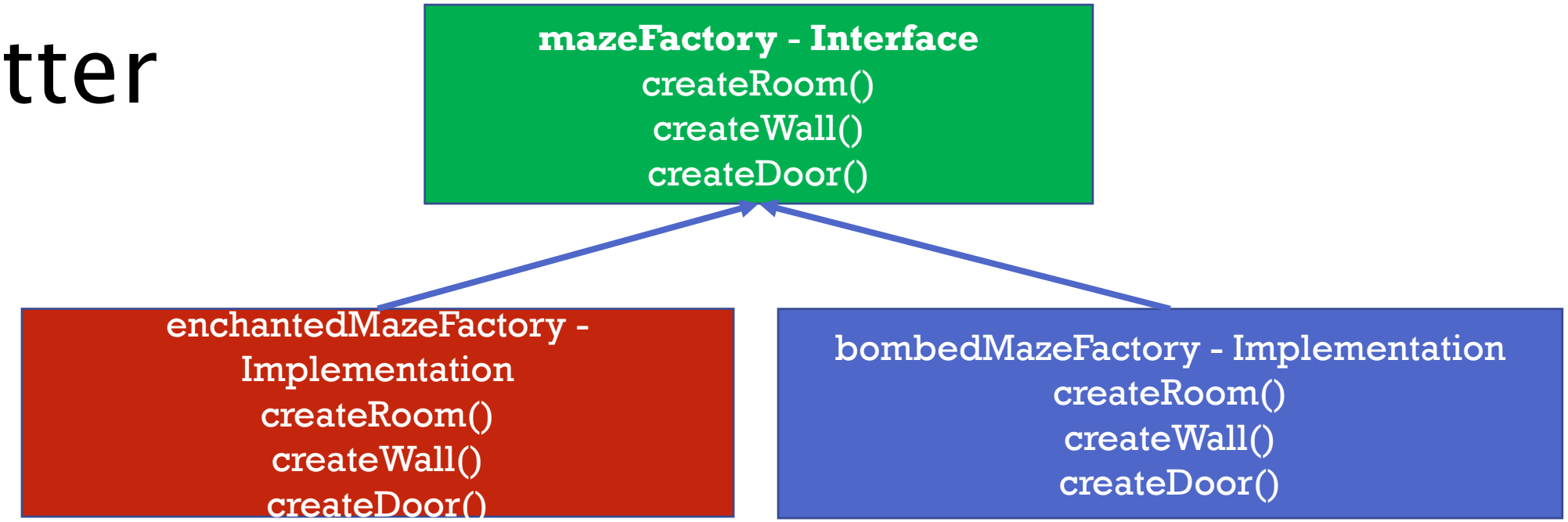
- No! Use the abstract factory pattern.

Better

- Create mazeFactory Interface
  - Create concrete classes enchantedMazeFactory & bombedMazeFactory



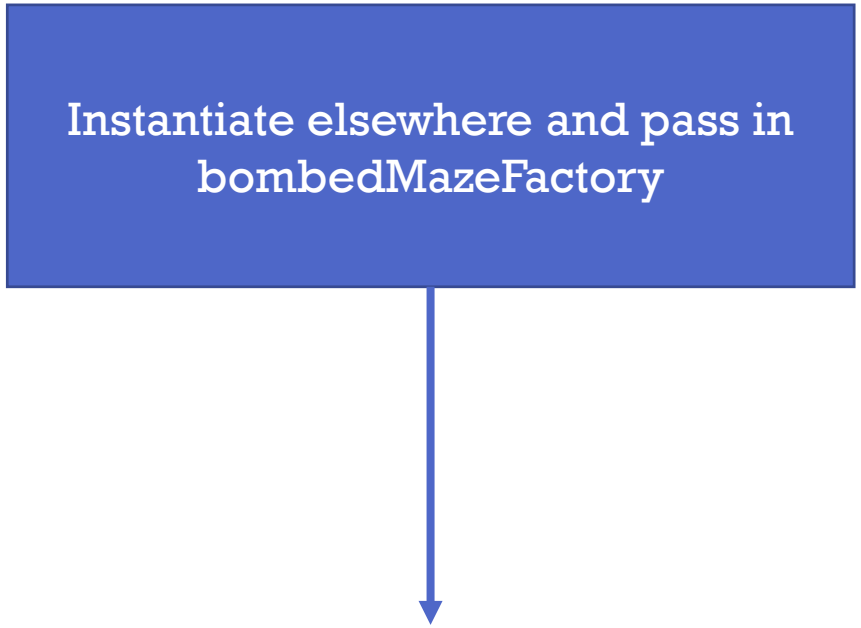
Better




```
public Maze create_maze(maze_factory& factory)
{
//code
}
```

- Modify create_maze function to take abstract maze_factory as parameter

Better




```
public Maze create_maze(maze_factory& factory)
{
//code
}
```

- Pass a concrete MazeFactory implementation to create different kinds of mazes

Better

Dependency Inversion

Pass in bombedMazeFactory

```
public Maze create_maze(maze_factory& factory)
{
      Maze maze = factory.create_maze();
      Room room = factory.create_room();
      Room room2 = factory.create_room();
      Door door = factory.create_door();
      maze.add_room(room);
      maze.add_room(room2);
      maze.add_door(door);
      return maze;
}
```

- Maze with bombed components is created using polymorphism

Better

Dependency Inversion

Pass in enchantedMazeFactory

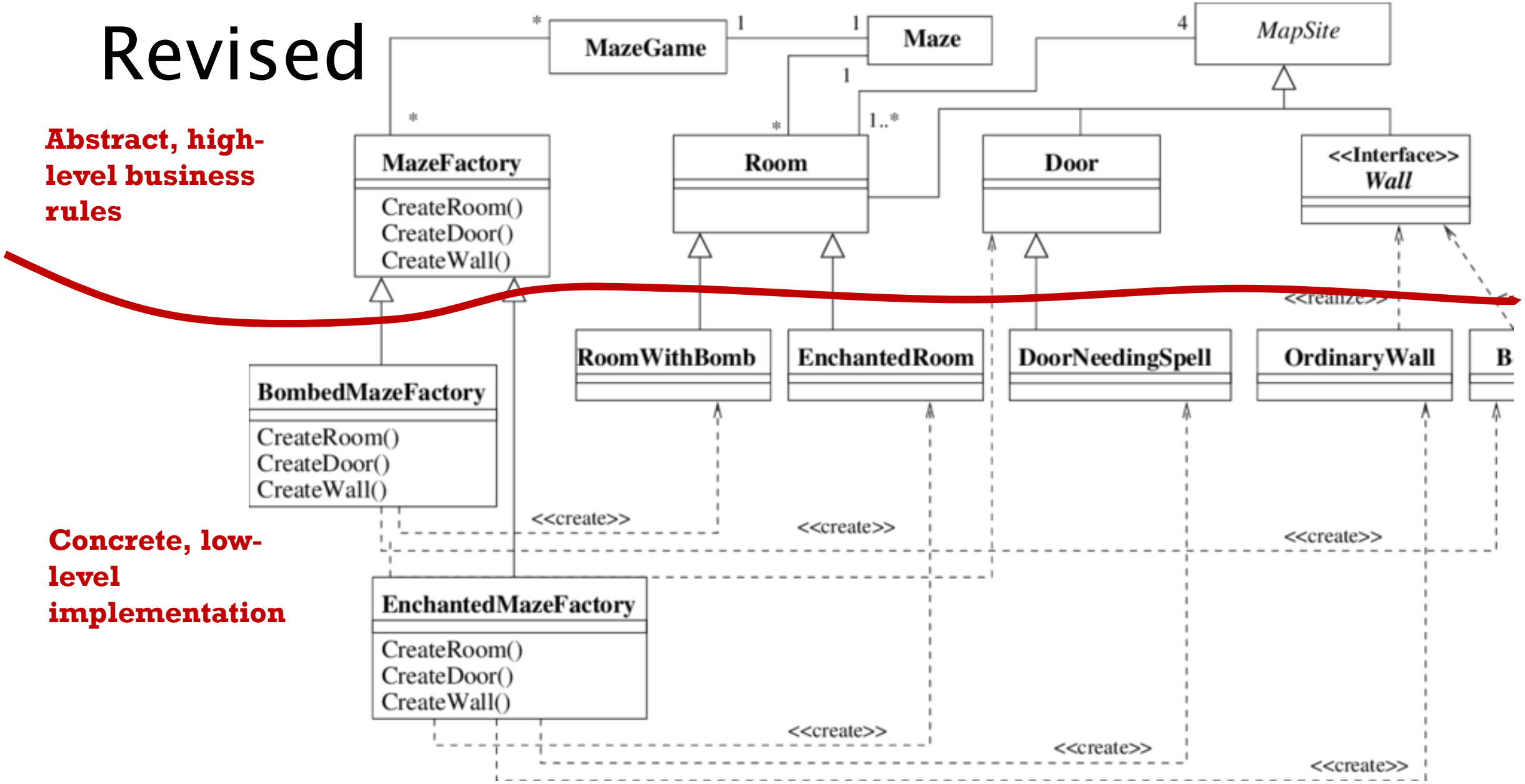
```
public Maze create_maze(maze_factory& factory)
{
      Maze maze = factory.create_maze();
      Room room = factory.create_room();
      Room room2 = factory.create_room();
      Door door = factory.create_door();
      maze.add_room(room);
      maze.add_room(room2);
      maze.add_door(door);
      return maze;
}
```

