Summary: Software Construction

1 - Flow Abstraction

- In java one file is one class convention

2 - Encapsulation

In software design we encapsulate both data and computation to limit the **number of contact points** between different parts of the code.

Advantages:

- it makes it easier to understand a piece of code in isolation
- it makes the use of the isolated part by the rest of the code less error-prone
- it makes it easier to change one part of the code without breaking anything else

Inappropriate Intimacy: Having a class that is mostly accessed through getters and setters

Immutability Because String objects are immutable they can be shared.

- Use classes to define how domain concepts are represented in code, as opposed to encoding instances of these concepts as values of primitive types.
- Use enumerated types to represent a value in a collection of a small number of elements that can be enumerated.
- Hide the internal implementation of an abstraction behind an interface that tightly controls how an abstraction can be used
    - Declare fields of a class private, unless you have a strong reason not to
    - Declare any method private if it should not be explicitly part of the type's interface
- Ensure that the design of your classes prevents any code from modifying the data stored in an object of the class without using a method of the class.
- To provide information about the internal data in an object without violating encapsulation, two strategies include extending the interface of the class or re-turning copies of internal objects.
- Object diagrams can help explain or clarify the structure of complex object graphs, or how references are shared.
- Make classes immutable if possible. In Java, it is only possible to ensure that a class is immutable through careful design and inspection.

3 - Interfaces

An interface to a class consists of the methods of that class that are accessible to another class.

Methods that do not have an implementation are called **abstract methods**.

```java
public interface CardSource {
    Card draw();
    boolean isEmpty();
}
```

In Java, interfaces provide a specification of the methods that it should be possible to invoke on the objects of a class

```
public class Deck implements CardSource {...}
```

Effects:

- Formal guarantee that instances of the class types will have concrete implementations for all the methods in the interface types
- Subtype relationship between the implementing class and the interface type

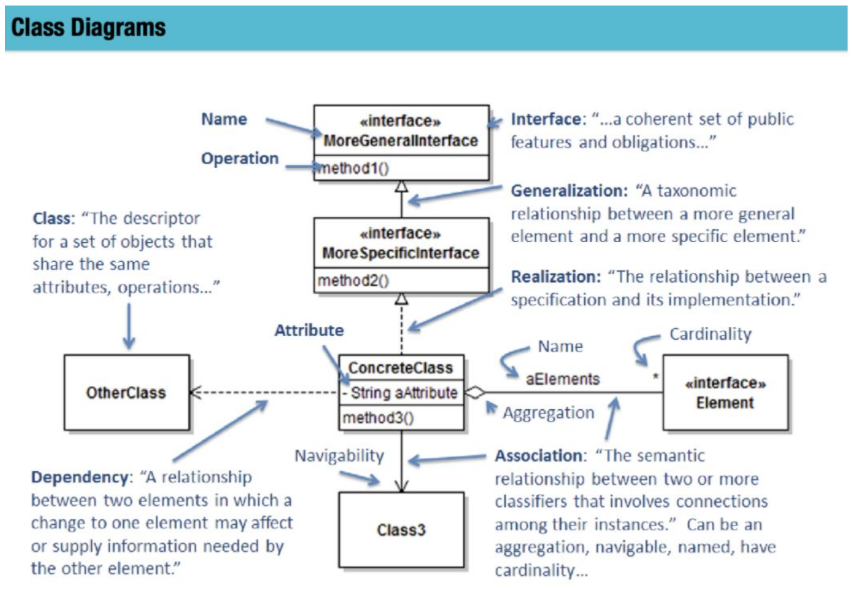
Comparable Interface Example

The library does **not** know how want to sort our cards. It can not anticipate all the possible objects a programmer may create.

The **Comparable<T>** interface helps to solve this problem.

It defines a method on how to compare objects

Class Diagrams



Class Diagrams
Name
Operation
«interface» MoreGeneralInterface
method1()
Interface: "...a coherent set of public features and obligations..."
Generalization: "A taxonomic relationship between a more general element and a more specific element."
Class: "The descriptor for a set of objects that share the same attributes, operations..."
«interface» MoreSpecificInterface
method2()
Realization: "The relationship between a specification and its implementation."
Attribute
Name
Cardinality
OtherClass
ConcreteClass
- String aAttribute
method3()
aElements
*
«interface» Element
Aggregation
Navigability
Association: "The semantic relationship between two or more classifiers that involves connections among their instances." Can be an aggregation, navigable, named, have cardinality...
Dependency: "A relationship between two elements in which a change to one element may affect or supply information needed by the other element."
Class3

Pros:

- Useful to represent how types are defined and related
- They are useful to capture the essence of one or more design decisions

Cons:

- Poor vehicle for capturing any runtime property of the code

Function Objects

Function objects are objects that provide the implementation for a single method.

Their interface typically maps one-to-one to that of an interface.

Normally, a Java interface represents only a subset of the complete interface of classes that implement it.

However, sometimes it is convenient to define classes that implement only the behaviour required by a small interface with only one method

In our card game, we want to implement different sorting strategies

- Modifying the code of compareTo (e.g., by using a flag to switch between different strategies) is deprecated.
- A better solution is to move the comparison code to a separate object

The solution is supported by the use of the **Comparator<T>** interface, which implements the method compare (as opposed to compareTo)

```
int compare (T pObject1, T pObject2)
```

```
public class RankFirstComparator implements Comparator<Card> {
    public int compare (Card pCard1, Card pCard2){
        /* Comparison code*/
    }
}
```

Problem:

- In some cases the information available to the class implementing the Comparator interface is enough (e.g., using the getter methods).
- Sometimes, however, implementing the compare method will require access to private members.

Three possible solutions:

1. Using a **nested class**
2. Through an **anonymous class**
3. Recurring to **lambda expressions**

Nested Class

```
public class Card
{
    static class CompareBySuitFirst implements Comparator<Card>
    {
        public int compare (Card pCard1, Card pCard2){
            /* Comparison code */
        }
    }
}
```

```
Collections.sort(aCards, new Card.CompareBySuitFirst())
```

Anonymous Class

```
public class Deck
{
    public void sort()
    {
        Collections.sort(aCards, new Comparator<Card>()){
            public int compare (Card pCard1, Card pCard2){
                /* Comparison code */
            }
        });
    }
}
```

Lambda Expressions

```
public class Deck
{
    public void sort(){
        Collections.sort(aCards, (card1, card2)=>
        card1.getRank().compareTo(card2.getRank()));
    }
}
```

Hiding internal structure

Iterator

```
public Iterator<Card> getCards(){...}
```

```
Iterator<Card> iterator = deck.getCards();
while(iterator.hasNext()){
    System.out.println(interator.next());
}
```

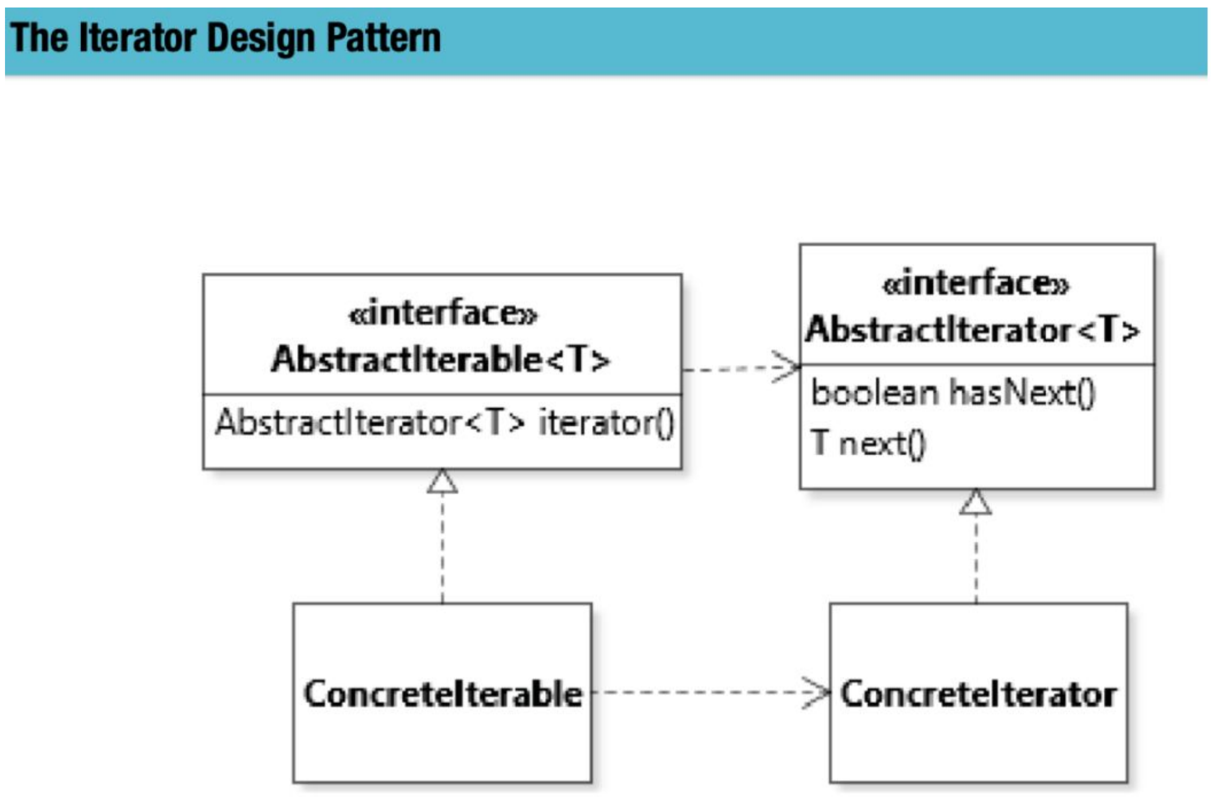
Iterable

```
public class Deck implements Iterable<Card>{
    private List<Card> aCards;
    public Iterator<Card> iterator() {
        return aCards.iterator();
    }
}
```

Iterating over a Deck

```
for(card card : deck){
    System.out.println(card);
}
```

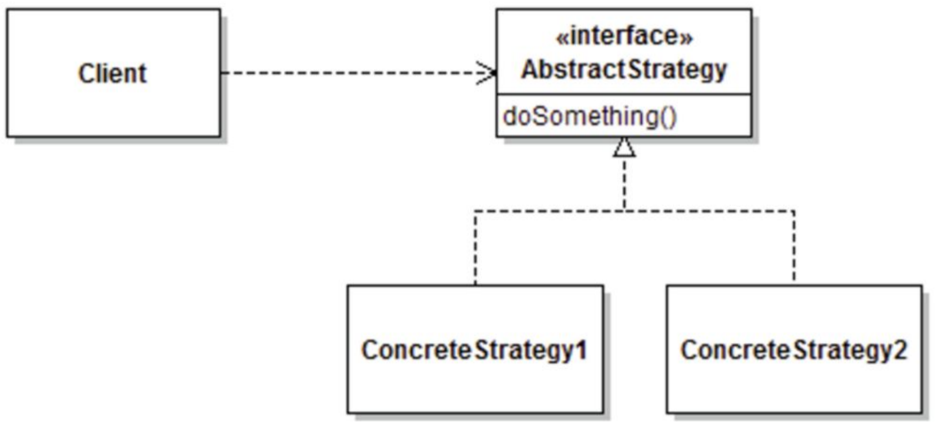
Iterator Design Pattern


The Iterator Design Pattern

Iterator
It provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Strategy Design Pattern

The Strategy Design Pattern



STRATEGY

It defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

- Does the AbstractStrategy need one or multiple methods to define the algorithm?
- Should the strategy method return anything or have a side-effect on the argument?
- Does a strategy need to store data?
- What should be the type of the return value and/or method parameters, as applicable? Ideally we want to choose these types to minimize coupling between a strategy and its clients.

