Object-Oriented Programming

CSCI-UA 0470-001
Class 25
Instructor: Randy Shepherd

Design Patterns

What are Design Patterns?

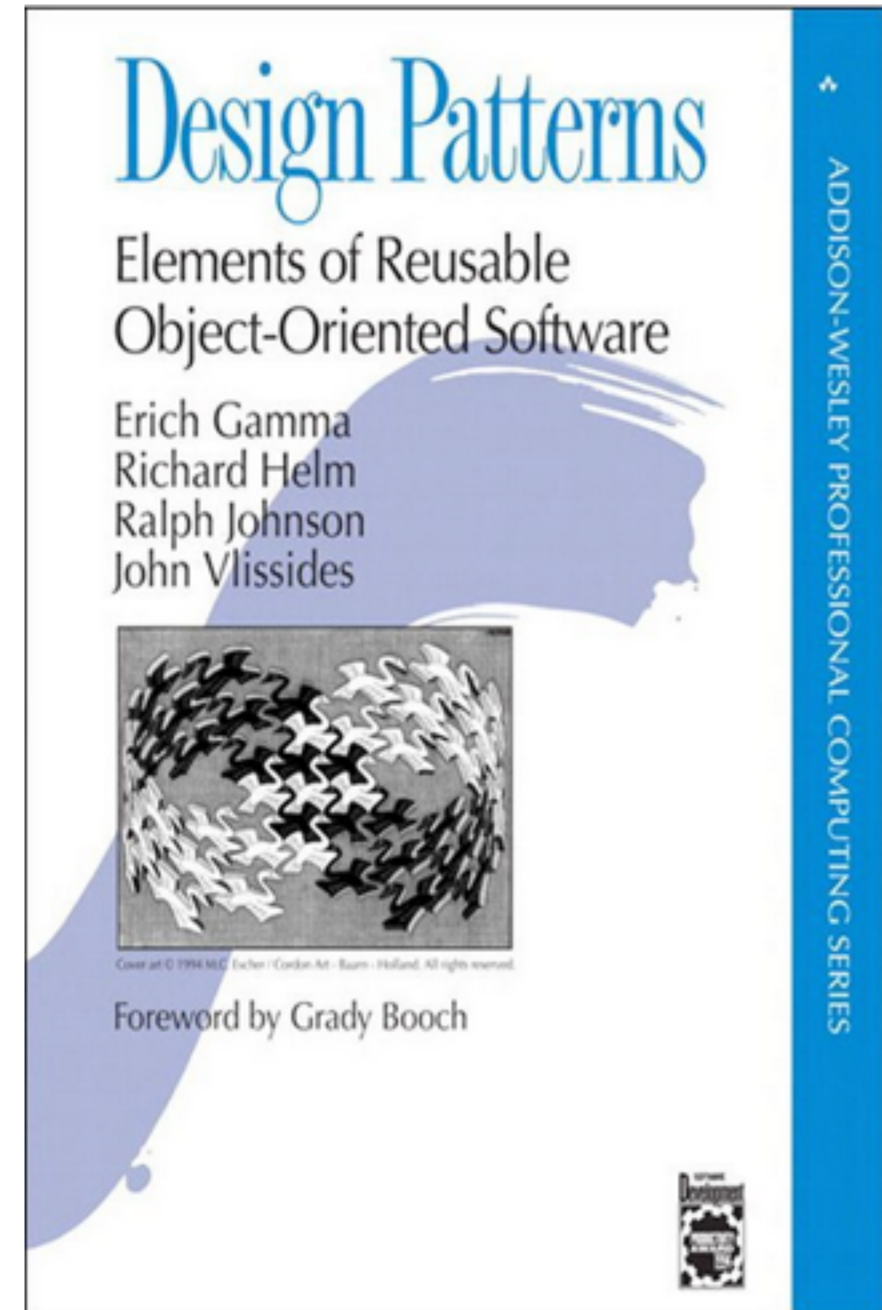
- is a general, reusable solution to a commonly occurring problem
- is abstract from programming languages
- identifies classes and their roles in the solution to a problem
- patterns are not code, only designs; must be applied

Why are Design Patterns?

- patterns are a common design vocabulary
- allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation
- patterns capture design expertise and allow that expertise to be communicated
- promotes design reuse and avoid mistakes
- improve documentation (less is needed) and understandability (patterns are described well once)

Canonical Text

- Design Patterns: Elements of Reusable Object-Oriented Software
- A classic.
- Authors known as "Gang of Four"
- If you are interested in this subject, this is the text to get.
- (Really useful knowledge for interviews!)



Delegation Pattern

- Relatively simple pattern, almost a design principle really.
- An object expresses certain behavior to the outside but in reality delegates responsibility for implementing that behavior to an associated object
- Moreover, Delegation is simply passing a duty off to some other class

Delegation Pattern

- In a way, it is ‘composition’ formalized as a pattern.
- It is one of the fundamental abstraction patterns that underlie other patterns such as the State, Decorator and Visitor Patterns.
- Its an effective strategy for buffering against changing requirements as it decouples and localizes method implementations from your primary domain objects.