- Maze with enchanted components is created using polymorphism

Revised

Abstract, high-level business rules

Concrete, low-level implementation



Why is it better?

1. Isolates concrete classes
2. Separates business (abstraction) logic
3. Makes exchanging modules or product families easy
4. Promotes consistency among products

OBSERVER

Introduction

- We like to partition our systems into cooperating classes
- Those classes share information
- We need to maintain consistency
- But we can't couple them tightly because that reduces their flexibility
- We use an idiom you will see often in programming called Publish-Subscribe:
  - The subject publishes notifications without knowing who observes
  - Any number of observers can subscribe to receive notifications*

* Sounds a little like Java GUI listeners to me!

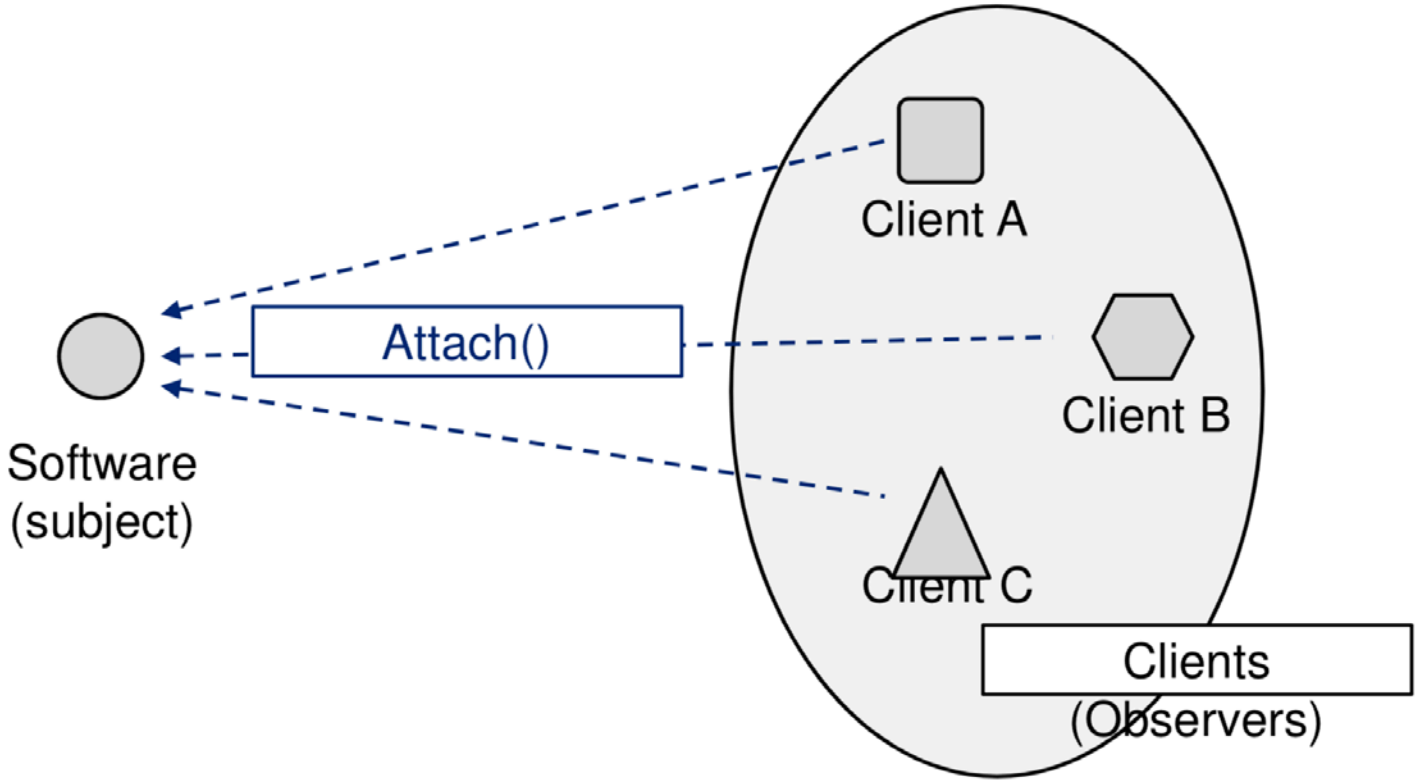
Description

- The Observer pattern describes how to establish these relationships
- There are two key objects:
  1. Subject may have any number of dependent observers
  2. Observers are all notified whenever the subject undergoes a change of state
- Each observer queries the object to synchronize their states
- Observer ensures that when a subject changes state all its dependents are automatically notified