Interface Segregation Principle

Interface Segregation Principle (ISP)

Client code should not be forced to depend on interfaces it does not need.

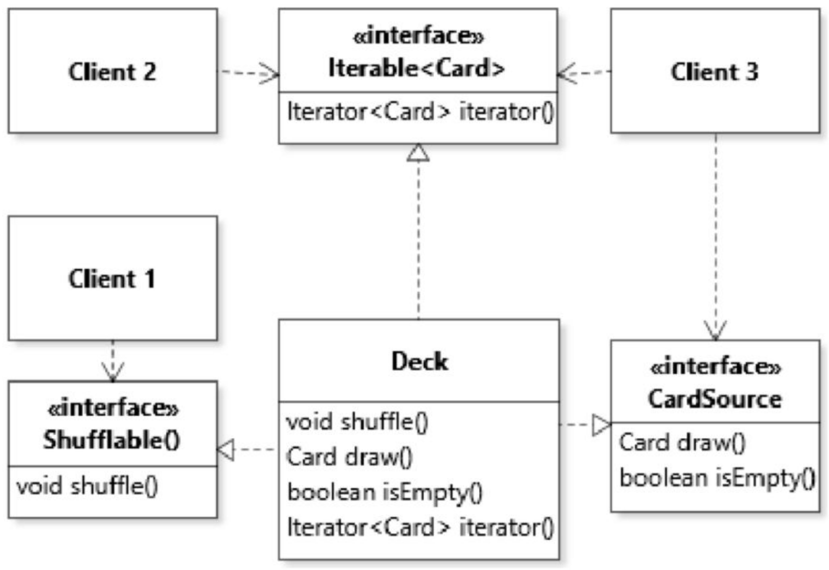
```java
public static List<Card> drawCards(Deck pDeck, int pNumber) {
    List<Card> result = new ArrayList<>();
    for( int i = 0; i < pNumber && !pDeck.isEmpty(); i++ ){
        result.add(pDeck.draw());
    }
    return result;
}
```

```java
public interface IDeck {
    void shuffle();
    Card draw();
    boolean isEmpty();
}
```

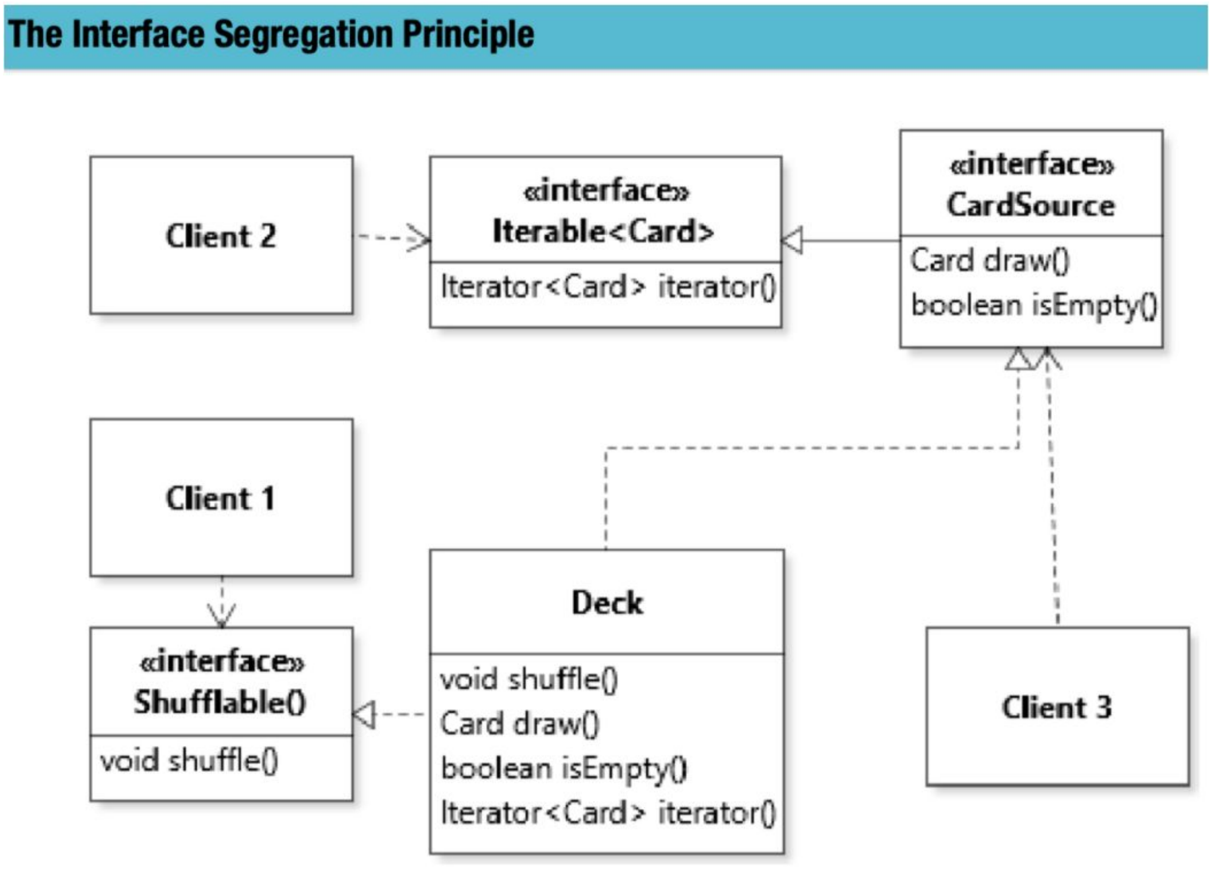
```java
public class Deck implements IDeck...
```

```java
public static List<Card> drawCards(IDeck pDeck, int pNumber)
```

The Interface Segregation Principle



```java
 public void displayCards(CardSource pSource) {
    if( !pSource.isEmpty() ) {
        pSource.draw();
        for( Card card : (Iterable<Card>) pSource) { ... }
    }
}
```



Questions & Summary

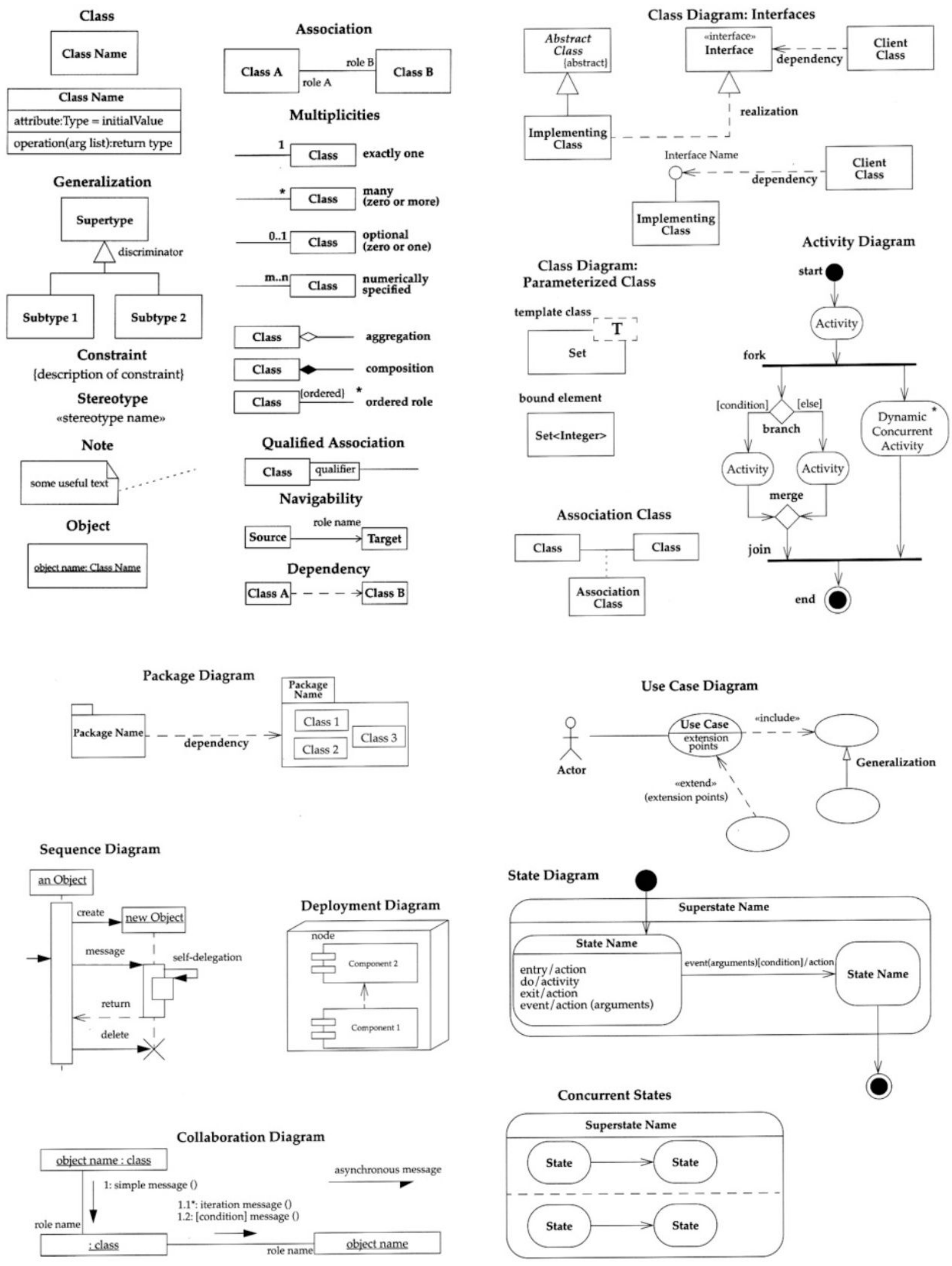
This lecture focused on how to use interfaces and polymorphism to achieve extensibility and reuse.