Delegation Pattern

- Simple example to illustrate the mechanic.
- We have a Printer that represents the public interface to users of our library.
- However, Printer really doesn't do anything, it delegates work to RealPrinter.

```
1   // the "delegator"
2   class Printer {
3       // create the delegate
4       RealPrinter p = new RealPrinter();
5
6       void print() {
7           p.print(); // calls the delegate
8       }
9
10      // to the outside world it looks like
11      // Printer actually does the work.
12      public static void main(String[] arguments) {
13          Printer printer = new Printer();
14          printer.print();
15      }
16  }
17
18  // the "delegate", actually does the work
19  class RealPrinter {
20      void print() {
21          System.out.println("something");
22      }
23  }
```

Delegation Pattern

- Print represents the external interface, we've encapsulated RealPrinter.
- RealPrinter can be swapped out for "NetworkPrinter" without affecting users of Printer.
- Hmmm… 'interface'

```
1  // the "delegator"
2  class Printer {
3      // create the delegate
4      RealPrinter p = new RealPrinter();
5
6      void print() {
7          p.print(); // calls the delegate
8      }
9
10     // to the outside world it looks like
11     // Printer actually does the work.
12     public static void main(String[] arguments) {
13         Printer printer = new Printer();
14         printer.print();
15     }
16 }
17
18 // the "delegate", actually does the work
19 class RealPrinter {
20     void print() {
21         System.out.println("something");
22     }
23 }
```

Delegation Pattern

- Delegation can be combined with interfaces to make powerfully flexible classes.
- Consider this interface and two implementations…

```
1  interface SoundBehaviour {
2      void makeSound();
3  }
4
5  class MeowSound implements SoundBehaviour {
6      public void makeSound() {
7          System.out.println("Meow");
8      }
9  }
10
11 class RoarSound implements SoundBehaviour {
12     public void makeSound() {
13         System.out.println("Roar!");
14     }
15 }
```

Delegation Pattern

- Now if we *compose* a Cat with a SoundBehavior, and *delegate* our makeSound method to it..

- We can change the behaviors of our object at runtime!

- Very powerful, yet simple technique.

```java
// A domain object that, among other things, makes sound.
class Cat {
    private SoundBehaviour sound = new MeowSound();

    public void makeSound() {
        this.sound.makeSound();
    }

    public void setSoundBehaviour(SoundBehaviour newsound) {
        this.sound = newsound;
    }

    public void otherCatThings() {
        //...
    }
}

// We can now compose behavior of cats at runtime!
public class Cats {
    public static void main(String args[]) {
        Cat c = new Cat();
        c.makeSound();  // Output: Meow
        // now to change the sound it makes
        c.setSoundBehaviour(new RoarSound());
        c.makeSound(); // Output: Roar!
    }
}
```

Decorator Pattern

- The Decorator pattern..
  - is an example of a pattern that is built upon Delegation
  - allows you to add *new* behavior to other objects at *runtime*.
  - is also known as "Wrapper". It provides its functionality by *wrapping* itself around the original object.

Decorator Pattern

- A Decorator object..
  - maintains the same interface as the object it 'decorates'
  - accepts the object it is 'decorating' as an argument in its constructor.
  - keeps this original object in a private member variable (thus 'wrapping' it)

Decorator Pattern

- We define our interface for a coffee, it has two methods.

- Along with a SimpleCoffee, which has no additional ingredients, and a cost of $1

```java
// The abstract Coffee class defines the
// functionality of Coffee implemented by decorator
interface Coffee {
    // Returns the cost of the coffee
    double getCost();
    // Returns the ingredients of the coffee
    String getIngredients();
}

// Implementation of a simple coffee
// without any extra ingredients
class SimpleCoffee implements Coffee {
    public double getCost() {
        return 1;
    }

    public String getIngredients() {
        return "Coffee";
    }
}
```

Decorator Pattern

- Here is a decorator.
- Note that it takes a "Coffee" as an argument to its constructor
- Note that it calls the 'wrapped' coffee to get the cost, then adds the cost of its additional ingredients.

```
1  // Decorator WithMilk mixes milk into coffee.
2  class WithMilk implements Coffee {
3      private Coffee c;
4
5      public WithMilk(Coffee c) {
6          this.c = c;
7      }
8
9      public double getCost() {
10         return c.getCost() + 0.5;
11     }
12
13     public String getIngredients() {
14         return c.getIngredients() + ", Milk";
15     }
16 }
```

Decorator Pattern

- Another decorator, similar to the last.
- This one adds sprinkles to a coffee.

```
1  // Decorator WithSprinkles mixes sprinkles
2  class WithSprinkles implements Coffee {
3      private Coffee c;
4
5      public WithSprinkles(Coffee c) {
6          this.c = c;
7      }
8
9      public double getCost() {
10         return c.getCost() + 0.2;
11     }
12
13     public String getIngredients() {
14         return c.getIngredients() + ", Sprinkles";
15     }
16 }
```

Decorator Pattern

- Now in our CoffeeShop, we can create a simple coffee and *decorate* it with sprinkles and milk.
- By line 13, we have coffee implementation *wrapped* in a WithMilk decorator which is in turn *wrapped* in a WithSprinkles decorator.
- Output

```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
```

```java
1  public class CoffeeShop {
2      public static void main(String[] args) {
3          Coffee c = new SimpleCoffee();
4          System.out.println(
5            " Cost: " + c.getCost() +
6            " Ingredients: " + c.getIngredients());
7
8          c = new WithMilk(c);
9          System.out.println(
10           " Cost: " + c.getCost() +
11           " Ingredients: " + c.getIngredients());
12
13         c = new WithSprinkles(c);
14         System.out.println(
15           " Cost: " + c.getCost() +
16           " Ingredients: " + c.getIngredients());
17     }
18 }
19
```

Adapter Pattern

- The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the some other.
- Moreover, an Adapter helps two incompatible interfaces to work together.
- This is another “wrapper” pattern.