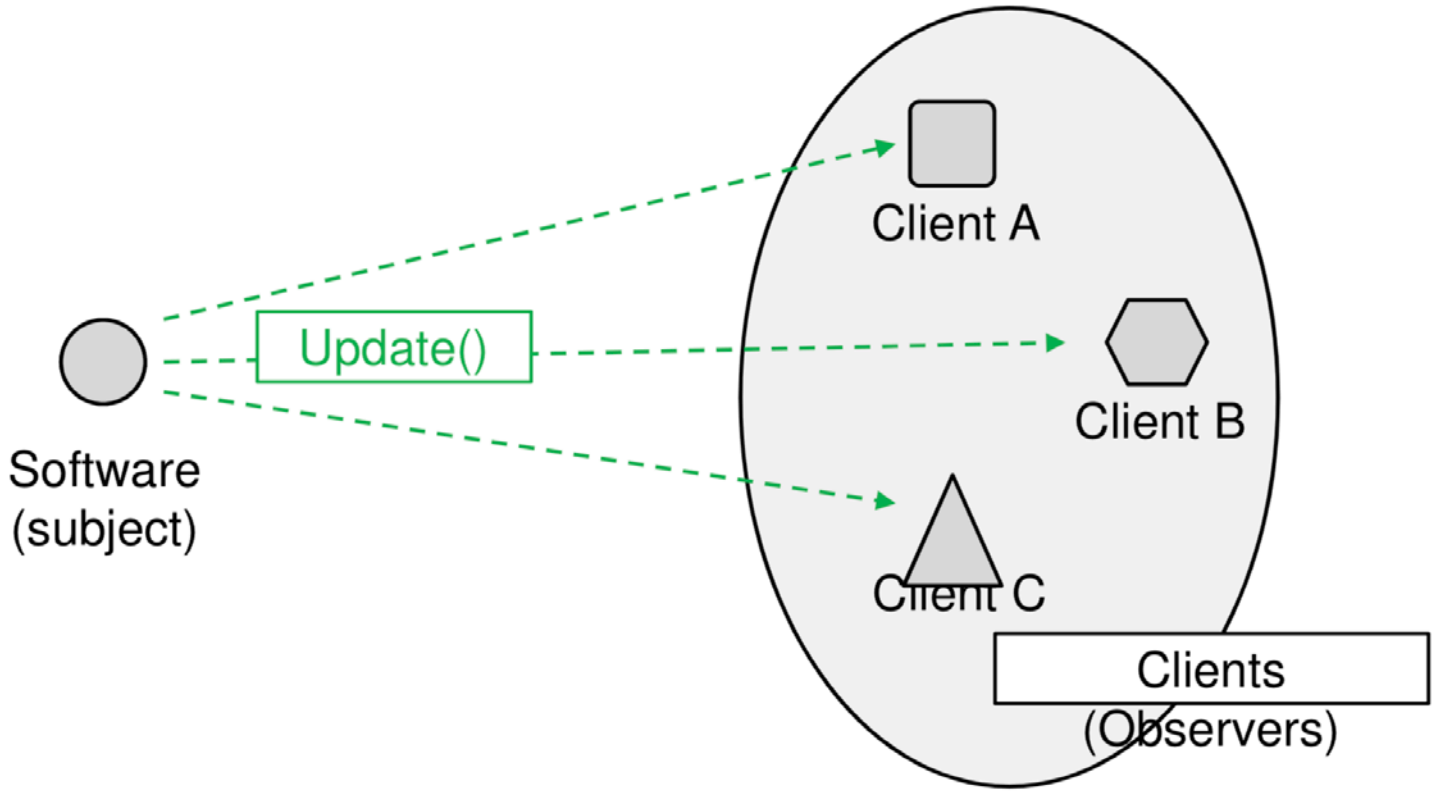
The pattern

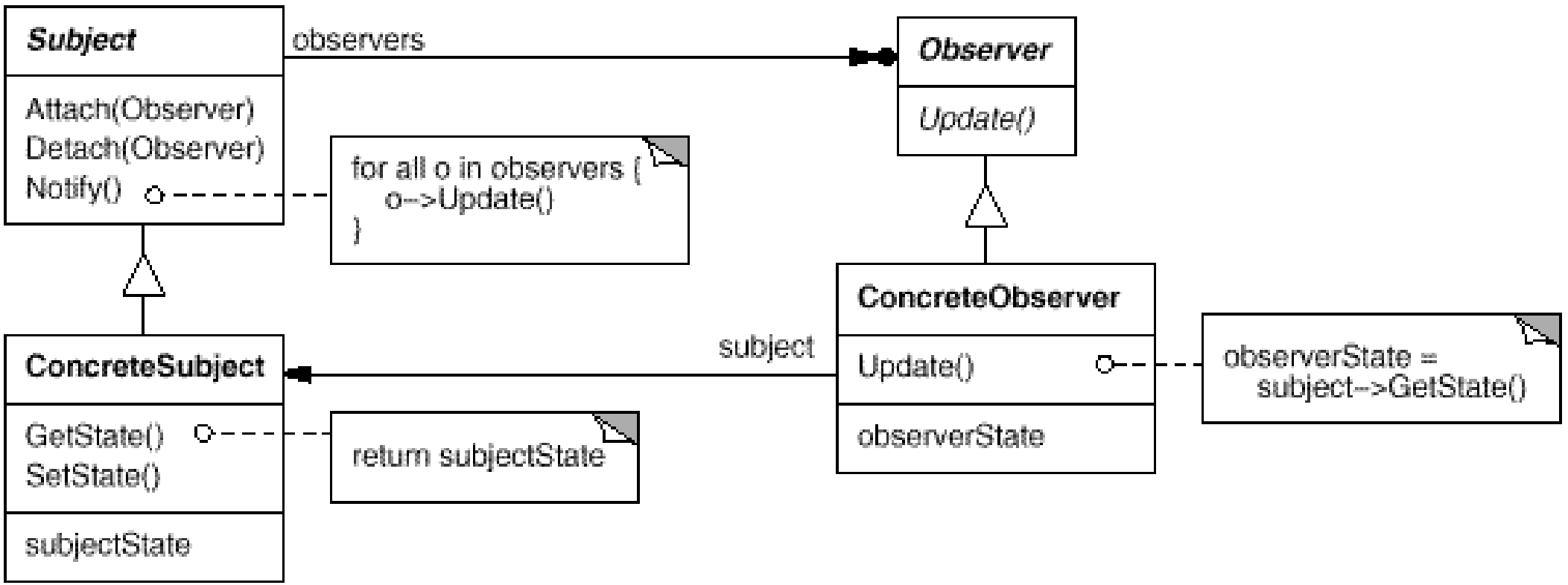
1. Subject
   - Knows its Observers
   - Any number of Observers may observe a subject
   - Provides an interface for attaching and detaching Observers
2. Observer
   - Defines an updating interface for objects that should be notified of changes in a subject
3. ConcreteSubject
   - Sends notification of its changed state to its observers
4. ConcreteObserver
   - Maintains a reference to a ConcreteSubject object
   - Stores state that needs to be consistent with the subject's
   - Implements the Observer updating interface

1. Attach


Client A
Attach()
Client B
Software
(subject)
Client C
Clients
(Observers)

2. Update


Client A
Update()
Client B
Software
(subject)
Client C
Clients
(Observers)


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for all o in observers {
o->Update()
}
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState
subject
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()