- Use interface types to decouple a specification from its implementation if you plan to have different implementations of that specification as part of your design;
- Define interfaces so that each interface groups a cohesive set of methods that are likely to be used together;
- Organize interfaces as subtypes of each other to create flexible groupings of behavior;
- Use library interfaces, such as Comparable, to implement commonly expected behavior;
- Use class diagrams to explore or capture important design decisions that have to do with how classes relate to each other;
- Consider function objects as a potential way to implement a small piece of required functionality, such as a comparison algorithm. Function objects can often be specified as instances of anonymous classes, or as lambda expressions;

- Use iterators to expose a collection of objects encapsulated within another with- out violating the encapsulation and information hiding properties of this object. This idea is known as the ITERATOR design pattern;
- Consider using the STRATEGY pattern if part of your design requires supporting an interchangeable family of algorithms;
- Make sure that your code does not depend on interfaces it does not need: break up large interfaces into smaller ones if you find that many methods of a type are not used in certain code locations.

4 - UML

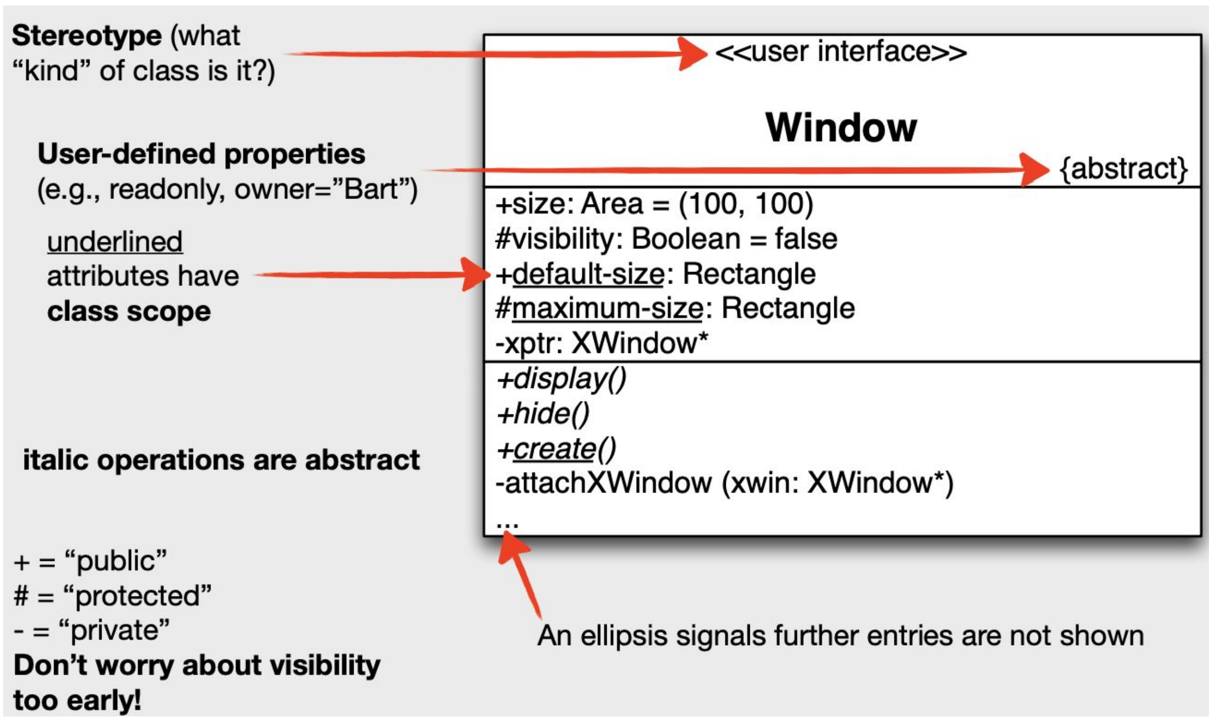
- Reduces risks by documenting assumptions
- domain models, requirements, architecture, design, implementation, ...
- Represents industry standard
    - more tool support, more people understand your diagrams, less education needed
- Is reasonably well-defined
    - ...although there are interpretations and dialects
- Is open
    - stereotypes, tags and constraints to extend basic constructs
    - has a meta-meta-model for advanced extensions



Class Diagrams

- "Class diagrams show generic descriptions of possible systems, and object diagrams show particular instantiations of systems and their behavior"
- Attributes and operations are also called features
- Danger: class diagrams risk turning into data models. Be sure to focus on behavior

Visibility and Scope of Features



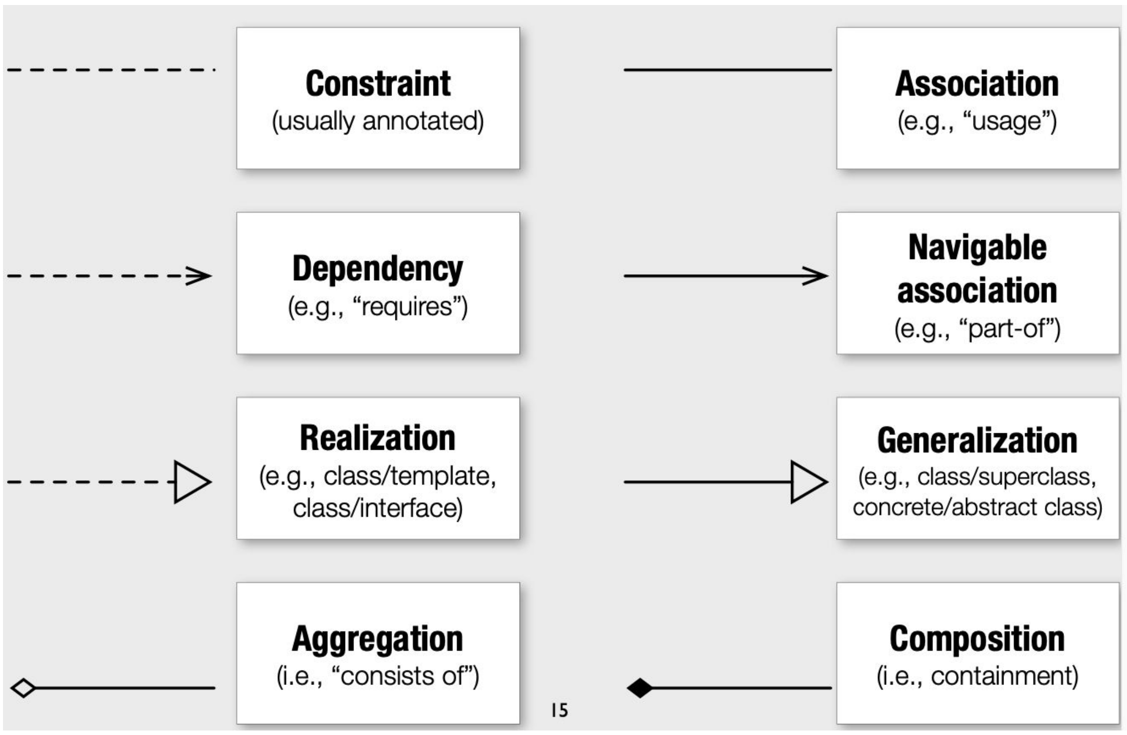
Stereotype (what "kind" of class is it?)
User-defined properties (e.g., readonly, owner="Bart")
underlined attributes have **class scope**
italic operations are abstract
+ = "public"
# = "protected"
- = "private"
Don't worry about visibility too early!
<<user interface>>
Window
{abstract}
+size: Area = (100, 100)
#visibility: Boolean = false
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindow*
+display()
+hide()
+create()
-attachXWindow (xwin: XWindow*)
...
An ellipsis signals further entries are not shown

Attributes

name: type = initialValue { property string }

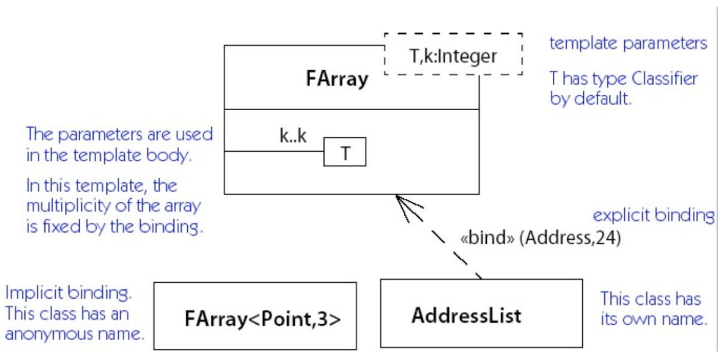
Operations

name (parameter: type = defaultValue, ...) : resultType



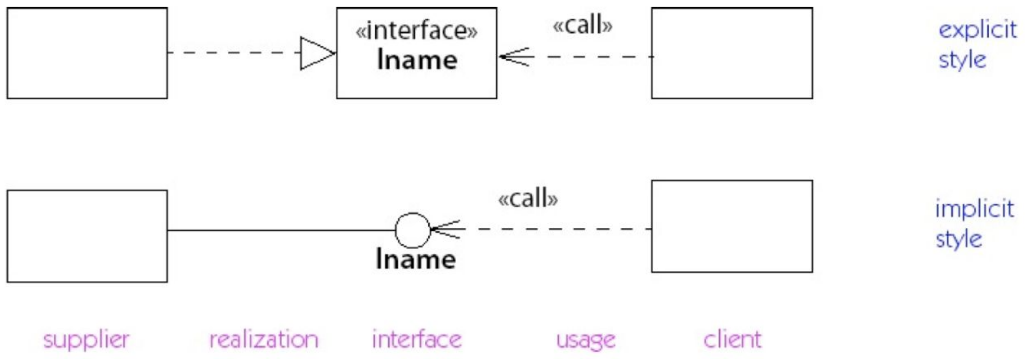
Constraint (usually annotated)
Association (e.g., "usage")
Dependency (e.g., "requires")
Navigable association (e.g., "part-of")
Realization (e.g., class/template, class/interface)
Generalization (e.g., class/superclass, concrete/abstract class)
Aggregation (i.e., "consists of")
Composition (i.e., containment)
15

Parameterized Classes

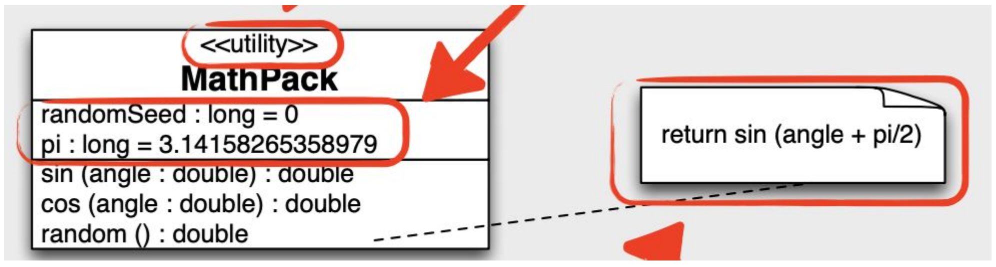

T,k:Integer
FArray
template parameters
T has type Classifier by default.
The parameters are used in the template body.
k..k
T
In this template, the multiplicity of the array is fixed by the binding.
explicit binding
«bind» (Address,24)
Implicit binding. This class has an anonymous name.
FArray<Point,3>
AddressList
This class has its own name.