Adapter Pattern

- Suppose we have an array of ints, like so…

```
int[] a = { 123, 321, 98, 23 };
```

- And suppose further that we want to use the Java Standard Library's Collections class to sort this array…

static <T extends **Comparable**<? super T>> void	**sort**(**List**<T> list) Sorts the specified list into ascending order, according to the **natural ordering** of its elements.

Adapter Pattern

- We've seen this before!
- We can use the AbstractList skeletal implementation to 'adapt' the int array to function as a list.
- We logically 'wrap' a type in another class, adapting it to be used by another library.
- Now we can do this..

```
int[] a = { 123, 321, 98, 23 };
Collections.sort(intArrayAsList(a));
```

```
1  // Concrete implementation built atop skeletal impl
2  static List<Integer> intArrayAsList(final int[] a) {
3
4   return new AbstractList<Integer>() {
5       public Integer get(int i) {
6           return a[i];
7       }
8
9       @Override
10      public Integer set(int i, Integer val) {
11          int oldVal = a[i];
12          a[i] = val;
13          return oldVal;
14      }
15
16      public int size() {
17          return a.length;
18      }
19    };
20
21 }
```

Facade Pattern

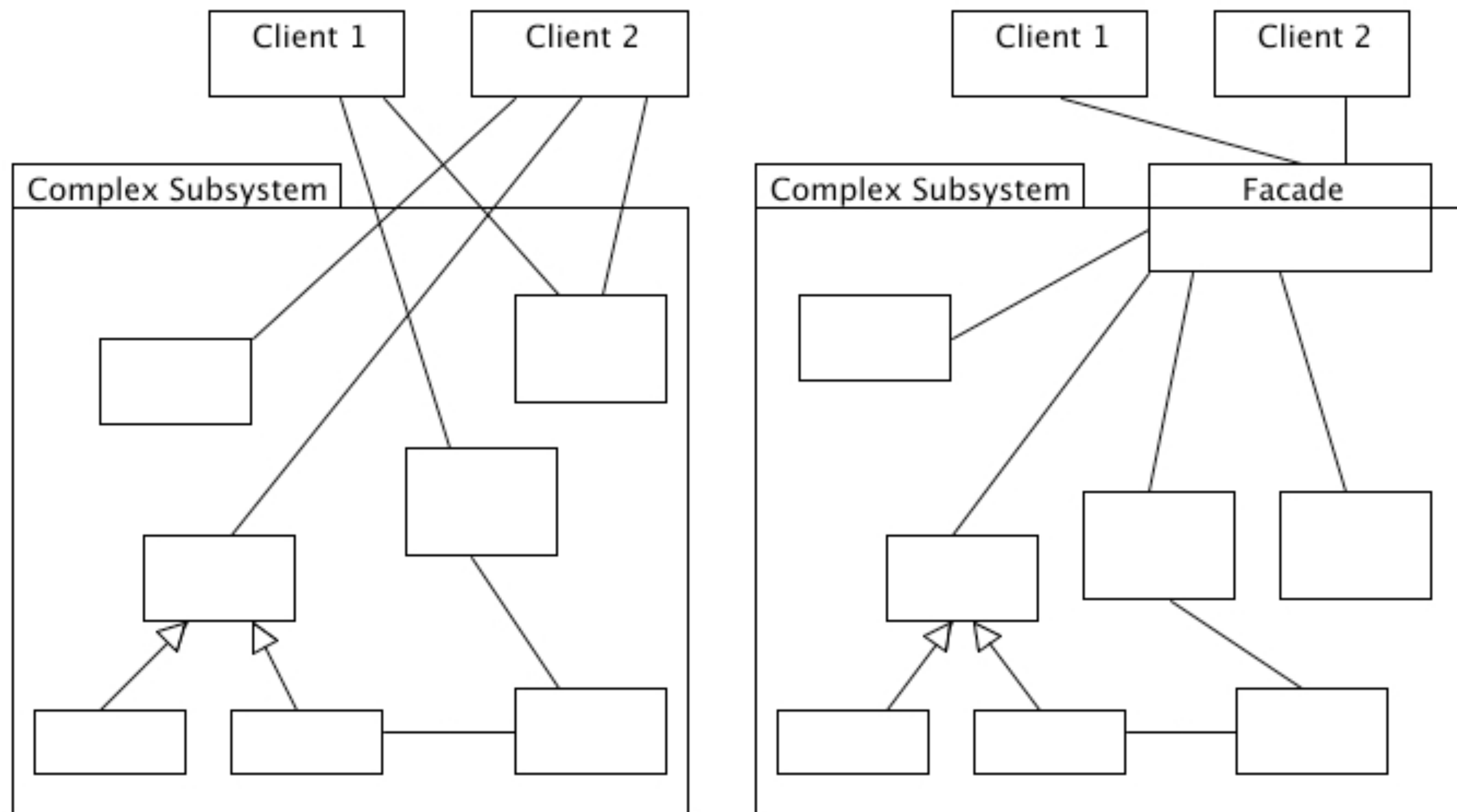
- Another pattern which uses ‘wrapping’, ’delegation’ and ‘composition’
- You should be noticing a pattern by now, decoupling and indirection creates flexibility and a few simple techniques can go a long way!
- The goal of a Facade is to decouple components of your system and provide a simplified interface to complex subsystems.