Use cases

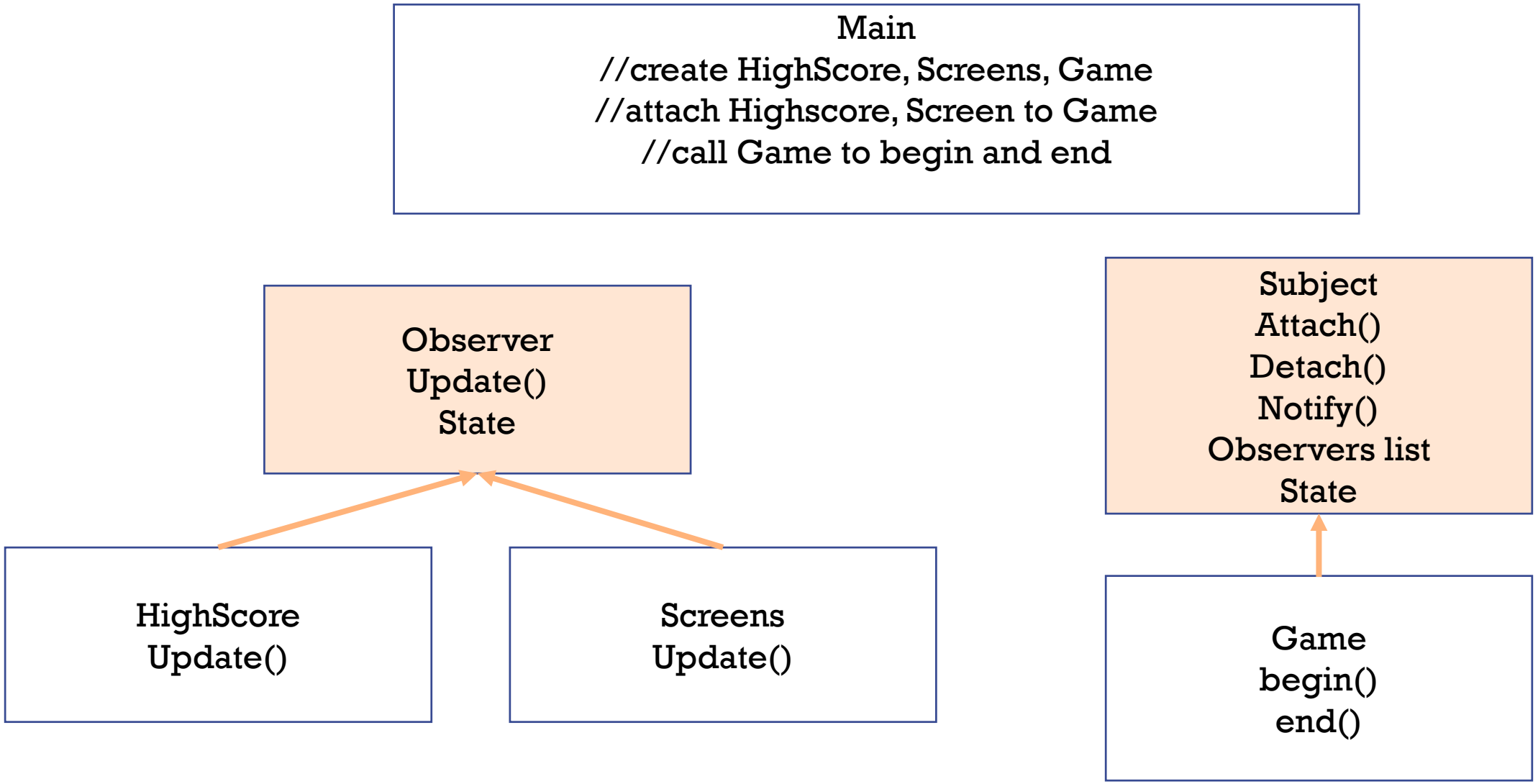
- Use the Observer pattern when:
  - A change to one subject requires changing others, and you don't know how many objects needs to be changed
  - An object needs to notify other objects without making any assumptions about what they are (loose coupling!)
- I need the professor to be notified when a student joins his/her class
- I want the display to update when the size of a window is changed
- I need the schedule view to update when the database is changed

Game example

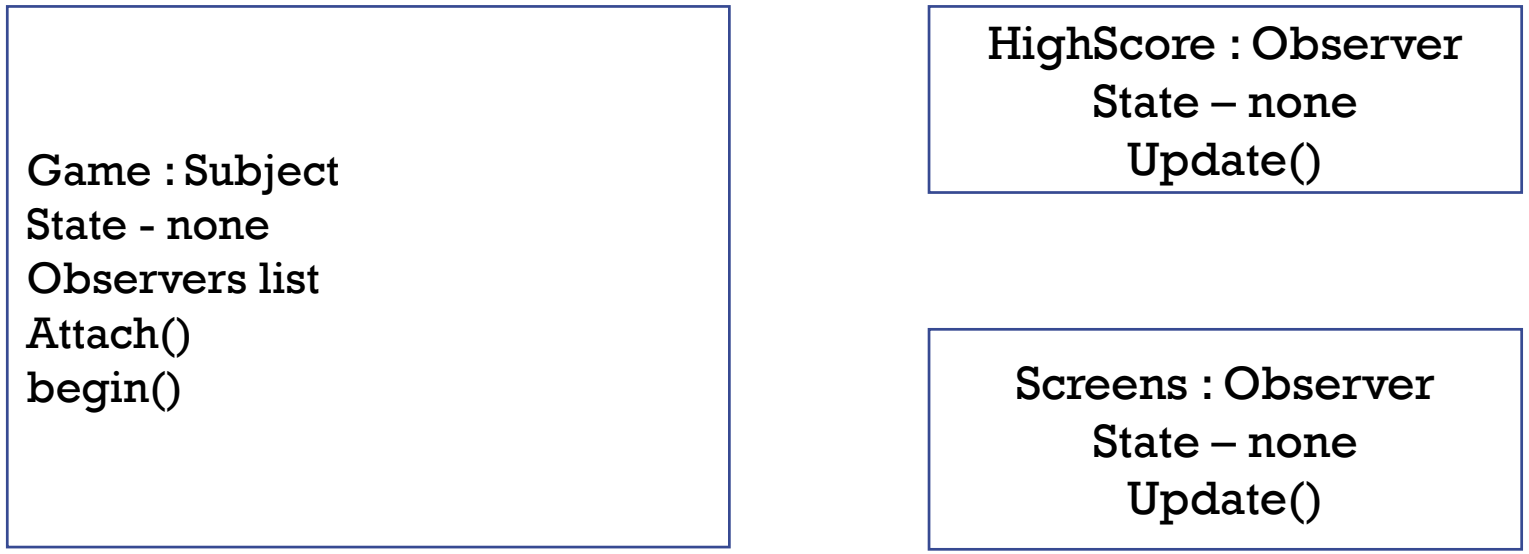
- Game class will notify observers when game begins, and game ends
  - Game begin – HighScores closed, game start screens shown
  - Game end – HighScores displayed, game end screen shown

- Game class is a Subject
  - Subject contains a vector of observers
  - All observers' update() called when subject notify() called
- HighScores and Screens class are Observers
  - All observers have an update() function
  - Perform own logic when this update function called by subject

Game example

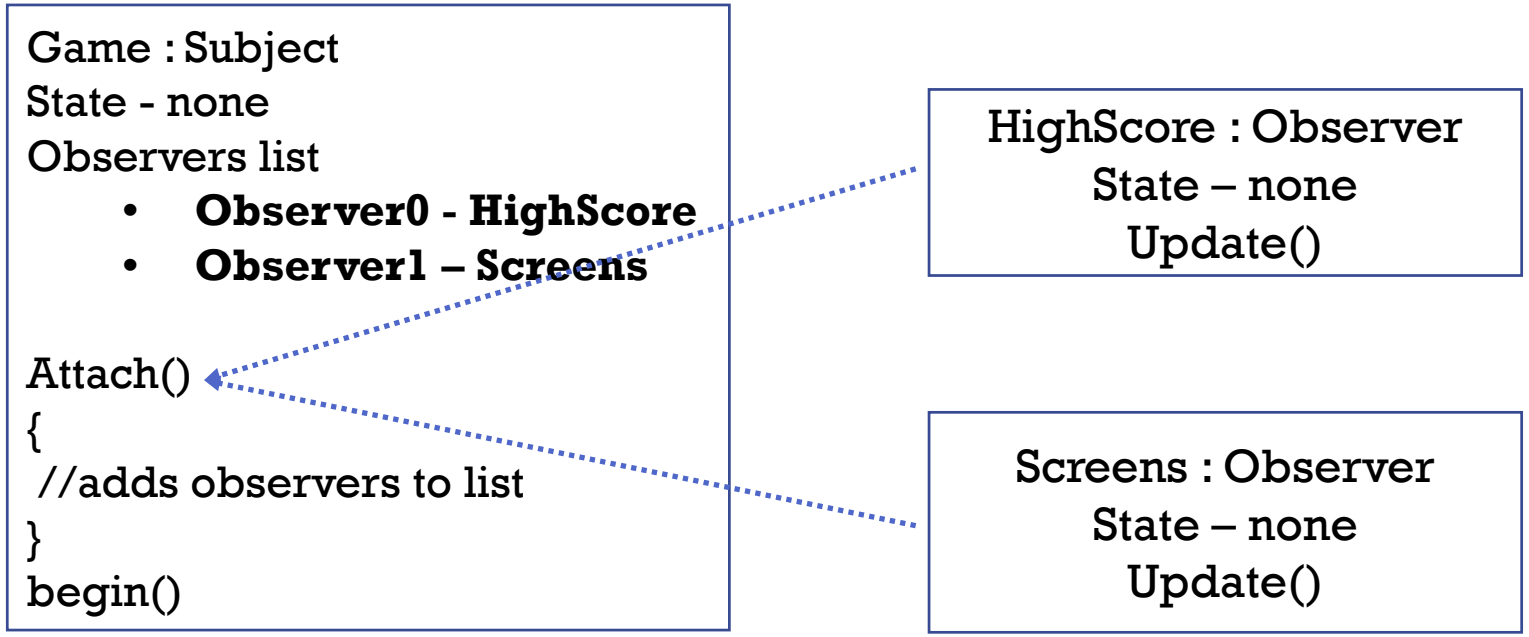


Game flow example



- Create Game, HighScore, and Screens objects

Game flow example



- HighScore and Screens are attached to game object
  - Adds them to the observers list
  - Game doesn't know they're HighScore and Screens objects
  - Game sees them as the abstract Observer type

