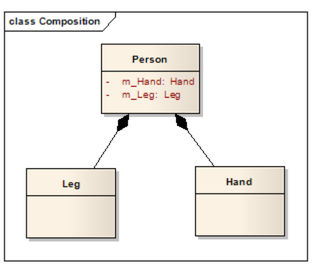
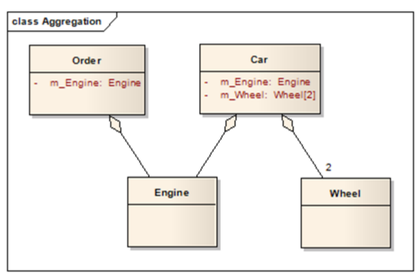
Prof. Jane Cleland-Huang
Two more patterns!!
sarec.nd.edu/courses/SE2017

Department of
Computer Science and
Engineering

UML Question



Composition implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.



Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

http://stackoverflow.com/questions/1644273/what-is-the-difference-between-aggregation-composition-and-dependency

http://aviadezra.blogspot.com/2009/05/uml-association-aggregation-composition.html

Chips Challenge

Concurrent Processes

- **Processes**: Self-contained execution environment. A process has its own memory space. Communication between processes typically uses Inter Process Communication (IPC) resources such as pipes and sockets.
- **Threads**: Lightweight processes – exist within a process. Share process's resources including memory and open files.
  A Java application starts with one *main thread* but we can create additional ones.

4

Java FX Thread

```
public static void main(String[] args) {
    new CatAndMouseGameFX();
    launch(args);
}

@Override
public void start(Stage stage) throws Exception {
    final Pane root = new AnchorPane();
    Scene scene = new Scene(root,800,800);
    stage.setScene(scene);
    stage.show();

    root.getChildren().add(mouse.getImageView());
    for(Cat cat: cats)
        root.getChildren().add(cat.getImageView());

    new AnimationTimer() {
        @Override
        public void handle(long now) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            mouse.mouseMove();
        }
    }.start();
}
```

Animation timer goes in the start method.

Only add sleep if you want to slow things down, but it can be a good idea.

Anything you want to happen in each loop goes here!

About using threads

- **<u>Downside</u>**: Can be harder to debug.
- **<u>Upside</u>**: Fun, interactive programs to write.
- **<u>Limitation</u>**: We will IGNORE synchronization problems. You will learn about this in your class on Distributed Systems.

6

Two Options

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

The thread class extends thread and implements run.

7

Design Pattern Overview

- **Creational Patterns:** Used to construct objects such that they can be decoupled from their implementing system.
- **Structural Patterns:** Used to form large object structures between many disparate objects.
- **Behavioral Patterns:** Used to manage algorithms, relationships, and responsibilities between objects.

Design Pattern Overview

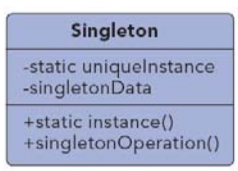
Object Scope: Deals with object relationships that can be changed at runtime.

Class Scope: Deals with class relationships that can be changed at compile time.

- C Abstract Factory
- S Adapter
- S Bridge
- C Builder
- B Chain of Responsibility
- B Command
- S Composite
- S Decorator
- S Facade
- C Factory Method
- S Flyweight
- B Interpreter
- B Iterator
- B Mediator
- B Memento
- C Prototype
- S Proxy
- B Observer
- C Singleton
- B State
- B Strategy
- B Template Method
- B Visitor

Singleton

Object Creational



Purpose

Ensures that only one instance of a class is allowed within a system.

Use When

- Exactly one instance of a class is required.
- Controlled access to a single object is necessary.

Example:

Most languages provide some sort of system or environment object that allows the language to interact with the native operating system. Since the application is physically running on only one operating system there is only ever a need for a single instance of this system object. The singleton pattern would be implemented by the language runtime to ensure that only a single copy of the system object is created and to ensure only appropriate processes are allowed access to it.

http://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

Singleton

Object Creational

```java
package controller.flightZone;

import controller.helper.Coordinates;

 * Establishes geographical zone for the simulation
public class ZoneBounds {
    long westLongitude = 0;
    long eastLongitude = 0;
    long northLatitude = 0;
    long southLatitude = 0;
    int maxAltitude = 0;

    private static ZoneBounds instance = null;
    protected ZoneBounds() {}

     * Return an instance of ZoneBounds
    public static ZoneBounds getInstance() {
        if(instance == null) {
            instance = new ZoneBounds();
        }
        return instance;
    }

     * setup the boundary of the Zone based on top left and bottom right coordinates
    public void setZoneBounds(long northLat, long westLon, long southLat, long eastL

     * Checks whether a coordinate is inside the zone
    public boolean inBounds(Coordinates coords) throws FlightZoneException{

     * Get westerly longitude degree
    public long getWestLongitude(){
```