Interfaces

Interfaces, equivalent to abstract classes with no attributes, are represented as classes with the stereotype <<interface>> or, alternatively with the “Lollipop-Notation”:



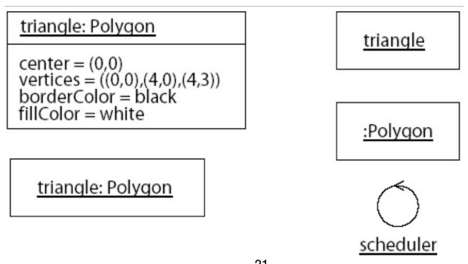
Utilities


<<utility>>
MathPack
randomSeed : long = 0
pi : long = 3.14158265358979
sin (angle : double) : double
cos (angle : double) : double
random () : double
return sin (angle + pi/2)

- A utility is a grouping of global attributes and operations. It is represented as a class with the stereotype <<utility>>. Utilities may be parameterized
- A utility’s attributes are already interpreted as being in class scope, so it is redundant to underline them
- A “note” is a text comment associated with a view, and represented as box with the top right corner folded over

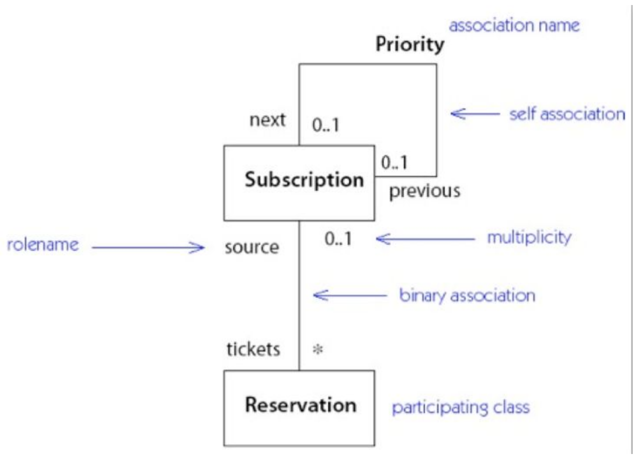
Objects

- Objects are shown as rectangles with their name and type underlined in one compartment, and attribute values, optionally in a second compartment
- At least one of the name or the type must be present



Associations

- Associations represent structural relationships between objects
    - usually binary (but may be ternary, etc.)
    - optional name and direction
    - (unique) role names and multiplicities at end points



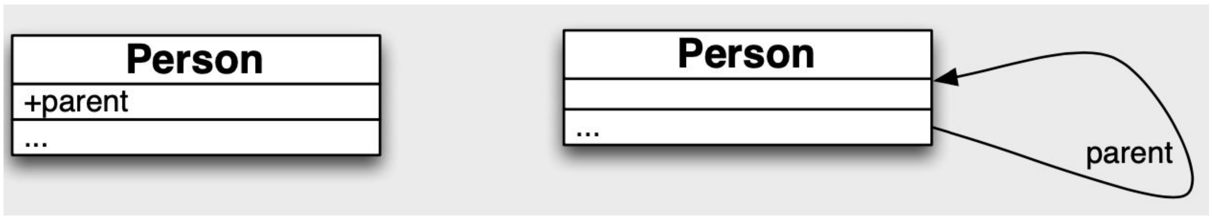
Multiplicity

The multiplicity of an association constrains how many entities one may be associated with

0..1	Zero or one entity
1	Exactly one entity
*	Any number of entities
1..*	One or more entities
1..n	One to n entities
	And so on...

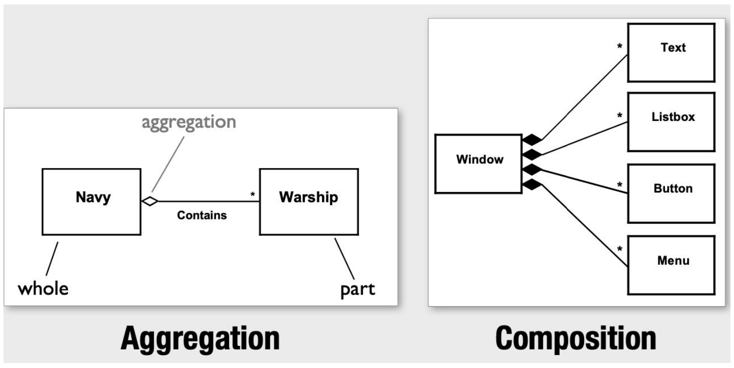
Associations and Attributes

- Associations may (but need not) be implemented as attributes



Aggregation and Composition

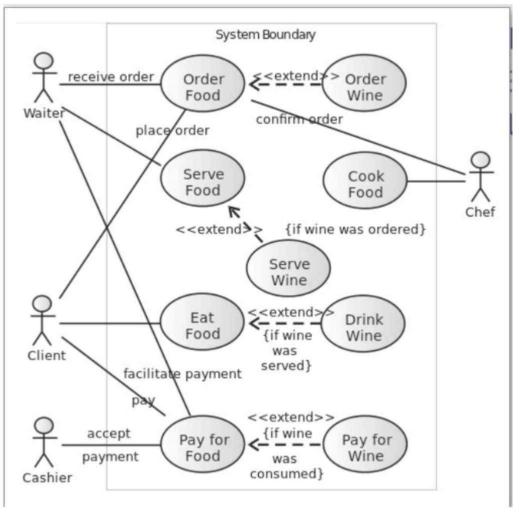
- Aggregation is denoted by a diamond and indicates a part‐‣ whole dependency
- A hollow diamond indicates a reference; a solid diamond an implementation (i.e., ownership)
- Aggregation: parts may be shared
- Composition: one part belongs to one whole



Aggregation or Composition?

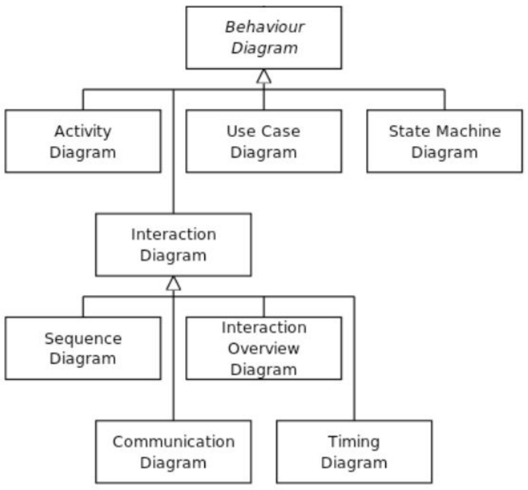
- The decision to use association or aggregation is a matter of ‣ judgment and is often arbitrary
- Example: What type of relationship should be used to model a car with its tires?
    - If the application is a service center, the only reason you care about the tire ‣ is because it is part of the car, so use an composition.
    - If the application is a tire store, you will care about the tire independent of the car, then the relationship should be an aggregation.

Use-Case diagram



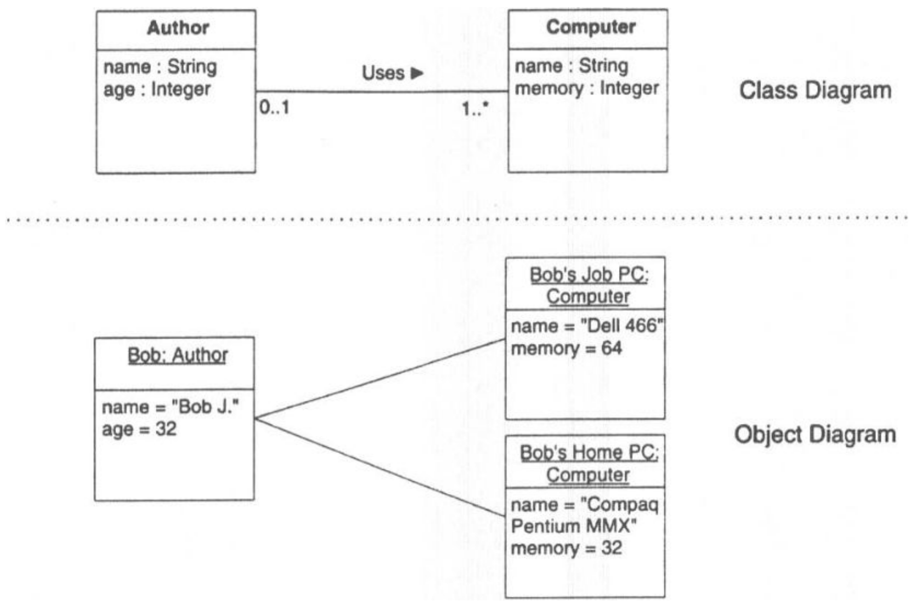
represents the functionality of the system from the point of the user of that system.

Class diagram



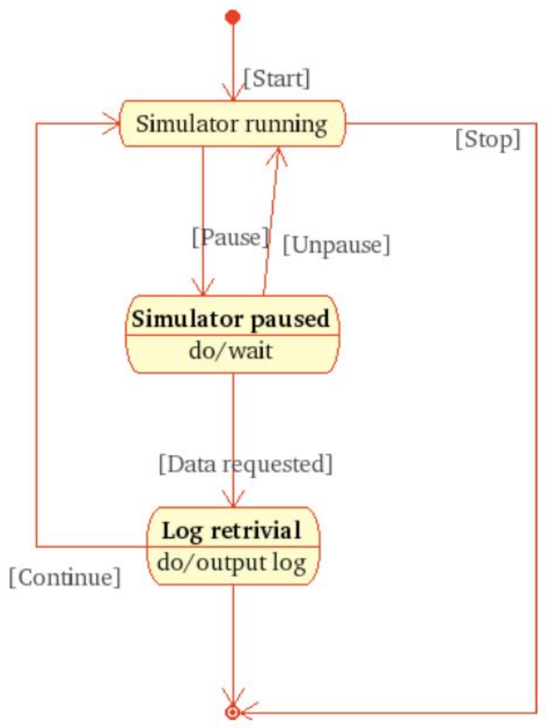
represents the static structure of the system in terms of classes and their relations.