Game flow example

Game : Subject
State - BeginState
Observers list
- Observer0 - HighScore
- Observer1 – Screens

Attach()
begin()
{
//change state, notify observers
}

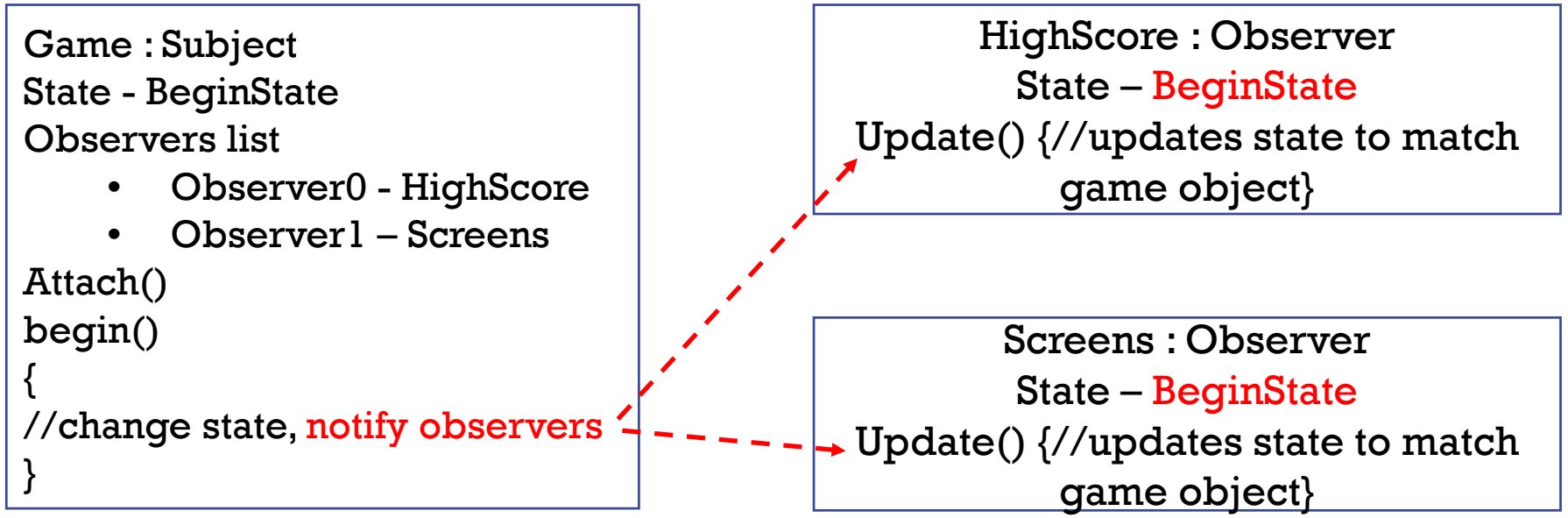
HighScore : Observer
State – none
Update()

Screens : Observer
State – none
Update()

- Game object's begin function called
  - Change the game's state

Game flow example

SubjectObserver.cpp



- Game object's begin function called
  - Change the game's state
  - Notify all observers in list that something has changed
  - Observers all query the game object to get its current state
  - Sets internal state to match game object

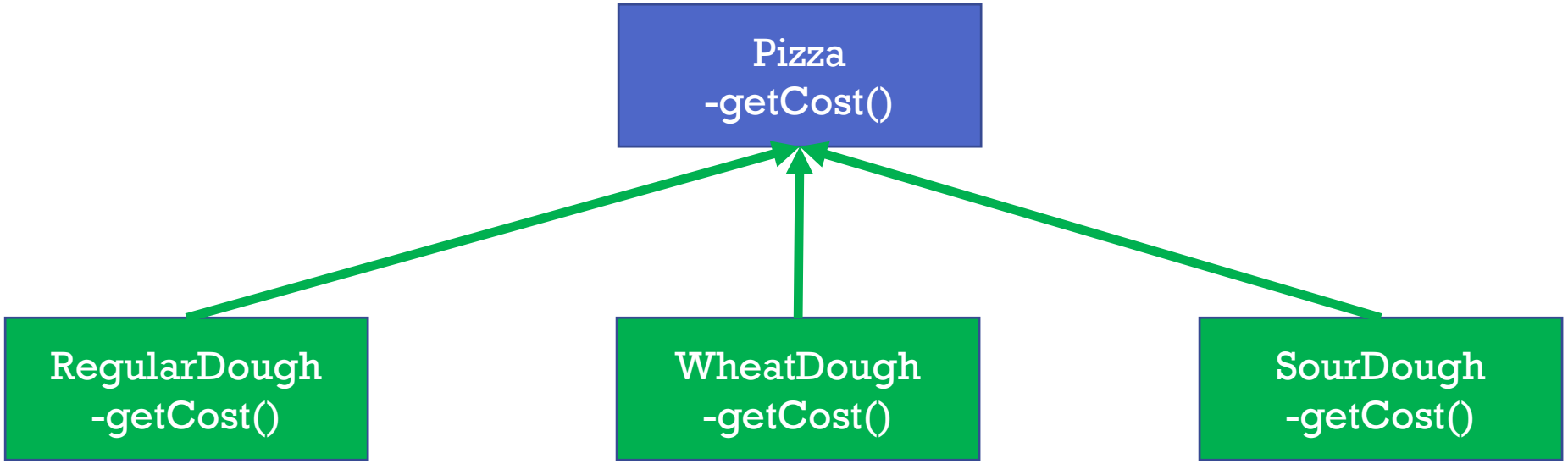
DECORATOR

Introduction - Problem

- Imagine I ask you to build a system to model a pizza shop
- You sell three different types of pizza
  - Regular dough
  - Wheat dough
  - Sour dough