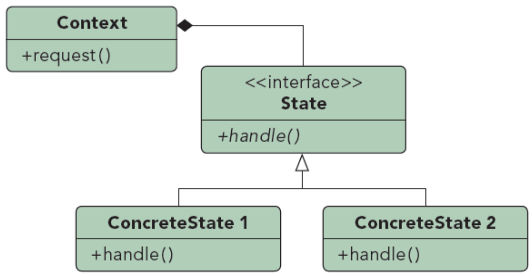
Singleton

Object Creational

```java
package controller.helper;
import java.awt.Point;

 * Given the window coordinates for the flight simulation, and the area of the map
public class DecimalDegreesToXYConverter {
    ZoneBounds zoneBounds;
    long xRange = 0;  // X coordinates in range of 0 to x
    long yRange = 0;  // Y coordinates in range of 0 to y
    double xScale = 0.0;  // The scale that transforms longitude to x coordinates
    double yScale = 0.0;  // The scale that transforms latitude to y coordinates
    long latitudeOffset=0;
    long longitudeOffset=0;
    int reservedLeftHandSpace = 0;

    long zoneXRange = 0l;
    long zoneYRange = 0l;

    private static DecimalDegreesToXYConverter instance = null;
    protected DecimalDegreesToXYConverter() {}

    /**
     * Return an instance of DecimalDegreesToXYConverter
     * @return
     */
    public static DecimalDegreesToXYConverter getInstance() {
        if(instance == null) {
            instance = new DecimalDegreesToXYConverter();
        }
        return instance;
    }

     * Setup
    public void setUp(long xSize, long ySize, int reservedLeftHandSpace){
```

State

Object Behavioral



Purpose

Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.

Use When

- The behavior of an object should be influenced by its state.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

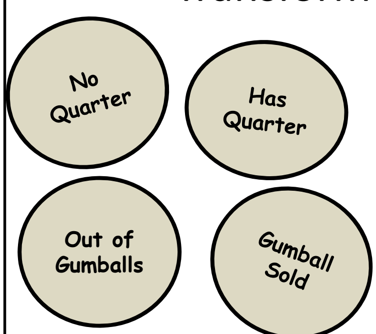
http://www.tutorialspoint.com/design_pattern/state_pattern.htm

Example:

An email object can have various states, all of which will change how the object handles different functions.

To avoid conditional statements in most or all methods there would be multiple state objects that handle the implementation with respect to their particular state. The calls within the Email object would then be delegated down to the appropriate state object for handling.

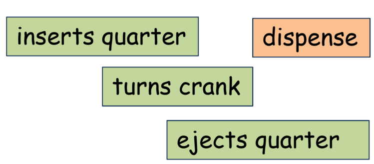
Transform State Machine to Code



Step 1: Identify states

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER=1;
final static int HAS_QUARTER=2;
final static int SOLD=3;

int state = SOLD_OUT;
```

Step 2: Create instance variables to hold current state and to define values for each state.



Step 3:
Identify actions

14

Create a class that acts as the state machine