Facade Pattern

- Lets say that you need a database in an application.
- Further, lets say that a brand-new fancy, columnar, document-oriented, distributed *experimental* database is in *v1*
- Your boss says.. “Lets try this new database, but if it fails in production we need to be able to swap it for something else quickly”

Facade Pattern

- Given that your job is on the line, which one of these diagrams seems like a better design? (‘complex subsystem’ is your database)



Facade Pattern

- Use the facade pattern when:
    - you want to provide a simple interface to a complex subsystem.
    - you want to reduce coupling between clients-subsystems or subsystems-subsystems.
    - you want to layer your subsystems.

Facade Pattern

- For example, lets say we are modeling a computer.
- A computer consists of multiple complex collaborating components.
- A user of the computer should not be required to understand how all these parts work, it should *abstracted* away from them.

```
1  /* Complex parts */
2
3  class CPU {
4      public void freeze() { /* complex logic */ }
5      public void jump(long position) { /* complex logic */ }
6      public void execute() { /* complex logic */ }
7  }
8
9  class Memory {
10     public void load(long position, byte[] data) { /*..*/ }
11 }
12
13 class HardDrive {
14     public byte[] read(long lba, int size) {
15         /* complex logic */
16         return null;
17     }
18 }
```

Facade Pattern

- A facade wraps all the components of a computer system.
- It exposes a simplified interface for users.
- Note that this would allow us to upgrade the RAM and this would have no effect on users of our facade!

```
1   class ComputerFacade {
2       public static final int BOOT_ADDRESS = 0x13423;
3       public static final int BOOT_SECTOR = 0x423;
4       public static final int SECTOR_SIZE = 0x23;
5
6       private CPU processor;
7       private Memory ram;
8       private HardDrive hd;
9
10      public ComputerFacade() {
11          this.processor = new CPU();
12          this.ram = new Memory();
13          this.hd = new HardDrive();
14      }
15
16      public void start() {
17          processor.freeze();
18          ram.load(
19            BOOT_ADDRESS,
20            hd.read(BOOT_SECTOR, SECTOR_SIZE)
21          );
22          processor.jump(BOOT_ADDRESS);
23          processor.execute();
24      }
25  }
```

Facade Pattern

- Using the computer now because as easy as pressing the ‘start’ button.

```
class ComputerUser {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
}
```

- In our database example, you would have a DatabaseFacade that provided “create”, “read”, “update” and “delete” methods.
- If the db fails, the facade’s interface remains the same, but we rewrite the internals to handle a different database that functions.

Pattern Categories

- All of the patterns we have looked at thus far are knows as ‘Structural Patterns’
- Structural design patterns are design patterns that ease the design by identifying a simple, flexible way to realize relationships between entities.
- There are many others.. ‘Flyweight’, ‘Bridge’, ‘Composite’, ‘Proxy’, etc.
- Next we will look at some ‘Behavioral Patterns’.
- Behavior patterns identify common communication patterns between objects decouple these from the logic being executed.

Chain of Responsibility

- Remember this?
- A design pattern consisting of a source of ‘command’ objects and a series of ‘processing’ objects
- Each processing object contains logic that defines the types of command objects that it can handle;
- A processing ‘pipeline’, good for programs that pass the same data through a series of ‘phases’.

Chain of Responsibility

- Our example before was as follows:
  - A company needs to have any expenditure approved.
  - Depending on the price, the expenditure needs approval by different levels of management
  - Small priced expenditures require manager approval whereas large priced expenditures require director approval and so on.

Chain of Responsibility

- We have an abstract class representing the power to approve a expenditure up to a certain price.
- PurchasePower will be extended by Managers, Directors, etc.

```java
abstract class PurchasePower {
  protected static final double BASE = 10;
  protected PurchasePower successor;

  public void setSuccessor(PurchasePower successor) {
    this.successor = successor;
  }

  abstract public void processRequest(PurchaseRequest request);
}

class PurchaseRequest {
  private double amount;

  public PurchaseRequest(double amount, String purpose) {
      this.amount = amount;
  }

  public double getAmount() {
      return amount;
  }
  public void setAmount(double amt)  {
      amount = amt;
  }
}
```