Introduction - Problem

- You might model it to look something like this

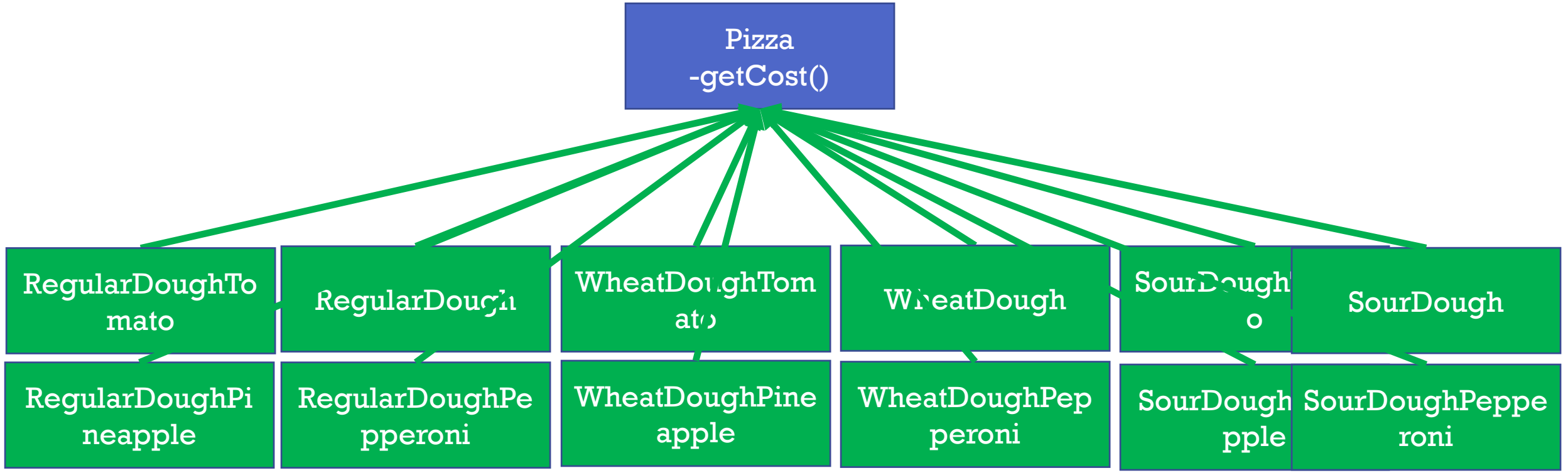


Introduction - Problem

- But then I ask you to have multiple toppings
  - Tomato
  - Pineapple
  - Pepperoni

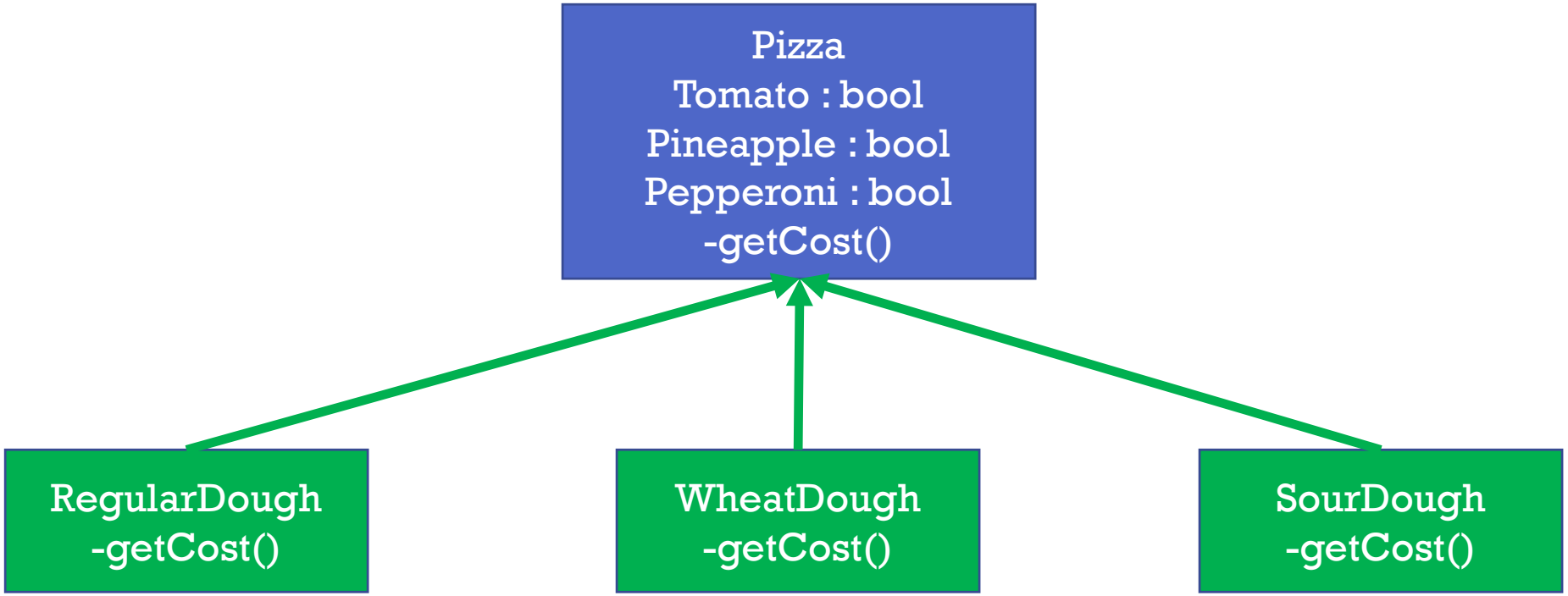
Introduction - Problem

- You might model it to look something like this (Please don't…)
- Imagine if I asked for combinations (SourDough Pineapple Tomato)



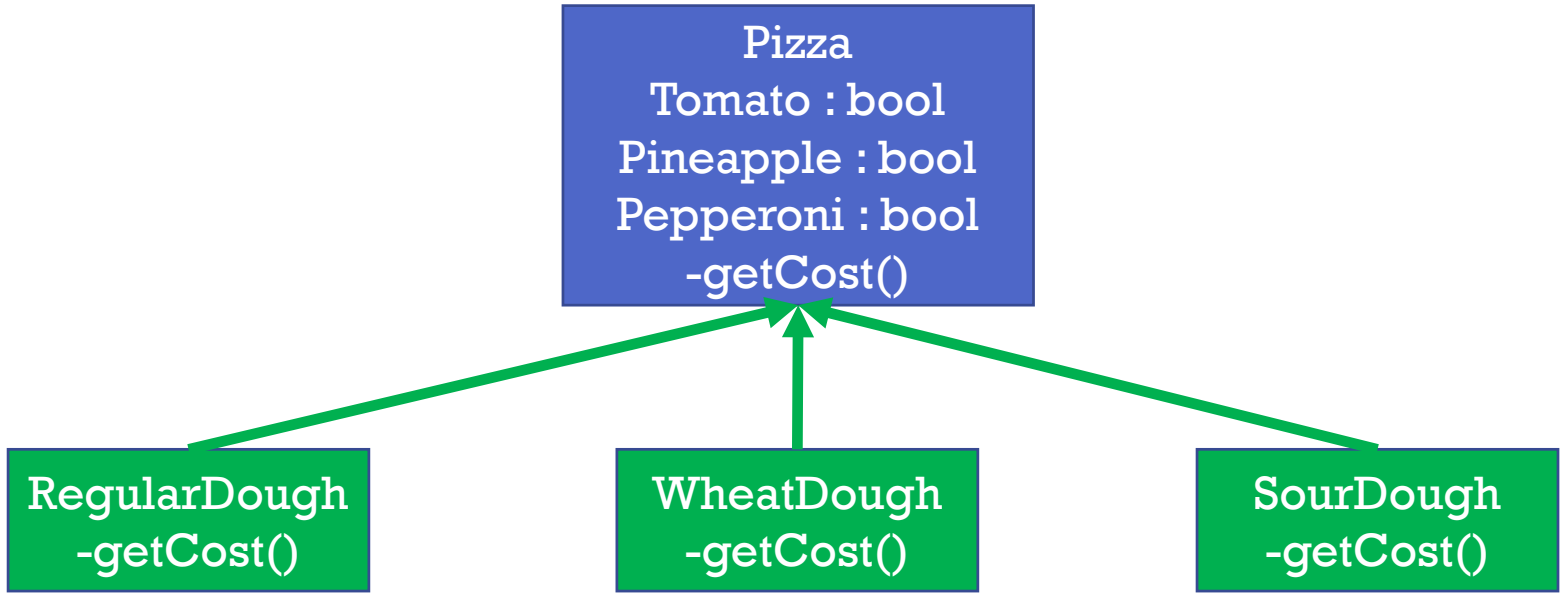
Introduction - Problem

- We can move the toppings to the Pizza instead



Introduction - Problem

- Issues
  - Violates Open-Closed Principle
  - Will need to modify parent Pizza if
    - New toppings added
    - Have double toppings
  - Some toppings might not go well with some dough, but still inherited