```
public void insertQuarter() {
   if (state == HAS_QUARTER){
        System.out.println("You can't insert another quarter");
   } else if (state == SOLD_OUT) {
        System.out.println("Returning  your Quarter. The machine
                          is sold out");
   } else if (state == SOLD){
        System.out.println("Please wait for your gumball");
   } else if (state = NO_QUARTER){
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter.
                          Now turn the crank");
```

15

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

Here are the four states; they match the states in Mighty Gumball's state diagram.

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD_OUT state.

Now we start implementing the actions as methods....

When a quarter is inserted, if....

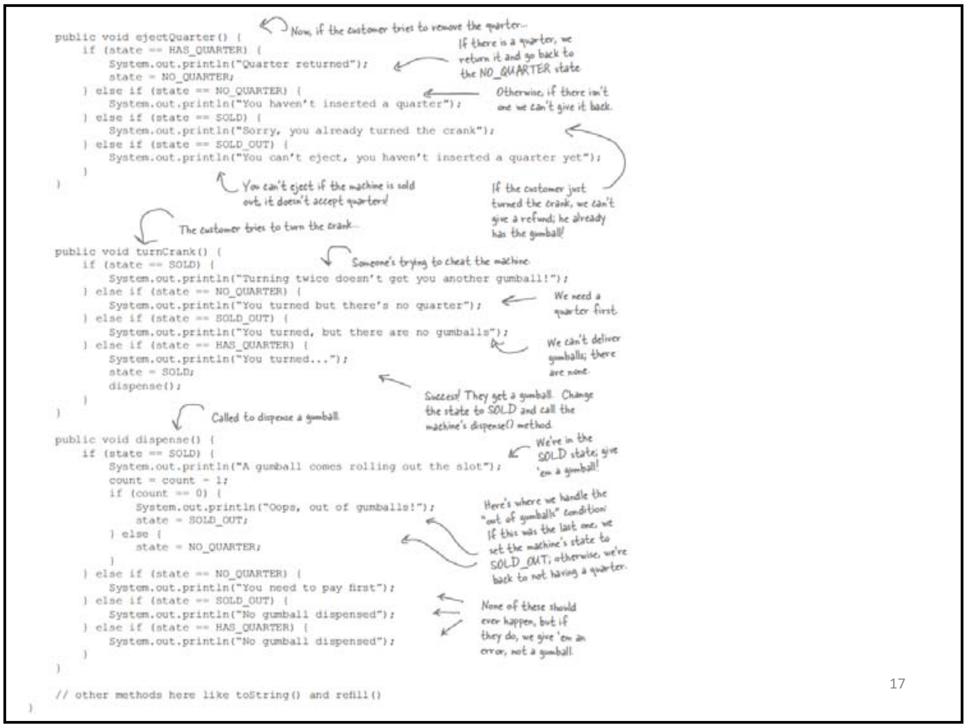
a quarter is already inserted we tell the customer;

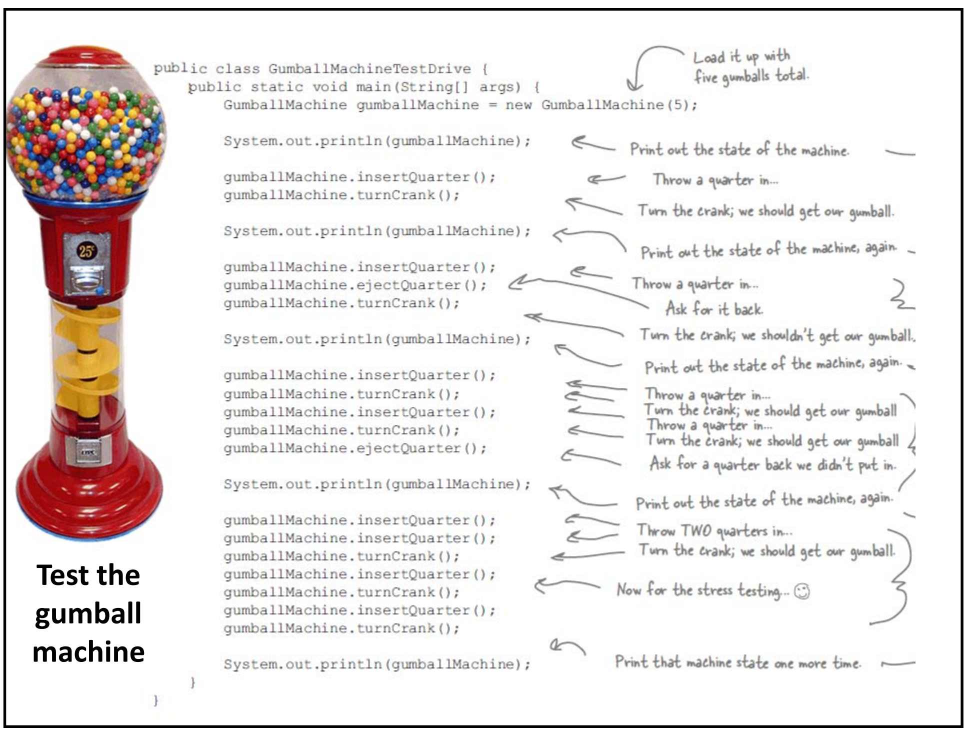
otherwise we accept the quarter and transition to the HAS_QUARTER state.

and if the machine is sold out, we reject the quarter.

If the customer just bought a gumball he needs to wait until the transaction is complete before inserting another quarter.

16


public void ejectQuarter() {
if (state == HAS_QUARTER) {
System.out.println("Quarter returned");
state = NO_QUARTER;
} else if (state == NO_QUARTER) {
System.out.println("You haven't inserted a quarter");
} else if (state == SOLD) {
System.out.println("Sorry, you already turned the crank");
} else if (state == SOLD_OUT) {
System.out.println("You can't eject, you haven't inserted a quarter yet");
}
}
Now, if the customer tries to remove the quarter...
If there is a quarter, we return it and go back to the NO_QUARTER state
Otherwise, if there isn't one we can't give it back.
You can't eject if the machine is sold out, it doesn't accept quarters!
If the customer just turned the crank, we can't give a refund; he already has the gumball!
The customer tries to turn the crank...
public void turnCrank() {
if (state == SOLD) {
System.out.println("Turning twice doesn't get you another gumball!");
} else if (state == NO_QUARTER) {
System.out.println("You turned but there's no quarter");
} else if (state == SOLD_OUT) {
System.out.println("You turned, but there are no gumballs");
} else if (state == HAS_QUARTER) {
System.out.println("You turned...");
state = SOLD;
dispense();
}
}
Someone's trying to cheat the machine.
We need a quarter first.
We can't deliver gumballs; there are none.
Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.
Called to dispense a gumball.
public void dispense() {
if (state == SOLD) {
System.out.println("A gumball comes rolling out the slot");
count = count - 1;
if (count == 0) {
System.out.println("Oops, out of gumballs!");
state = SOLD_OUT;
} else {
state = NO_QUARTER;
}
} else if (state == NO_QUARTER) {