Chain of Responsibility

- Note that PurchasePower has a successor.
- In other words, if a price is too high for this tier of management to approach, the request is delegated to the next tier in the chain.

```
1  abstract class PurchasePower {
2    protected static final double BASE = 10;
3    protected PurchasePower successor;
4
5    public void setSuccessor(PurchasePower successor) {
6      this.successor = successor;
7    }
8
9    abstract public void processRequest(PurchaseRequest request);
10 }
11
12 class PurchaseRequest {
13   private double amount;
14
15   public PurchaseRequest(double amount, String purpose) {
16       this.amount = amount;
17   }
18
19   public double getAmount() {
20       return amount;
21   }
22   public void setAmount(double amt)  {
23       amount = amt;
24   }
25 }
```

Chain of Responsibility

- PurchasePower subclasses will implement the processRequest method, which will check to see if the cost is low enough for approval by that tier of management.
- processRequest takes a PurchaseRequest, which represents the expenditure that needs to be approved.

```
abstract class PurchasePower {
  protected static final double BASE = 10;
  protected PurchasePower successor;

  public void setSuccessor(PurchasePower successor) {
    this.successor = successor;
  }

  abstract public void processRequest(PurchaseRequest request);
}

class PurchaseRequest {
  private double amount;

  public PurchaseRequest(double amount, String purpose) {
      this.amount = amount;
  }

  public double getAmount() {
      return amount;
  }
  public void setAmount(double amt)  {
      amount = amt;
  }
}
```

Chain of Responsibility

- There can N number of processing objects int the chain.
- DirectorPPower implements PurchasePower, implementing the logic appropriate for a director.
- PresidentPower will be the terminal processing object in the chain. Note that to does not pass on to a successor.

```
1  class DirectorPPower extends PurchasePower {
2    private final double ALLOWABLE = 20 * BASE;
3
4    public void processRequest(PurchaseRequest request) {
5      if (request.getAmount() < ALLOWABLE) {
6        System.out.println(
7          "Director will approve $" + request.getAmount());
8      } else if (successor != null) {
9        successor.processRequest(request);
10     }
11   }
12 }
13
14 class PresidentPPower extends PurchasePower {
15   private final double ALLOWABLE = 40 * BASE;
16
17   public void processRequest(PurchaseRequest request) {
18     if (request.getAmount() < ALLOWABLE) {
19       System.out.println(
20         "President will approve $" + request.getAmount());
21     } else {
22       System.out.println(
23         "Your request for $" + request.getAmount()
24           + " needs a board meeting!");
25     }
26   }
27 }
```

Chain of Responsibility

- Then we build up our chain and execute the pipeline with a PurchaseRequest

- When we first looked at this it may not have been apparent how applicable this was to your translator, but perhaps now it is.

```
1  public static void main(String[] args) {
2    ManagerPPower manager = new ManagerPPower();
3    DirectorPPower director = new DirectorPPower();
4    VicePresidentPPower vp = new VicePresidentPPower();
5    PresidentPPower president = new PresidentPPower();
6    manager.setSuccessor(director);
7    director.setSuccessor(vp);
8    vp.setSuccessor(president);
9
10   while (true) {
11       System.out.println("Enter the amount.");
12       double d = // take input
13       manager.processRequest(new PurchaseRequest(d));
14   }
15 }
16
```

Command Pattern

- The Command Pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.

- For example, lets say you have a series of tasks that you want to queue up and then do in sequence at a later time.

- Or perhaps you have a few different operations that can happen based some condition that you don't know yet.

Command Pattern

- Four entities associated with the command pattern: *command, receiver, invoker* and *client.*
  - A ***command*** object encapsulates the command to executed.
  - The ***receiver*** knows how to process the command.
  - An **invoker** object knows how to execute a command on the receiver.
    - The invoker does not know anything about a concrete command, it knows only the command interface.
  - The **client** decides when and which commands to execute.
    - Both an invoker object and command object(s) are held by a client.

Command Pattern

- Suppose we have some home security software that based on some set of criteria turns on and off different appliances throughout the house while the residents are on vacation.
- Using the command pattern we can build a set of commands that that will get triggered at some point throughout the day.
- We'll start with something that simply turns on and off the lights in the house.

```
1   // The Command interface
2   interface Command {
3     void execute();
4   }
5
6   // Concrete commands
7   class FlipUpCommand implements Command {
8     private Light l;
9
10    public FlipUpCommand(Light light) {
11      l = light;
12    }
13
14    @Override public void execute() { l.turnOn(); }
15  }
16
17  class FlipDownCommand implements Command {
18    private Light l;
19
20    public FlipDownCommand(Light light) {
21      l = light;
22    }
23
24    @Override public void execute() { l.turnOff(); }
25  }
```

Command Pattern

- At line 2 we have the Command interface. Very simple, just one method that performs whatever logic the implementation provides.
- At lines 7 and lines 17 we have two implementations of the Command interface.
- One to turn on the lights, one to turn off the lights of the house.
- Note that they are constructed with a Light object. (Another wrapper!)

```
1  // The Command interface
2  interface Command {
3    void execute();
4  }
5
6  // Concrete commands
7  class FlipUpCommand implements Command {
8    private Light l;
9
10   public FlipUpCommand(Light light) {
11     l = light;
12   }
13
14   @Override public void execute() { l.turnOn(); }
15 }
16
17 class FlipDownCommand implements Command {
18   private Light l;
19
20   public FlipDownCommand(Light light) {
21     l = light;
22   }
23
24   @Override public void execute() { l.turnOff(); }
25 }
```

Command Pattern

- Here we have our Receiver, the thing that does the actual work.