Introduction - Decorator

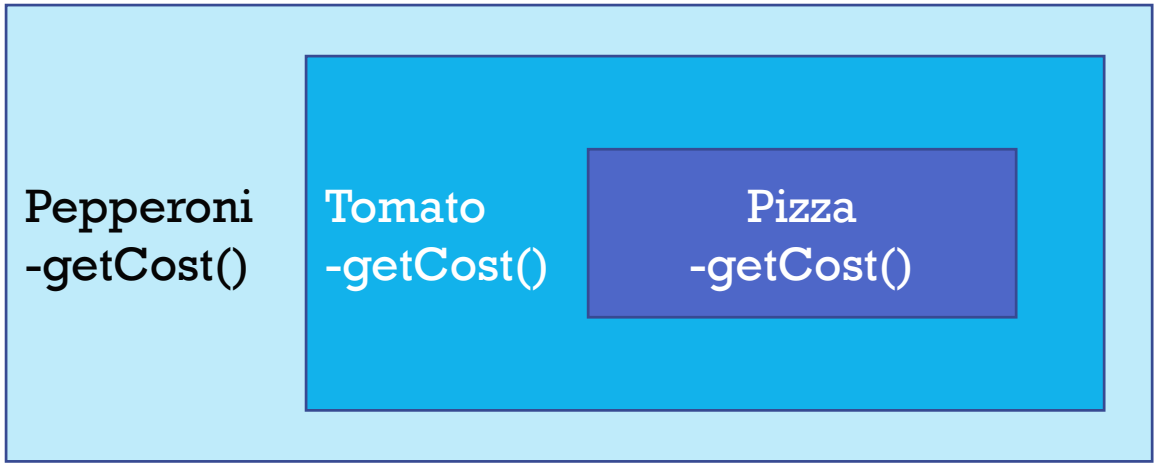
- Let's try something different
- Let's start with a pizza object

Introduction - Decorator

- We'll "decorate" our pizza object with toppings
  - Wrap our pizza object with Tomato topping

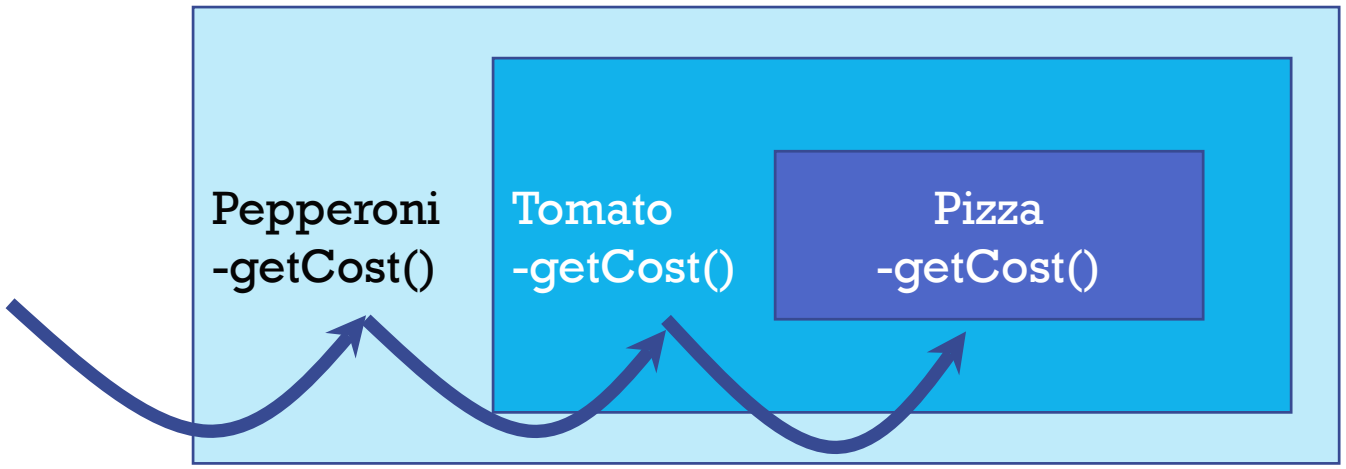
Introduction - Decorator

- We'll "decorate" our pizza object with toppings
  - Wrap our pizza object with Tomato topping
  - Wrap that with another topping, Pepperoni



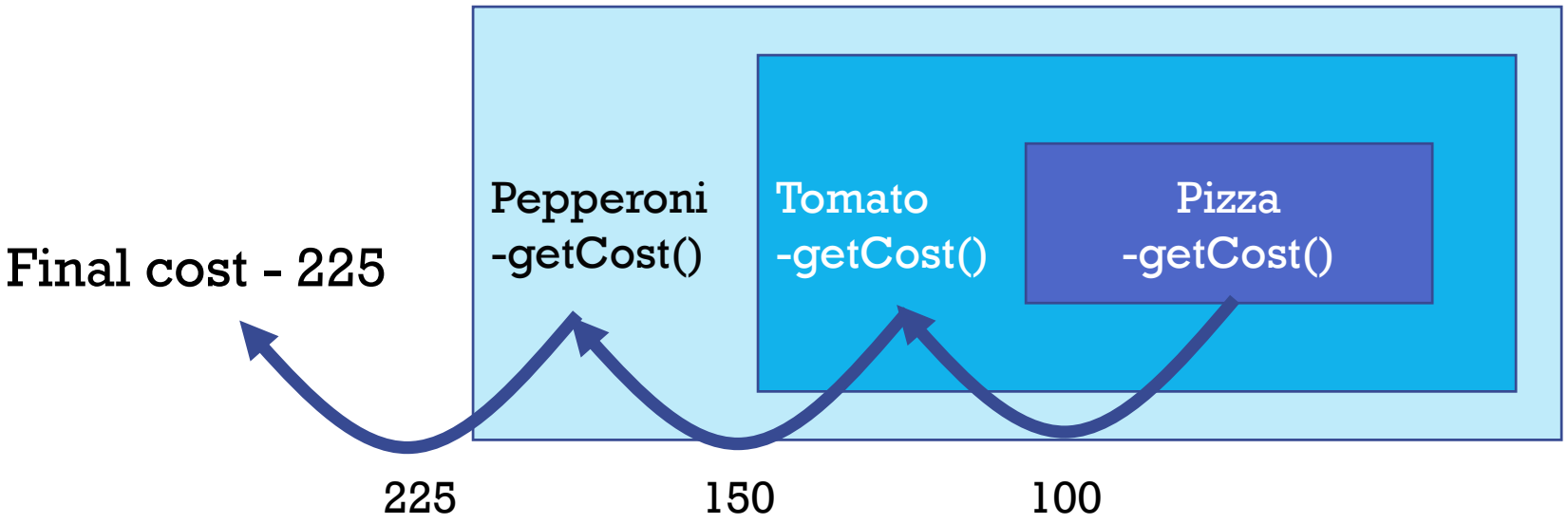
Introduction - Decorator

- Get total cost by calling getCost()
  - getCost() calls inner getCost() functions until we reach the base