Object Diagram



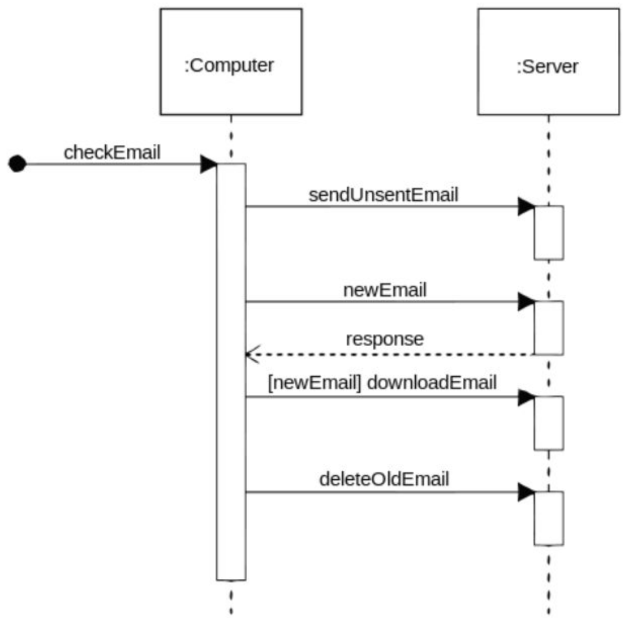
represents objects and their relations; corresponds to simplified collaboration diagrams (no message sends).

State diagram



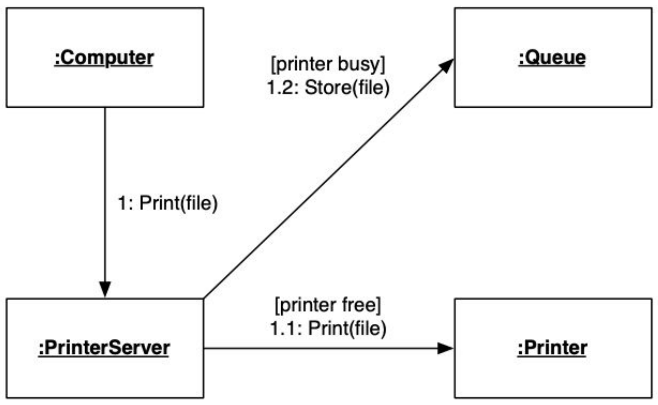
represents the behaviour of a class in terms of (evolution of) its state.

Sequence diagram



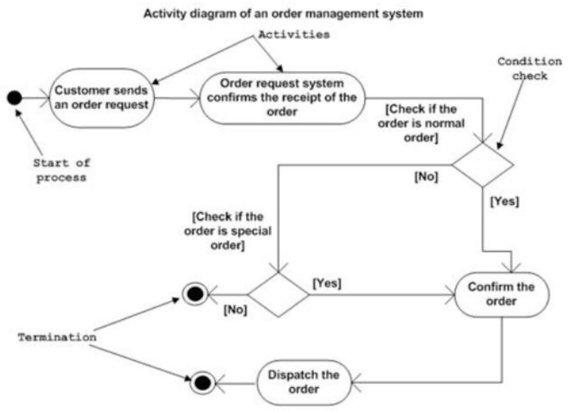
is a temporal representation of objects and their interaction.

Collaboration diagram



is a spatial representation of objects, relations and interactions.

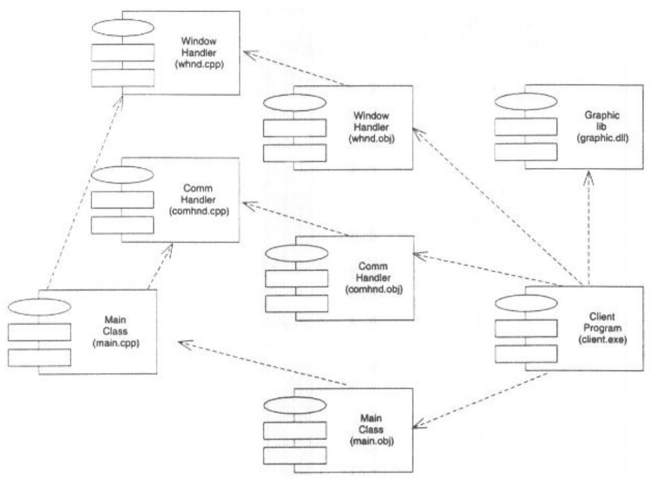
Activity diagram



Activity diagram of an order management system
Activities
Condition check
Customer sends an order request
Order request system confirms the receipt of the order
[Check if the order is normal order]
Start of process
[No]
[Yes]
[Check if the order is special order]
[Yes]
[No]
Confirm the order
Termination
Dispatch the order

represents the behaviour of one operation in terms of actions.

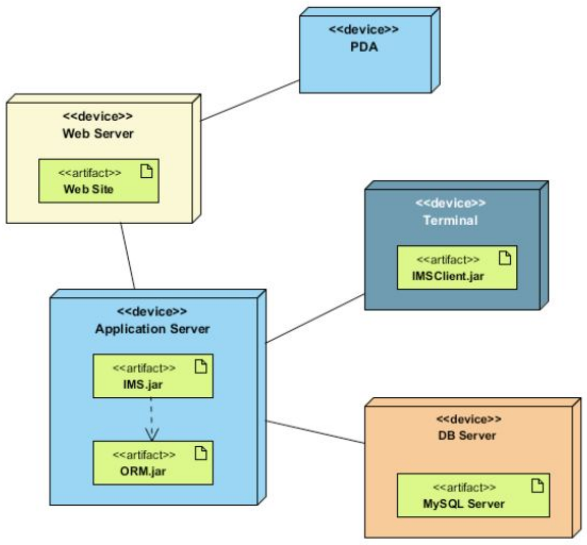
Component diagram



Window Handler (whnd.cpp)
Window Handler (whnd.obj)
Graphic lib (graphic.dll)
Comm Handler (comhnd.cpp)
Comm Handler (comhnd.obj)
Main Class (main.cpp)
Client Program (client.exe)
Main Class (main.obj)

represents the physical components of a system.

Deployment diagram



represents the deployment of a system on hardware.

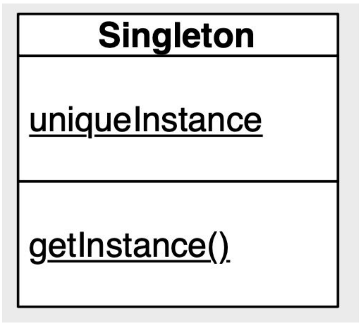
What you should know!

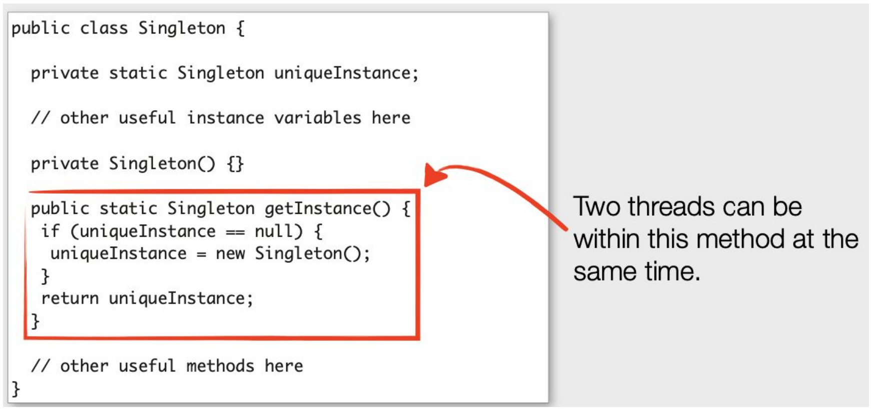
- What is the purpose of a use case diagram?
- Why do scenarios depict objects but not classes?
- How can timing constraints be expressed in scenarios?
- How do you specify and interpret message labels in a scenario?
- How do you use nested state diagrams to model object behavior?
- What is the difference between “external” and “internal” transitions?
- How can you model interaction between state diagrams for several classes?

5 - Design Patterns 1

Singleton Pattern

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.





```
public class Singleton {

  private static Singleton uniqueInstance;

  // other useful instance variables here

  private Singleton() {}

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

  // other useful methods here
}
```

Synchronized Solution

```
public class Singleton {
  private static Singleton uniqueInstance;
  // other useful instance variables here
  private Singleton() {}
public static synchronized Singleton getInstance() { if (uniqueInstance
== null) {
uniqueInstance = new Singleton(); }
   return uniqueInstance;
  }
  // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

Eagerly Created Instance

```
public class Singleton {
private static Singleton uniqueInstance = new Singleton();
  // other useful instance variables here
  private Singleton() {}
  public static Singleton getInstance() {
      return uniqueInstance;
}
  // other useful methods here
}
```

- Go ahead and create an instance of Singleton in a static initializer. This code is thread safe.
- We've already got an instance, so just return it.

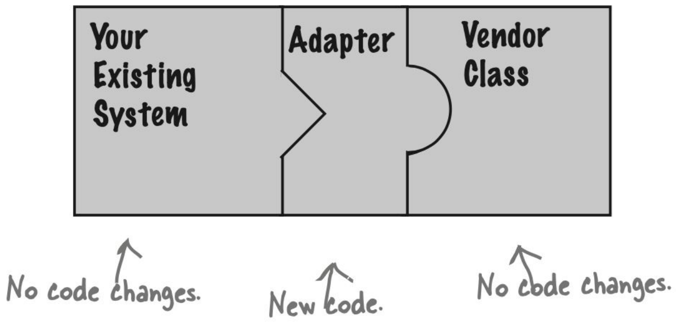
double-checked locking