- Note that this is the class we were wrapping with our Command objects, nothing special about it!

```
1  // The Receiver class
2  // The thing that ultimately does the work.
3  class Light {
4    public void turnOn() {
5      System.out.println("The light is on");
6    }
7
8    public void turnOff() {
9      System.out.println("The light is off");
10   }
11 }
```

Command Pattern

- Next our Invoker, the thing that actually triggers the commands to execute.
- Note that knows nothing about the Light class. Its completely decoupled.
- This invoker doesn't care if it effectively is turning on lights, or the television or watering the lawn.

```
1  // The Invoker class
2  class Switch {
3    public void storeAndExecute(Command cmd) {
4      cmd.execute();
5    }
6  }
```

Command Pattern

- Finally our client.
- In this case our logic is very simple, but there is no reason could not be much more clever.
- For example, we could load all the commands in queue and run the Switch on a different thread that consumes asynchronously.

```java
// The client
public class PressSwitch {
  public static void main(String[] args) {
    String onOrOff = "ON";
    if(new Random().nextInt() % 2 == 0) {
      onOrOff = "OFF";
    }

    Light lamp = new Light();
    Command switchUp = new FlipUpCommand(lamp);
    Command switchDown = new FlipDownCommand(lamp);

    Switch s = new Switch();
    switch (onOrOff) {
      case "ON":
          s.storeAndExecute(switchUp);
          break;
      case "OFF":
          s.storeAndExecute(switchDown);
          break;
    }
  }
}
```

Strategy Pattern

- The strategy pattern enables an algorithm's behavior to be selected at runtime.
- The strategy pattern…
  - defines a family of algorithms
  - encapsulates each algorithm
  - makes the algorithms interchangeable within that family.

Strategy Pattern

- Let's say we are writing some software to manage a bar.
- Depending on whether or not its happy hour, we want to charge the customers differently.
- Our strategy for pricing drinks changes over time.

```
1  // The strategy interface
2  interface BillingStrategy {
3    public double getActPrice(double rawPrice);
4  }
5
6  // Normal billing strategy (unchanged price)
7  class NormalStrategy implements BillingStrategy {
8    @Override
9    public double getActPrice(double rawPrice) {
10     return rawPrice;
11   }
12 }
13
14 // Strategy for Happy hour (50% discount)
15 class HappyHourStrategy implements BillingStrategy {
16   @Override
17   public double getActPrice(double rawPrice) {
18     return rawPrice * 0.5;
19   }
20 }
21
```

Strategy Pattern

- On line 2 we have our strategy interface, with on method, modify the price for a drink based on some algorithm.
- On lines 7 and 15 we have our implementations, one for non-happy hour and one for happy hour.
- Note that we could have others.. for example a "RegularCustomerStrategy" or "LastCallStrategy"

```
1  // The strategy interface
2  interface BillingStrategy {
3    public double getActPrice(double rawPrice);
4  }
5
6  // Normal billing strategy (unchanged price)
7  class NormalStrategy implements BillingStrategy {
8    @Override
9    public double getActPrice(double rawPrice) {
10     return rawPrice;
11   }
12 }
13
14 // Strategy for Happy hour (50% discount)
15 class HappyHourStrategy implements BillingStrategy {
16   @Override
17   public double getActPrice(double rawPrice) {
18     return rawPrice * 0.5;
19   }
20 }
21
```

Strategy Pattern

- Here if our Tab class, which keeps track of the bill that a given customer might have.
- Note that its constructed with a BillingStrategy interface, but has a mutator method where the strategy can be altered (line 27)

```
1  // The customer who will have some number of drinks,
2  // priced depending on the strategy
3  class Tab {
4    private List<Double> drinks;
5    private BillingStrategy strategy;
6
7    public Tab(BillingStrategy strategy) {
8      this.drinks = new ArrayList<Double>();
9      this.strategy = strategy;
10   }
11
12   public void add(double price, int quantity) {
13     drinks.add(strategy.getActPrice(price * quantity));
14   }
15
16   // Payment of bill
17   public void printBill() {
18     double sum = 0;
19     for (Double i : drinks) {
20         sum += i;
21     }
22     System.out.println("Total due: " + sum);
23     drinks.clear();
24   }
25
26   // Set Strategy
27   public void setStrategy(BillingStrategy strategy) {
28     this.strategy = strategy;
29   }
30 }
```

Strategy Pattern

- On line 12, we have an add method which takes a price and a quantity, and adds an actual price to the drink array based on the strategy.

```
1  // The customer who will have some number of drinks,
2  // priced depending on the strategy
3  class Tab {
4    private List<Double> drinks;
5    private BillingStrategy strategy;
6
7    public Tab(BillingStrategy strategy) {
8      this.drinks = new ArrayList<Double>();
9      this.strategy = strategy;
10   }
11
12   public void add(double price, int quantity) {
13     drinks.add(strategy.getActPrice(price * quantity));
14   }
15
16   // Payment of bill
17   public void printBill() {
18     double sum = 0;
19     for (Double i : drinks) {
20         sum += i;
21     }
22     System.out.println("Total due: " + sum);
23     drinks.clear();
24   }
25
26   // Set Strategy
27   public void setStrategy(BillingStrategy strategy) {
28     this.strategy = strategy;
29   }
30 }
```