System.out.println("You need to pay first");
} else if (state == SOLD_OUT) {
System.out.println("No gumball dispensed");
} else if (state == HAS_QUARTER) {
System.out.println("No gumball dispensed");
}
}
We're in the SOLD state; give 'em a gumball!
Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.
None of these should ever happen, but if they do, we give 'em an error, not a gumball.
// other methods here like toString() and refill()
}
17

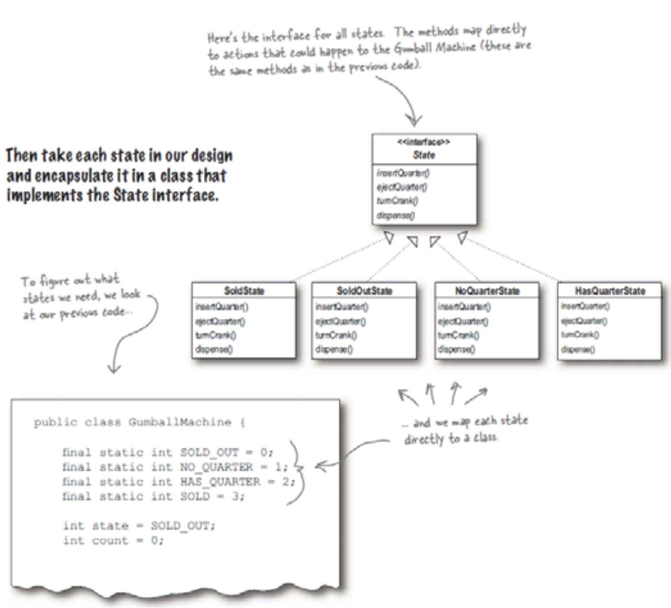

Test the gumball machine
public class GumballMachineTestDrive {
public static void main(String[] args) {
GumballMachine gumballMachine = new GumballMachine(5);
System.out.println(gumballMachine);
gumballMachine.insertQuarter();
gumballMachine.turnCrank();
System.out.println(gumballMachine);
gumballMachine.insertQuarter();
gumballMachine.ejectQuarter();
gumballMachine.turnCrank();
System.out.println(gumballMachine);
gumballMachine.insertQuarter();
gumballMachine.turnCrank();
gumballMachine.insertQuarter();
gumballMachine.turnCrank();
gumballMachine.ejectQuarter();
System.out.println(gumballMachine);
gumballMachine.insertQuarter();
gumballMachine.insertQuarter();
gumballMachine.turnCrank();
gumballMachine.insertQuarter();
gumballMachine.turnCrank();
gumballMachine.insertQuarter();
gumballMachine.turnCrank();
System.out.println(gumballMachine);
}
}
Load it up with five gumballs total.
Print out the state of the machine.
Throw a quarter in...
Turn the crank; we should get our gumball.
Print out the state of the machine, again.
Throw a quarter in...
Ask for it back.
Turn the crank; we shouldn't get our gumball.
Print out the state of the machine, again.
Throw a quarter in...
Turn the crank; we should get our gumball
Throw a quarter in...
Turn the crank; we should get our gumball
Ask for a quarter back we didn't put in.
Print out the state of the machine, again.
Throw TWO quarters in...
Turn the crank; we should get our gumball.
Now for the stress testing... ☺
Print that machine state one more time.

Solution

1. **First, we're going to define a State interface that contains a method for every action in the Gumball Machine.**
2. **Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.**
3. **Finally, we're going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.**

19

First let's create an interface for State, which all our states implement:



20

First we need to implement the State interface.

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And, you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment.

21

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

Old code

In the GumballMachine, we update the code to use the new classes rather than the static integers. The code is quite similar, except that in one class we have integers and in the other objects...

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;
```