```
public class Singleton {
  private volatile static Singleton uniqueInstance;
  // other useful instance variables here
  private Singleton() {}
public static Singleton getInstance() { if (uniqueInstance == null) {
synchronized (Singleton.class) { if (uniqueInstance == null) {
uniqueInstance = new Singleton(); }
} }}
  // other useful methods here
}
```

The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

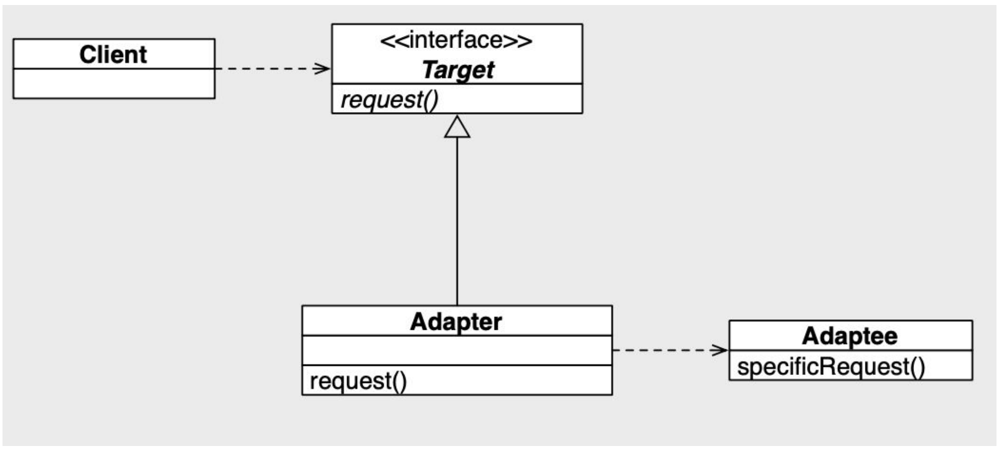
The Adapter Pattern

The **Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.



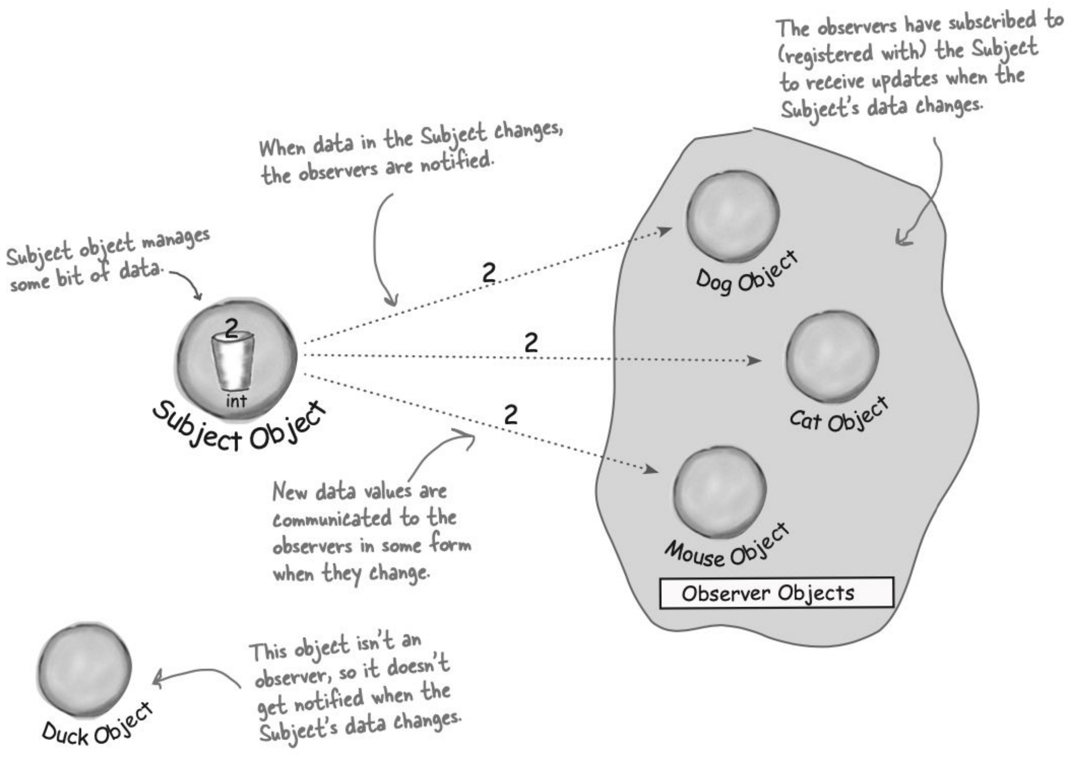
```
WildTurkey turkey = new WildTurkey();
Duck turkeyAdapter = new TurkeyAdapter(turkey);

public interface Turkey {
  public void gobble();
  public void fly();
}

public interface Duck {
  public void quack();
  public void fly();
}
```

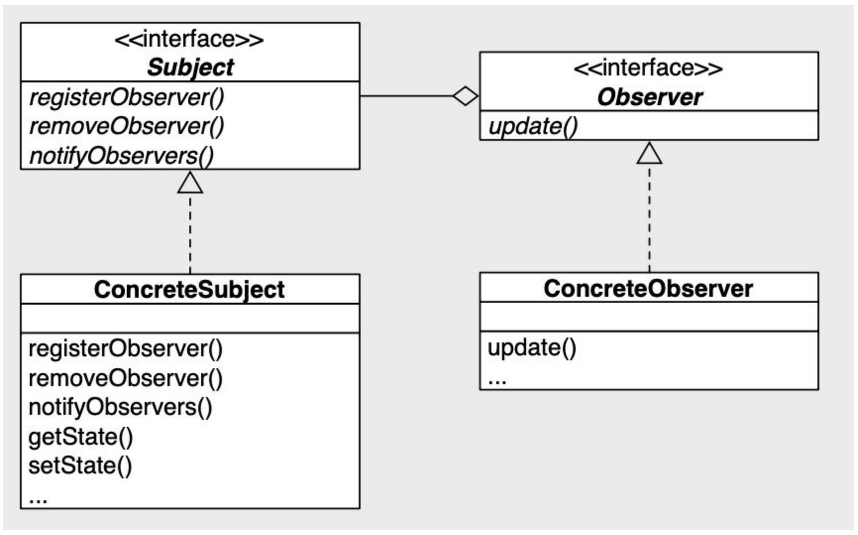
```
public class WildTurkey implements Turkey {
  public void gobble() { System.out.println("Gobble gobble"); }
  public void fly() { System.out.println("I'm flying short distance"); }
}

public class MallardDuck implements Duck {
public void quack() { System.out.println("Quack"); }
public void fly() { System.out.println("I'm flying"); }
}


public class TurkeyAdapter implements Duck {
  Turkey turkey;

  public TurkeyAdapter(Turkey turkey) { this.turkey = turkey; }
  public void quack() { turkey.gobble(); }
  public void fly() {
    for(int i=0; i < 5; i++) {
      turkey.fly();
    }
}
```



Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.


The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.
When data in the Subject changes, the observers are notified.
Subject object manages some bit of data.
2
int
Subject Object
2
2
2
Dog Object
Cat Object
Mouse Object
Observer Objects
New data values are communicated to the observers in some form when they change.
Duck Object
This object isn't an observer, so it doesn't get notified when the Subject's data changes.

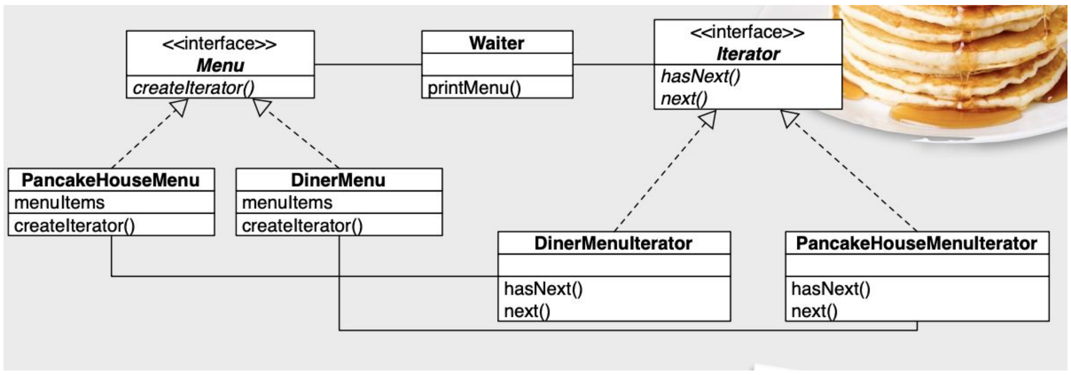


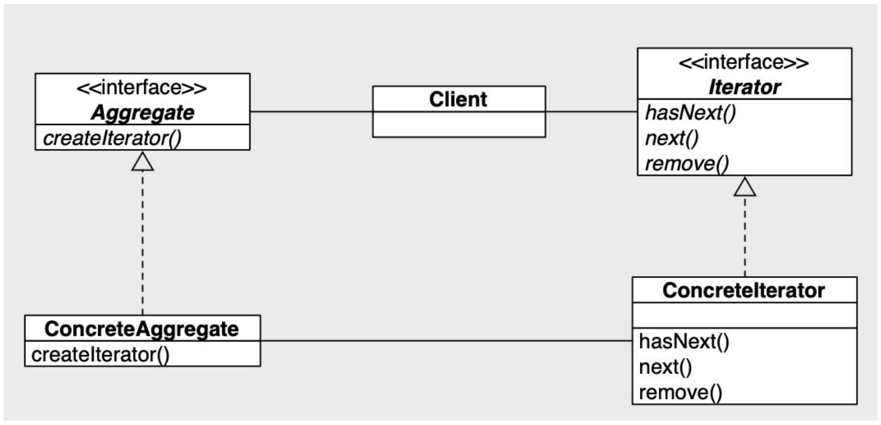
- When two objects are loosely coupled, they can interact, but ‣ have very little knowledge of each other.
- The Observer Pattern provides an object design where subjects and observers are loosely coupled.
    - The only thing the subject knows about an observer is that it implements a certain interface.
    - We can add new observers at any time.
    - We never need to modify the subject to add new types of observers.
    - We can reuse subjects or observers independently of each other.
    - Changes to either the subject or an observer will not affect the other.

Strive for loosely coupled designs between objects that interact.

Iterator Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.





A class should have only one reason to change.

What is a pattern?

- Pattern name
  - Increase of design vocabulary
- Problem description
  - When to apply it, in what context to use it
- Solution description (generic!)
  - The elements that make up the design, their relationships, responsibilities, and collaborations
- Consequences
  - Results and trade-offs of applying the pattern

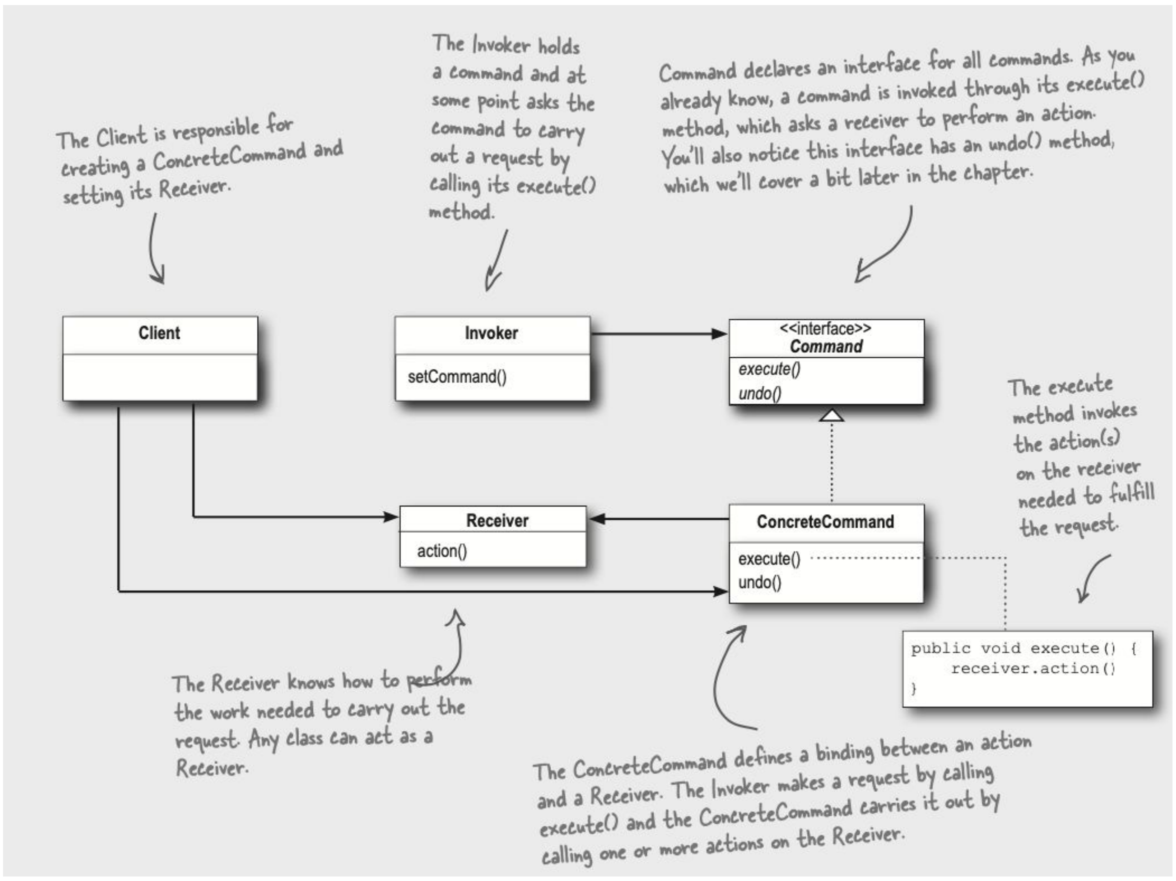
Why design patterns?

- Smart
  - Elegant solutions that a novice would not think of
- Generic
  - Independent on specific system type, language
  - Although slightly biased towards C++
- Well-proven

    - Successfully tested in several systems
- Simple
    - Combine them for more complex solutions
- Well-known
    - Can be used to communicate complex ideas more easily

Command Pattern

The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.

The execute method invokes the action(s) on the receiver needed to fulfill the request.