Strategy Pattern

- Finally our Bar program.
- Over time, we modify the strategy in use based on the changing condition.
- Moreover, we change the algorithm at runtime!

```
1  // Executes the serving of drinks
2  public class Bar {
3    public static void main(String[] args) {
4      Tab a = new Tab(new NormalStrategy());
5
6      // Normal billing
7      a.add(1.0, 1);
8
9      // Start Happy Hour
10     a.setStrategy(new HappyHourStrategy());
11     a.add(1.0, 2);
12
13     // New Customer
14     Tab b = new Tab(new HappyHourStrategy());
15     b.add(0.8, 1);
16     // The Customer pays
17     a.printBill();
18
19     // End Happy Hour
20     b.setStrategy(new NormalStrategy());
21     b.add(1.3, 2);
22     b.add(2.5, 1);
23     b.printBill();
24   }
25 }
```

Creational Patterns

- creational patterns are design patterns that deal with object creation mechanisms
- creational design patterns are composed of two dominant ideas:
  - encapsulating knowledge about which concrete classes the system uses.
  - hiding how instances of these concrete classes are created and combined.
- We'll look at just one such pattern. The Singleton.

Singleton Pattern

- In many cases, it is important for some classes having at most one instance.
- Ex. there might exist many printers in the system, but there should be only one printer spooler.
- But how can we ensure that there is only one instance and that this instance is accessible?

Singleton Pattern

- The class can ensure that only one instance is created by making constructors private and providing accessor methods to that one instance.
- This is called the Singleton pattern, it consists of just one class, all methods which provide a way to access the instance are usually static.
- Sounds easy enough.. right?

Singleton

- Here we have a singleton.
- We have one private static instance of the class that initialized when the JVM loads the class.
- This is known as '*eager initialization*'.

```
public class Singleton {
  private static final Singleton INSTANCE = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() {
      return INSTANCE;
  }
}
```

Singleton

- Note that the constructor is private.
- No other class in our system can instantiate a Singleton, so we can be confident that the single instance invariant is satisfied.

```
public class Singleton {
  private static final Singleton INSTANCE = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() {
      return INSTANCE;
  }
}
```

Singleton

- Finally, there is the getInstance method with returns the single instance referred to by our static member.
- Pretty straight-forward!
- However, what if our instance is very expensive to create, and we want to create it only on demand?

```
public class Singleton {
  private static final Singleton INSTANCE = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() {
      return INSTANCE;
  }
}
```

Singleton

- Here we have an alternate approach, exhibiting ‘lazy initialization’.
- The first time the getInstance method is called, we’ll construct our instance.

```java
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }

    return instance;
  }
}
```

Singleton

- Seems easy enough, except there is a subtle and evil bug here.
- Can you see it?

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }

    return instance;
  }
}
```

Singleton

- Seems easy enough, except there is a subtle and evil bug here.
- Can you see it?
- Have you done multi-threaded programming?

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }

    return instance;
  }
}
```

Singleton

- Ok no problem. Slap a synchronized keyword on the getInstance method.
- Fixed.

```
public class Singleton {
    private static Singleton instance;
    private Singleton() { }

    public static synchronized Singleton getInstance() {
      if (instance == null) {
          instance = new Singleton();
      }

      return instance;
    }
}
```

Singleton

- However, getInstance is called many times over the life of the application and every time a lock must be acquired.
- This is “a bad thing™”
- Synchronizing a method can in some extreme cases decrease performance by a factor of 100 or higher!

```
public class Singleton {
    private static Singleton instance;
    private Singleton() { }

    public static synchronized Singleton getInstance() {
      if (instance == null) {
          instance = new Singleton();
      }

      return instance;
    }
}
```

Singleton

- Well if we can move the synchronized from the method to a block inside the method, we can try to decrease the need for a lock.
- But… where do we put it?

```
public class Singleton {
    private static Singleton instance;
    private Singleton() { }

    public static synchronized Singleton getInstance() {
      //  synchronized (Singleton.class) { // this doesn't help
      if (instance == null) {
        //  synchronized (Singleton.class) { // race condition!
              instance = new Singleton();
        //  }
      }
      // }

      return instance;
    }
}
```

Singleton

- 'Double-checked locking' is the answer.
- Another design pattern!

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null ) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }

    return instance;
  }
}
```

Singleton

- 'Double-checked locking' is the answer.
- Another design pattern!

```java
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null ) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }

    return instance;
  }
}
```

Singleton

- ‘double-checked locking’ is used to reduce the overhead of acquiring a lock by first testing the locking criterion without actually acquiring the lock.

- The first call to getInstance() will create the object and only the few threads trying to access it during that time need to be synchronized; after that all calls just get a reference to the member variable.

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null ) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }

    return instance;
  }
}
```

Singleton

- Ok! now we are safe.. right?
- Actually no.
- The code generated by the compiler is allowed to update the shared variable to point to a partially constructed object
- i.e. before instance has finished performing the initialization!

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null ) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }

    return instance;
  }
}
```

Singleton

- We have to add the *volatile* keyword to our instance declaration.
- This will ensure that no partially constructed objects will be returned.
- Complicated!