New code

All the State objects are created and assigned in the constructor.

This now holds a State object, not an integer.

22

```
public class Gumball Machine{
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumBalls){
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumBalls;
        if(numberGumBalls>0){
                state = newQuarterState;
        }
    }
```

Any errors?

23

```
  public void insertQuarter(){
    state.insertQuarter();
  }
  public void ejectQuarter(){
    state.ejectQuarter();
  }
  public void turnCrank(){
    state.turnCrank();
    state.dispense();
  }
  void setState(State state){
    this.state = state;
  }
  void releaseBall(){
    System.out.println("Here is your gumball");
    if(count !=0){
        count = count – 1;
    }
  }
}
```

24

```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.

An inappropriate action for this state.

Return the customer's quarter and transition back to the NoQuarterState.

When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

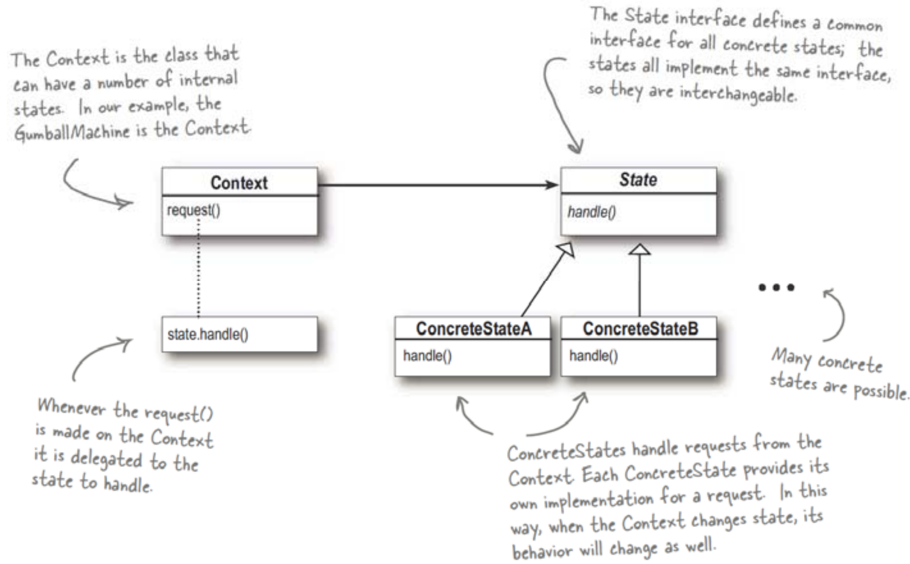
Another inappropriate action for this state.

What have we done?

1. Localized the behavior of each state into its own class.
2. Removed all of the problematic "if" statements that would have been difficult to maintain.
3. Closed each state for modification – while leaving the the Gumball Machine open to extension by adding new state classes.
4. Created a code base and class structure that matches the Gumball Machine state diagram.

26

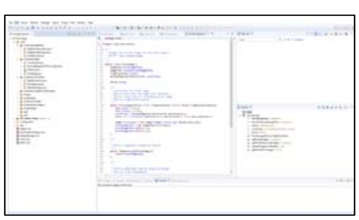
State Design Pattern



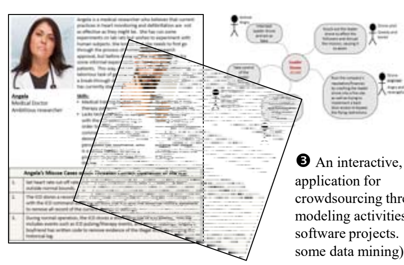
27

Team Projects

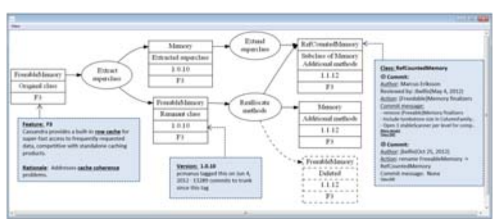
❶ An eclipse-based tool for supporting the creation of Safety Assurance Cases.



❷ An eclipse plugin for supporting impact analysis in Safety Critical Systems. Includes visualization.



❸ An interactive, GUI based application for crowdsourcing threat modeling activities for software projects. (Requires some data mining)

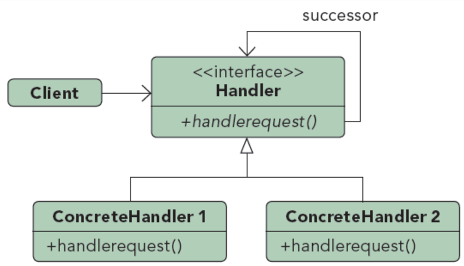


❹ A utility for interactively visualizing the evolution of requirements and source code. Sits on top of a Github repository

3-4 people per team.
Max three teams per project.

Chain of Responsibility

Object Behavioral



Purpose

Gives more than one object an opportunity to handle a request by linking receiving objects together.

Use When

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

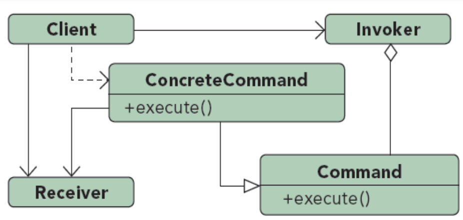
Example:

Exception handling. When an exception is thrown in a method the runtime checks to see if the method has a mechanism to handle the exception or if it should be passed up the call stack.

When passed up the call stack the process repeats until code to handle the exception is encountered or until there are no more parent objects to hand the request to.

Command

Object Behavioral



Purpose

Encapsulates a request allowing it to be treated as an object. This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

Use When

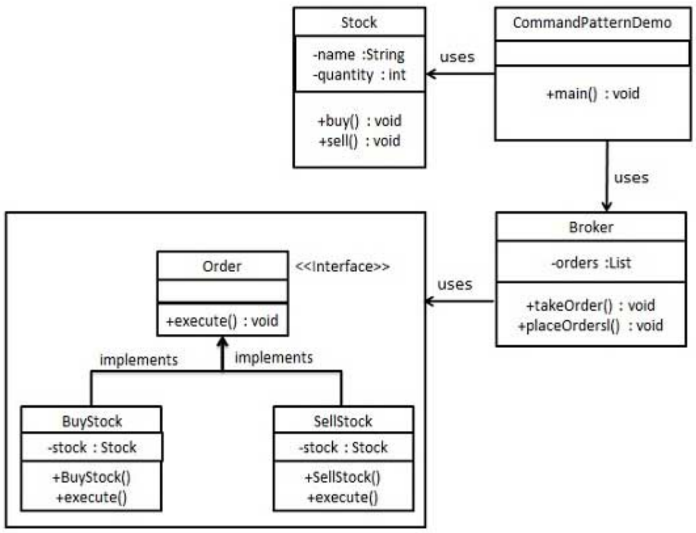
- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

Example:

Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is enqueued implements its particular algorithm within the confines of the interface the queue is expecting.

http://www.tutorialspoint.com/design_pattern/command_pattern.htm

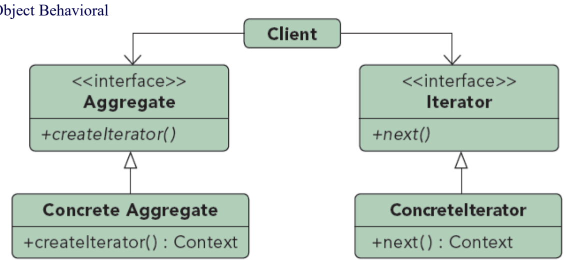
Command



http://www.tutorialspoint.com/design_pattern/command_pattern.htm

Iterator

Object Behavioral



Purpose

Allows for access to the elements of an aggregate object without allowing access to its underlying representation.

Use When

- Access to elements is needed without access to the entire representation.
- Multiple or concurrent traversals of the elements are needed.
- A uniform interface for traversal is needed.