```
public void execute() {
    receiver.action()
}
```

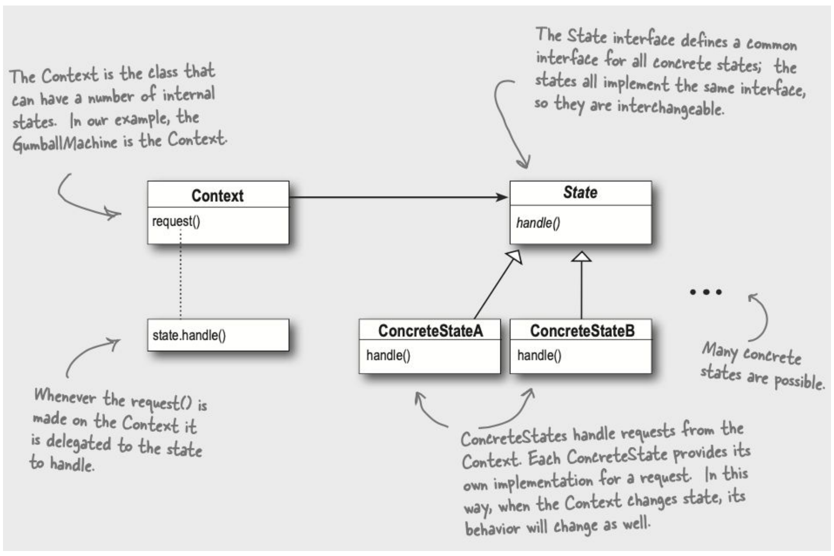
The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

```
// Interface
public interface Command {
  public void execute();
}

// Implementation
public class LightCommandOn
                implements Command {
Light light;
public LightOnCommand(Light l) { this.light = l;
}
public void execute() { light.on();
} }

// Using command
public class SimpleRemoteControl {
  Command slot;
  public SimpleRemoteControl() {}
  public void setCommand(Command c) {
    slot = c;
  }
  public void pressButton() {
    slot.execute();
  }
}

// Testing it
@Test
public void testSimpleRemoteControl() {
SimpleRemoteControl r = new SimpleRemoteControl();
Light light = new Light();
LightCommandOn lightOn = new LightCommandOn(light);
remote.setCommand(lightOn);
remote.pressButton();
}
```

State Pattern

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.
The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.
Context
request()
State
handle()
state.handle()
ConcreteStateA
handle()
ConcreteStateB
handle()
Many concrete states are possible.
Whenever the request() is made on the Context it is delegated to the state to handle.
ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

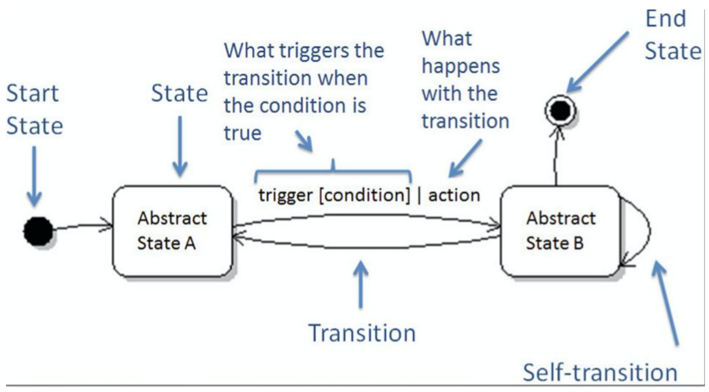
7 - Object State

- Use Class Diagram for static perspective
- Use different UML Diagrams for dynamic perspective

Object State
The state of an object refers to:
- the pieces of information the object represents at a given moment
- the actions that the object can perform at a given moment

the software design task of state space partitioning is to define abstract states that correspond to characteristics that will help construct a clean solution.

Definitions:
- state space - “set of all possible states for an object”
- concrete state - “value(s) stored at a given moment”
- abstract state - “arbitrarily-defined subset of the state space”


Start State
State
What triggers the transition when the condition is true
What happens with the transition
End State
trigger [condition] | action
Abstract State A
Abstract State B
Transition
Self-transition

state diagrams allow us to self-evaluate the impact of design decisions on the complexity of the abstract state space that must be considered when working with an object

Minimize the state space of objects to what is absolutely necessary for the object to fulfill its responsibilities.

- Try to not have temporary fields, this is an anti patterns and introduces unnecessary state

Nullability
Should not be null case

```
public class Card {
 private Rank aRank; // Should not be null
 private Suit aSuit; // Should not be null
 public Card(Rank pRank, Suit pSuit) {
  aRank = pRank;
  aSuit = pSuit;
 }
}
```

Cannot be null case

```
public class Card {
private Rank aRank; // Cannot be null
private Suit aSuit; // Cannot be null

public Card(Rank pRank, Suit pSuit) {
if( pRank == null || pSuit == null) {
throw new IllegalArgumentException();
}
aRank = pRank;
aSuit = pSuit;
}
}
```

Assert they not null

```
/**
* @pre pRank != null && pSuit != null;
*/
public Card(Rank pRank, Suit pSuit) {
 assert pRank != null && pSuit != null;
 aRank = pRank;
 aSuit = pSuit;
}
```

Final Fields

```
public class Deck {
private final List<Card> aCards = new ArrayList<>();
}
```

Final fields do not make the referenced objects immutable.

You can not replace this list with a completely new list.

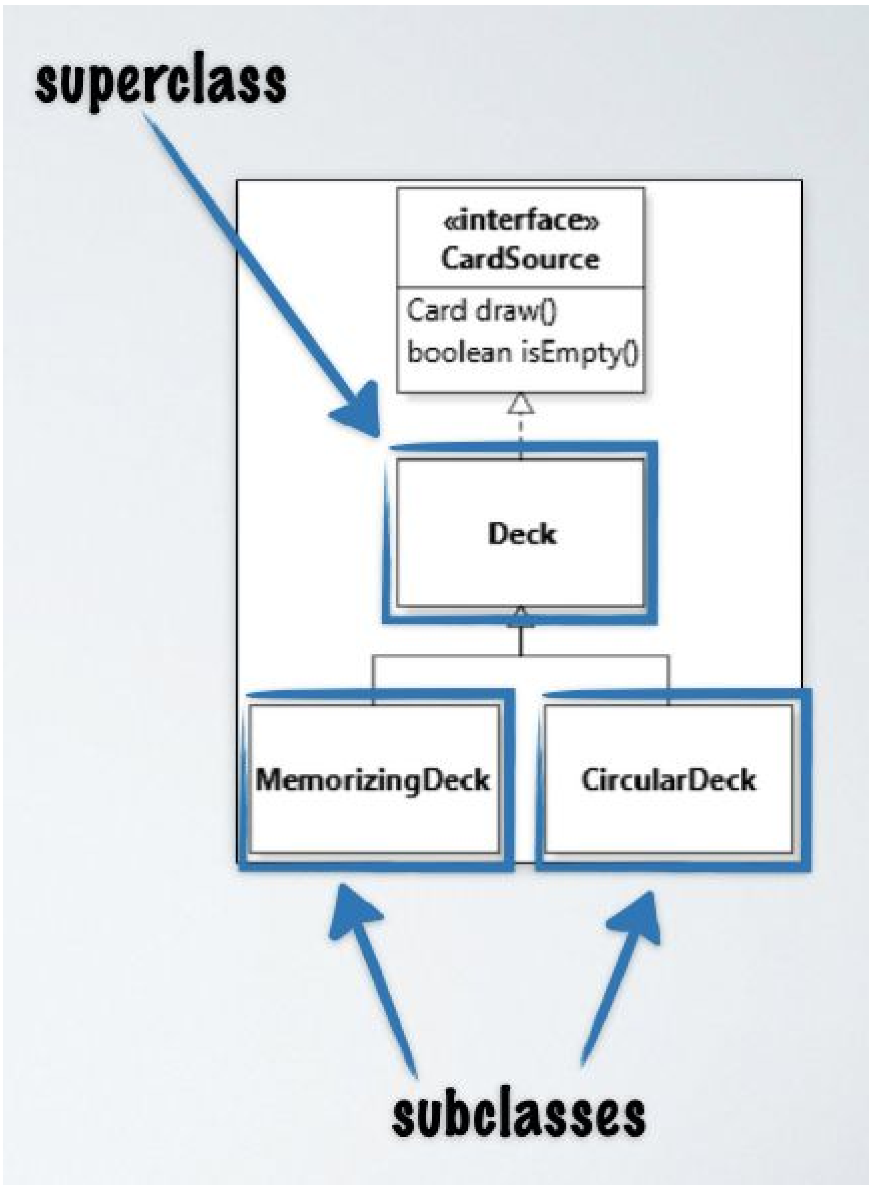
Object Identity, Equality, and Uniqueness
Replacing the equals method from the class object.