```
public class Singleton {
  private volatile static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
    if (instance == null ) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }

    return instance;
  }
}
```

Singleton

- Fortunately some clever researchers have figured out another way to write a singleton that is lazy, thread-safe and simple.
- “initialization on demand holder idiom”

```
public class Singleton {
  private Singleton() { }

  private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Singleton

- This relies on the fact that inner classes are not loaded until they are referenced.
- This means that on the first call to getInstance, the class loader will load the SingleHolder class.
- On load the static INSTANCE member will be initialized.
- So we eagerly-initialize on-demand.

```
public class Singleton {
  private Singleton() { }

  private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Singleton Pattern

- Why some much focus on the Singleton Pattern?
- We'll its kind of fun, as its deceptively complex.
- But also its a very common interview question
- And its a pervasively applied pattern for better or worse.

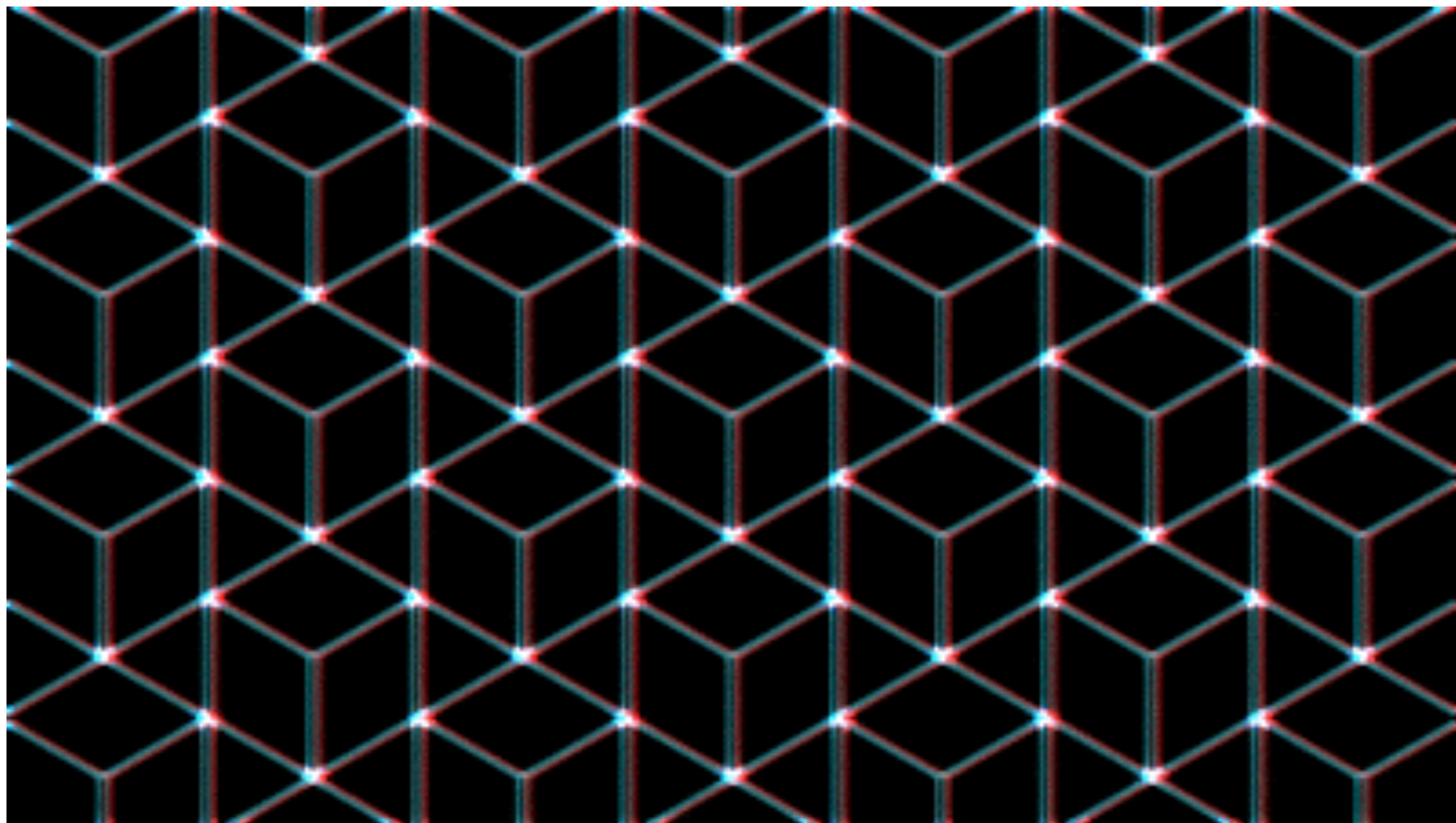
Singleton Pattern

- It has many detractors. Many people rail against it.
- Why?
  - It can lead to problems with garbage collection
  - Its often misapplied
  - It also can be used as a type of global state.

Design Pattern Code

As usual all the code is in Github:

https://github.com/nyu-oop/design-patterns



Project Option

- If you prefer, choose one of these design patterns and apply it in the context of your translator in a *substantive* way (check with me if you are unsure)
- Do this, rather than implementing memory management with the smart pointer.