- Subtle differences exist between the implementation details of various iterators.

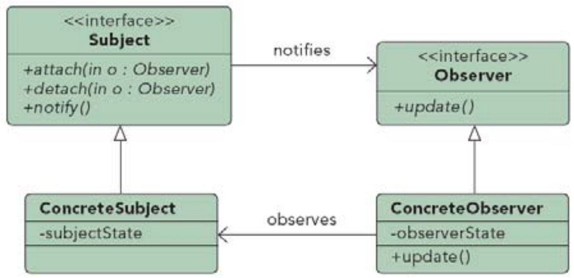
http://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

Example:

The Java implementation of the iterator pattern allows users to traverse various types of data sets without worrying about the underlying implementation of the collection. Since clients simply interact with the iterator interface, collections are left to define the appropriate iterator for themselves. Some will allow full access to the underlying data set while others may restrict certain functionalities, such as removing items.

Observer

Object Behavioral



Purpose

Lets one or more objects be notified of state changes in other objects within the system.

Use When

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.

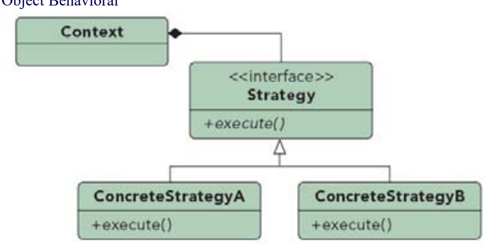
http://www.tutorialspoint.com/design_pattern/observer_pattern.htm

Example:

This pattern can be found in almost every GUI environment. When buttons, text, and other fields are placed in applications the application typically registers as a listener for those controls. When a user triggers an event, such as clicking a button, the control iterates through its registered observers and sends a notification to each.

Strategy

Object Behavioral



Purpose

Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

Use When

- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

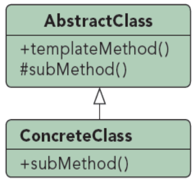
http://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

Example:

When importing data into a new system different validation algorithms may be run based on the data set. By configuring the import to utilize strategies the conditional logic to determine what validation set to run can be removed and the import can be decoupled from the actual validation code. This will allow us to dynamically call one or more strategies during the import.

Template

Class Behavioral



Purpose

Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.

Use When

- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most or all subclasses need to implement the behavior.

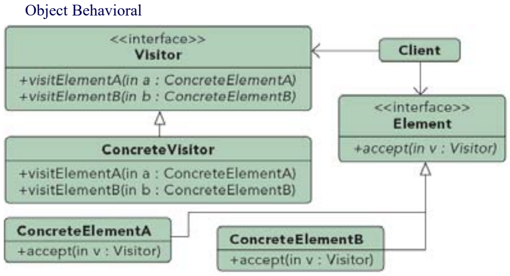
http://www.tutorialspoint.com/design_pattern/template_pattern.htm

Example:

A parent class, InstantMessage, will likely have all the methods required to handle sending a message. However, the actual serialization of the data to send may vary depending on the implementation. A video message and a plain text message will require different algorithms in order to serialize the data correctly. Subclasses of InstantMessage can provide their own implementation of the serialization method, allowing the parent class to work with them without understanding their implementation details.

Visitor

Object Behavioral



Purpose

Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.

Use When

- An object structure must have many unrelated operations performed upon it.
- The object structure can't change but operations performed on it can.
- Operations must be performed on the concrete classes of an object structure.
- Exposing internal state or operations of the object structure is acceptable.
- Operations should be able to operate on multiple object structures that implement the same interface sets.

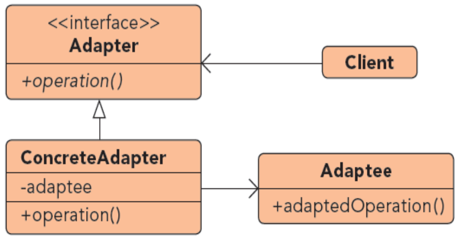
Example:

Calculating taxes in different regions on sets of invoices would require many different variations of calculation logic. Implementing a visitor allows the logic to be decoupled from the invoices and line items. This allows the hierarchy of items to be visited by calculation code that can then apply the proper rates for the region. Changing regions is as simple as substituting a different visitor.