Introduction - Decorator

- Get total cost by calling getCost()
  - Values are retrieved on return

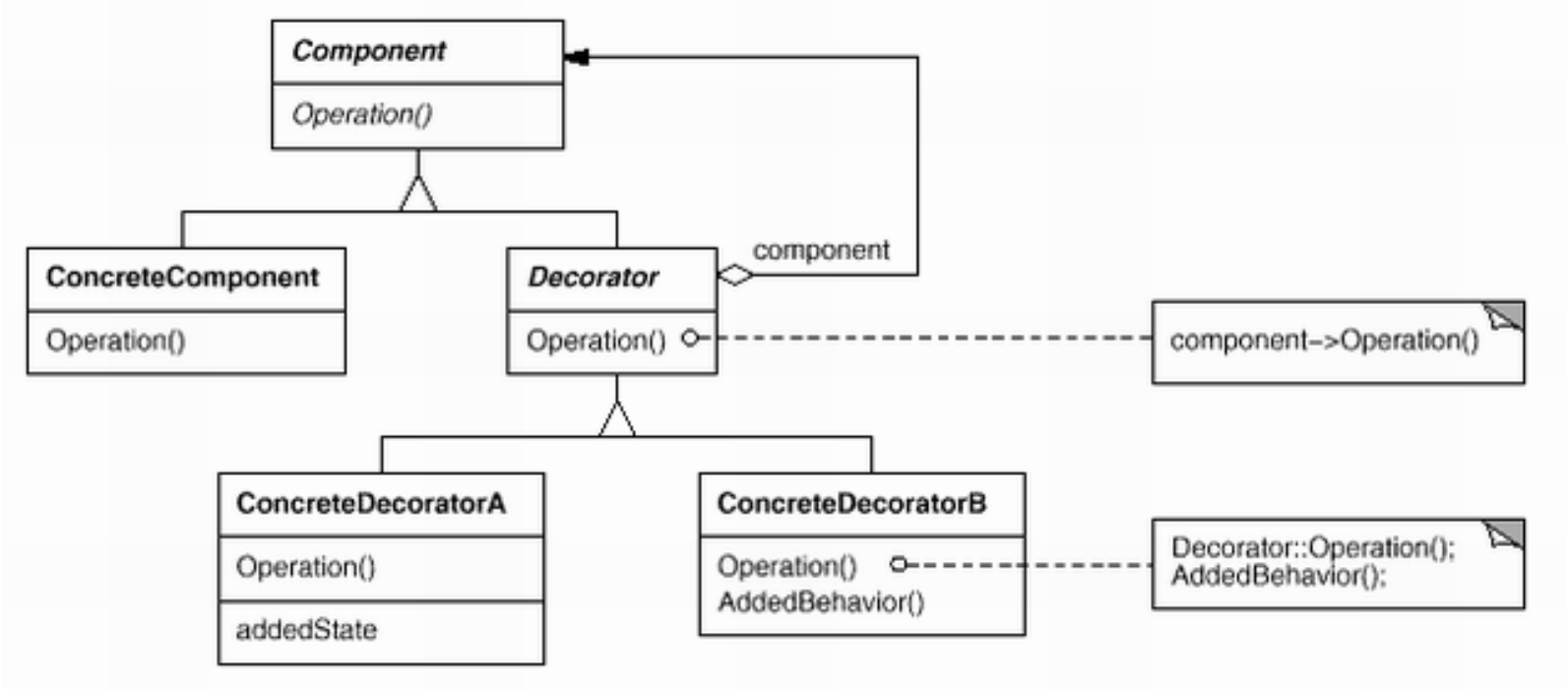


Summary

- We want to attach additional responsibilities to an object dynamically, not to an entire class
- We don’t want a huge pile of subclasses
- So we wrap it
- We want to enclose the component in another object that adds the additional responsibility
- The enclosing object is called a decorator
- The decorator conforms to the interface of the component it decorates so that its presence is transparent to clients

Description

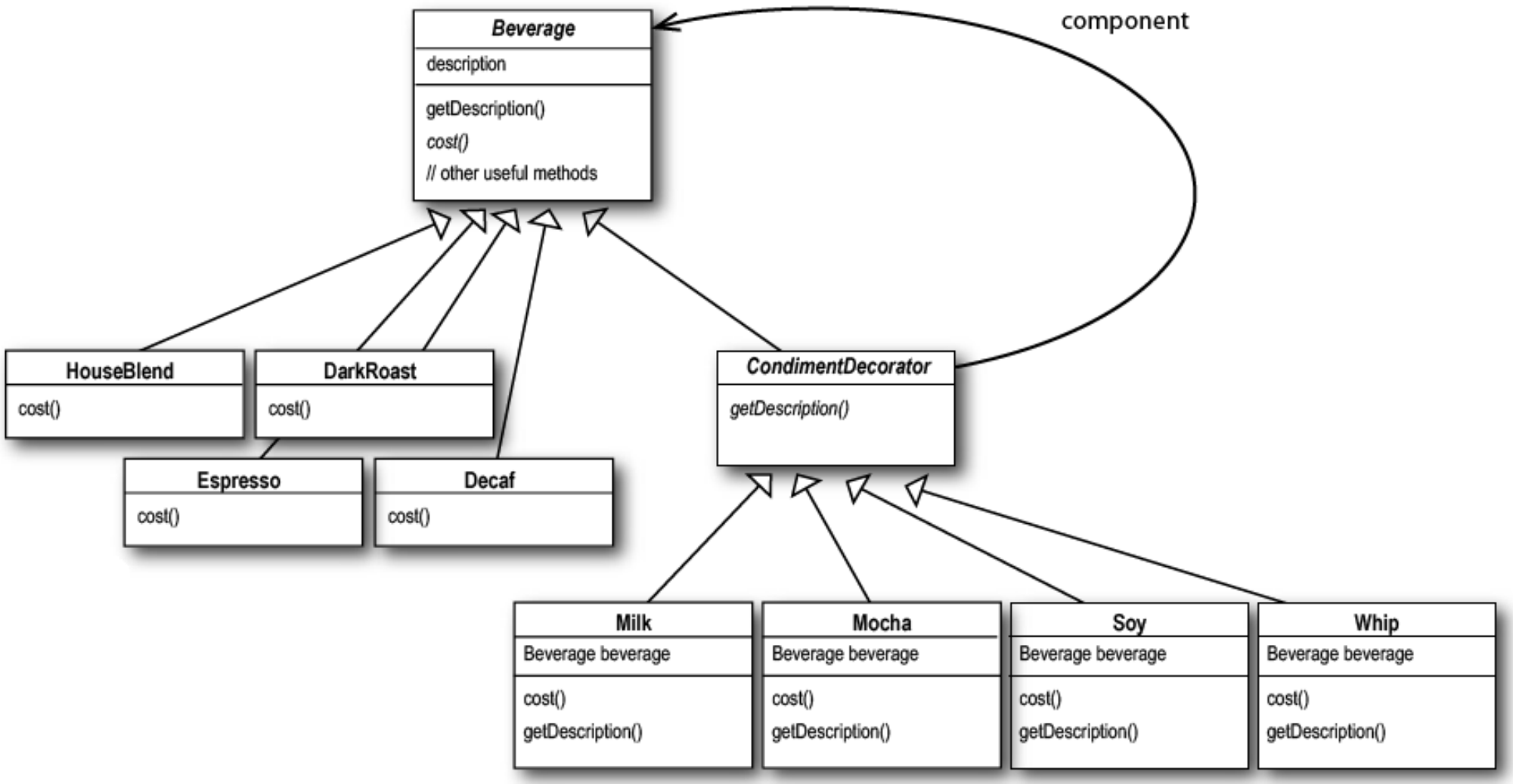
- The decorator forwards requests to the inner component and may perform additional actions before or after forwarding



Parts

- Component defines the interface for objects that have responsibilities added to them dynamically
- ConcreteComponent defines an object to which additional responsibilities can be attached
- Decorator maintains a reference to a Component object and defines an interface that conforms to Component's interface
- ConcreteDecorator adds responsibilities to the Component

Those coffees…



Benefits

- More flexible than static inheritance
  - Responsibilities can be added and removed at runtime simply by attaching and removing them
- Avoids feature-laden classes high up in the hierarchy
- Works best when we keep the Component classes lightweight:
  - Focus on defining an interface, not storing data
  - Defer the definition of data storage to subclasses.

Decorator.cpp