http://www.tutorialspoint.com/design_pattern/visitor_pattern.htm

Adapter

Class and Object Structural



Purpose

Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.

Use When

- A class to be used doesn't meet interface requirements.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

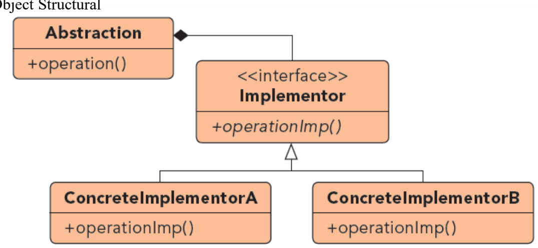
http://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

Example:

A billing application needs to interface with an HR application in order to exchange employee data, however each has its own interface and implementation for the Employee object. In addition, the SSN is stored in different formats by each system. By creating an adapter we can create a common interface between the two applications that allows them to communicate using their native objects and is able to transform the SSN format in the process.

Bridge

Object Structural



Purpose

Defines an abstract object structure independently of the implementation object structure in order to limit coupling.

Use When

- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

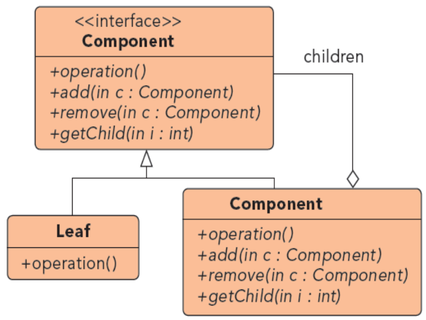
http://www.tutorialspoint.com/design_pattern/bridge_pattern.htm

Example:

The Java Virtual Machine (JVM) has its own native set of functions that abstract the use of windowing, system logging, and byte code execution but the actual implementation of these functions is delegated to the operating system the JVM is running on. When an application instructs the JVM to render a window it delegates the rendering call to the concrete implementation of the JVM that knows how to communicate with the operating system in order to render the window.

Composite

Object Structural



Purpose

Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

Use When

- Hierarchical representations of objects are needed..
- Objects and compositions of objects should be treated uniformly.

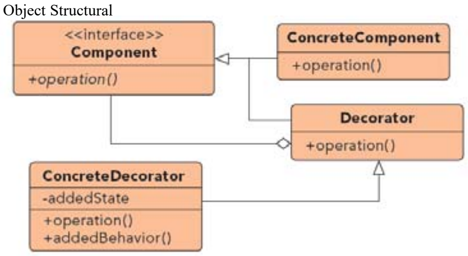
http://www.tutorialspoint.com/design_pattern/composite_pattern.htm

Example:

Sometimes the information displayed in a shopping cart is the product of a single item while other times it is an aggregation of multiple items. By implementing items as composites we can treat the aggregates and the items in the same way, allowing us to simply iterate over the tree and invoke functionality on each item. By calling the getCost() method on any given node we would get the cost of that item plus the cost of all child items, allowing items to be uniformly treated whether they were single items or groups of items.

Decorator

Object Structural



Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

Use When

- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Subclassing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.

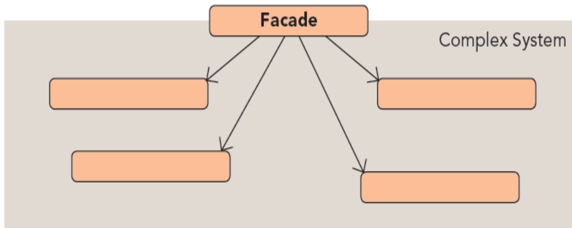
http://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Example:

Many businesses set up their mail systems to take advantage of decorators. When messages are sent from someone in the company to an external address the mail server decorates the original message with copyright and confidentiality information. As long as the message remains internal the information is not attached. This decoration allows the message itself to remain unchanged until a runtime decision is made to wrap the message with additional information.

Facade

Object Structural



Purpose

Supplies a single interface to a set of interfaces within a system.

Use When

- A simple interface is needed to provide access to a complex system.
- There are many dependencies between system implementations and clients.
- Systems and subsystems should be layered.

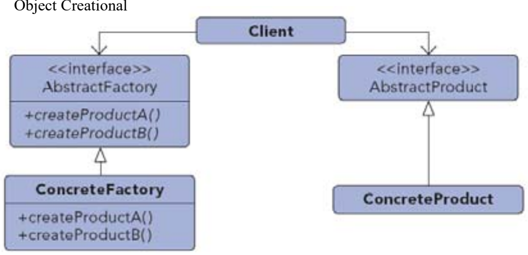
http://www.tutorialspoint.com/design_pattern/facade_pattern.htm

Example:

By exposing a set of functionalities through a web service the client code needs to only worry about the simple interface being exposed to them and not the complex relationships that may or may not exist behind the web service layer. A single web service call to update a system with new data may actually involve communication with a number of databases and systems, however this detail is hidden due to the implementation of the façade pattern.

Abstract Factory

Object Creational



Purpose

Provide an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.

Use When

- The creation of objects should be independent of the system utilizing them.
- Systems should be capable of using multiple families of objects.
- Families of objects must be used together.
- Libraries must be published without exposing implementation details.
- Concrete classes should be decoupled from clients.

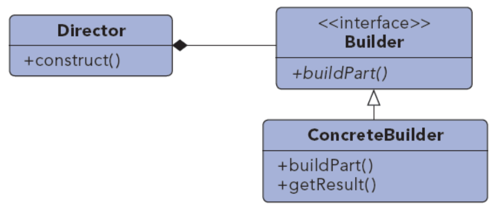
http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm

Example:

Email editors support multiple formats including plain text, rich text, and HTML. If the message is plain text then there could be a body object that represented just plain text and an attachment object that simply encrypted the attachment into Base64. If the message is HTML then the body object would represent HTML encoded text and the attachment object would allow for inline representation and a standard attachment. By utilizing an abstract factory for creation we can ensure that the appropriate object sets are created based upon the style of email that is being sent.

Builder

Object Creational



Purpose

Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

Use When

- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

Example:

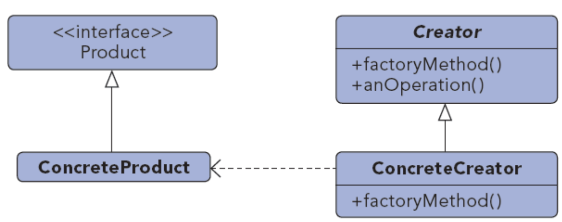
A file transfer application could possibly use many different protocols to send files and the actual transfer object that will be created will be directly dependent on the chosen protocol. Using a builder we can determine the right builder to use to instantiate the right object. If the setting is FTP then the FTP builder would be used when creating the object.

Builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. A Builder class builds the final object step by step.

http://www.tutorialspoint.com/design_pattern/builder_pattern.htm

Factory Method

Object Creational



Purpose

Exposes a method for creating objects, allowing subclasses to control the actual creation process.

Use When

- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

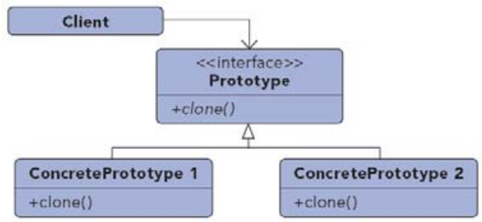
Example:

Many applications have some form of user and group structure for security. When the application needs to create a user it will typically delegate the creation of the user to multiple user implementations. The parent user object will handle most operations for each user but the subclasses will define the factory method that handles the distinctions in the creation of each type of user.

http://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Prototype

Object Creational



Purpose

Create objects based upon a template of an existing objects through cloning.

Use When

- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

http://www.tutorialspoint.com/design_pattern/prototype_pattern.htm

Example:

Rates processing engines often require the lookup of many different configuration values, making the initialization of the engine a relatively expensive process. When multiple instances of the engine is needed, say for importing data in a multi-threaded manner, the expense of initializing many engines is high. By utilizing the prototype pattern we can ensure that only a single copy of the engine has to be initialized then simply clone the engine to create a duplicate of the already initialized object.

Competition Time

http://www.vincehuston.org/dp/patterns_quiz.html