```
public boolean equals(Object pObject) {
if( pObject == null )
{ return false; } // As required by the specification else if( pObject
== this )
{ return true; } // Standard optimization
else if( pObject.getClass() != getClass())
{ return false; }
else
{
// Actual comparison code
return aRank == ((Card)pObject).aRank && ((Card)pObject).aSuit == aSuit;
} }
```

What you should know and do!

- For stateful classes, consider using state diagrams to reason systematically about the abstract states of the object of the classes and their life cycle;
- Try to minimize the number of meaningful abstract states for objects of a class;
- Avoid using null references to represent legal information in objects and variables;
- Consider declaring instance variables final whenever possible;
- Be explicit about whether objects of a class should be unique or not;
- If objects are not designed to be unique, override the equals and hashCode methods;
- Consider using the Singleton pattern for classes that should yield only one instance of a stateful object.

8 - Inheritance



Defining a subclass (e.g., MemorizingDeck) as an extension of a superclass (e.g., Deck) means that when objects of the subclass are instantiated, the objects will be created by using the declaration of the subclass and of the declaration of the superclass.

In Java, once an object is created, its run-time type remains unchanged.

How to define a subclass in Java

```java
public class MemorizingDeck extends Deck {
private CardStack aDrawnCards;
}
```

Casting

```
public static boolean isMemorizing(Deck pDeck){
return pDeck instanceof MemorizingDeck;
}

public static void main(String[] args) {
 Deck deck = new MemorizingDeck();
 MemorizingDeck memorizingDeck = (MemorizingDeck) deck; // downcast
 boolean isMemorizing1 = isMemorizing(deck); // true
 boolean isMemorizing2 = isMemorizing(memorizingDeck);// true
}
```

Downcasting
Checks whether downcast works

```
assert deck instanceof MemorizingDeck;
// or
if (deck instanceof MemorizingDeck) {
return ((MemorizingDeck)deck).getDrawnCards();
}
```

Classes in Java are organized into a single-rooted class hierarchy. If a class does not declare to extend any class, by default it extends the library class Object.
Class Object constitutes the root of any class hierarchy in Java.

In Java the fields of an object are initialized “top down”, from the field declarations of the most general superclass down to the most specific class (the one named in the **new** statement).

Initializer auto. calls super.

An instance (i.e., non-static) method is just a different way to express a function that takes an object of its declaring class as its first argument

```
@override // needed if you override super class method
Public Card draw(){

}
```

Overloading
Declaring the same method with different argument types.

9 - Debugging and Testing

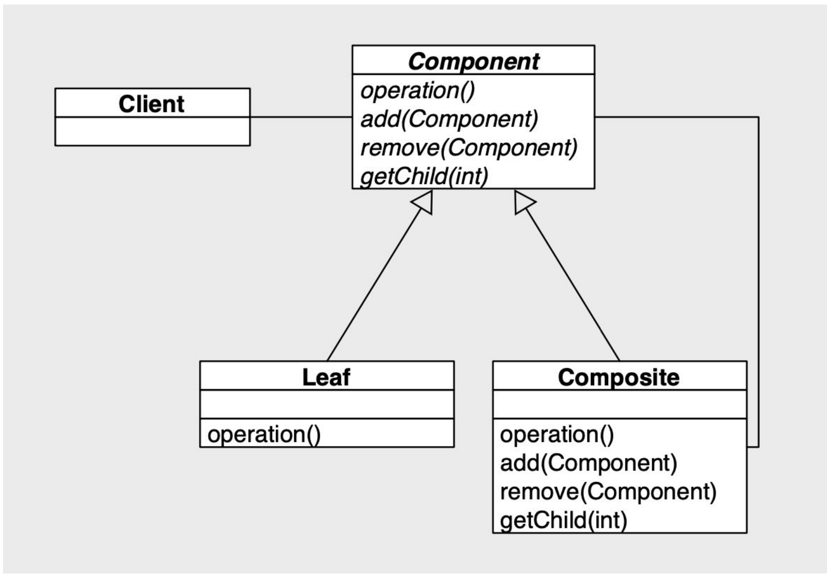
See Slides I guess

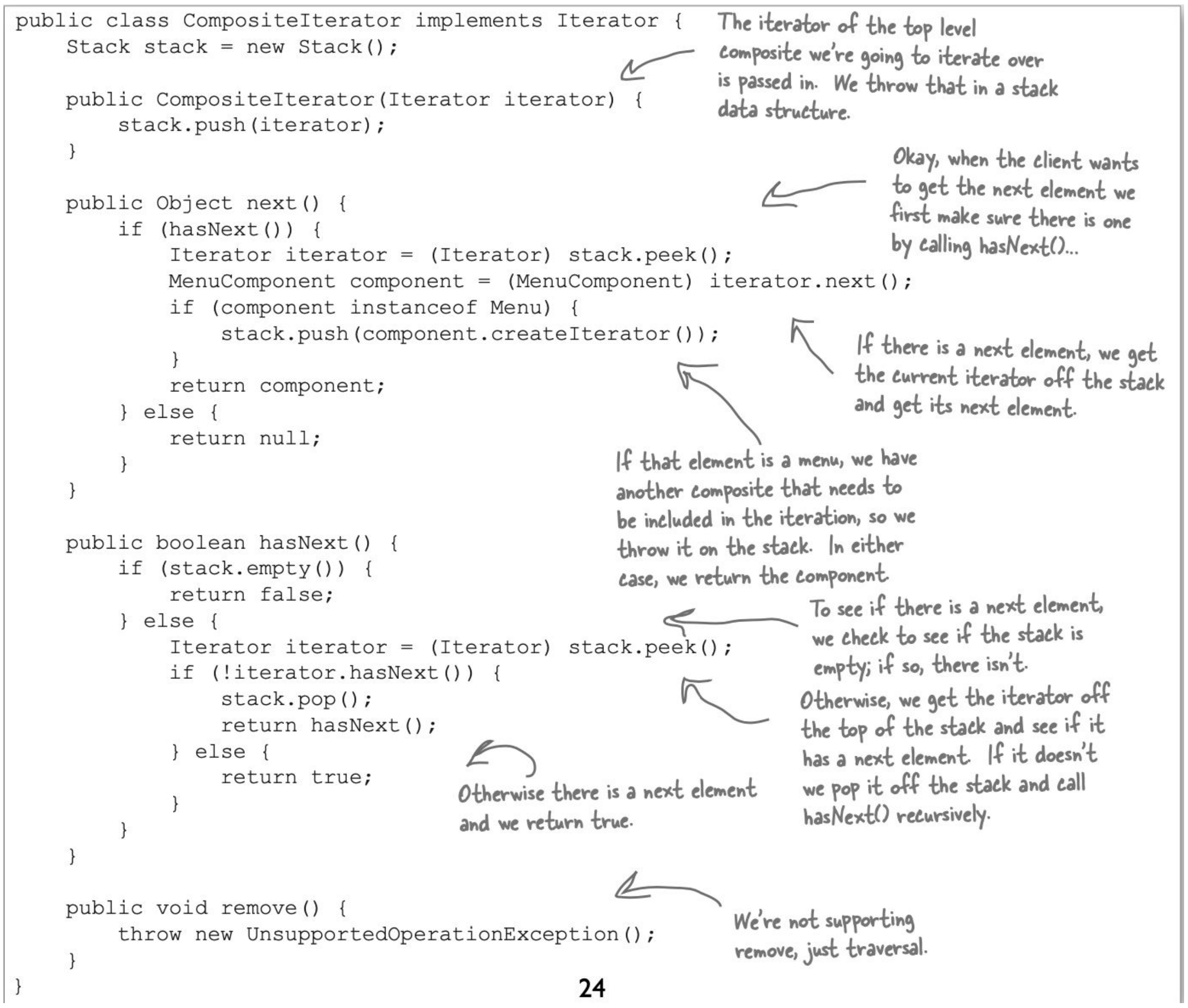
10 - Design Patterns 3

The Composite Pattern

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.
- Using a composite structure, we can apply the same operations over both composites and individual objects.
- In other words, in most cases we can ignore the differences between compositions of objects and individual objects.





```
public class CompositeIterator implements Iterator {
    Stack stack = new Stack();

    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator iterator = (Iterator) stack.peek();
            MenuComponent component = (MenuComponent) iterator.next();
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
            return component;
        } else {
            return null;
        }
    }

    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator iterator = (Iterator) stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

The iterator of the top level composite we're going to iterate over is passed in. We throw that in a stack data structure.

Okay, when the client wants to get the next element we first make sure there is one by calling hasNext()...

If there is a next element, we get the current iterator off the stack and get its next element.

If that element is a menu, we have another composite that needs to be included in the iteration, so we throw it on the stack. In either case, we return the component.

To see if there is a next element, we check to see if the stack is empty; if so, there isn't.

Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call hasNext() recursively.

Otherwise there is a next element and we return true.

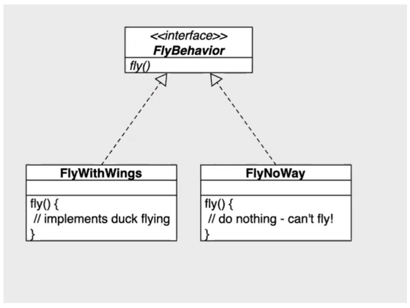
We're not supporting remove, just traversal.

24

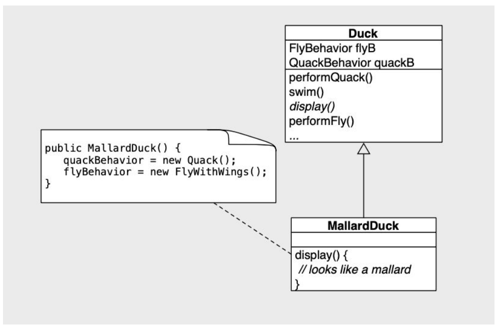
Behavior Pattern

Create an interface for a pattern and then create actual implementations of that pattern.

The Duck behaviors will live in a separate class,
a class that implements a particular behavior interface.
Other types of objects can reuse our behaviors.



Favor composition over inheritance



```
public MallardDuck() {
   quackBehavior = new Quack();
   flyBehavior = new FlyWithWings();
}
```

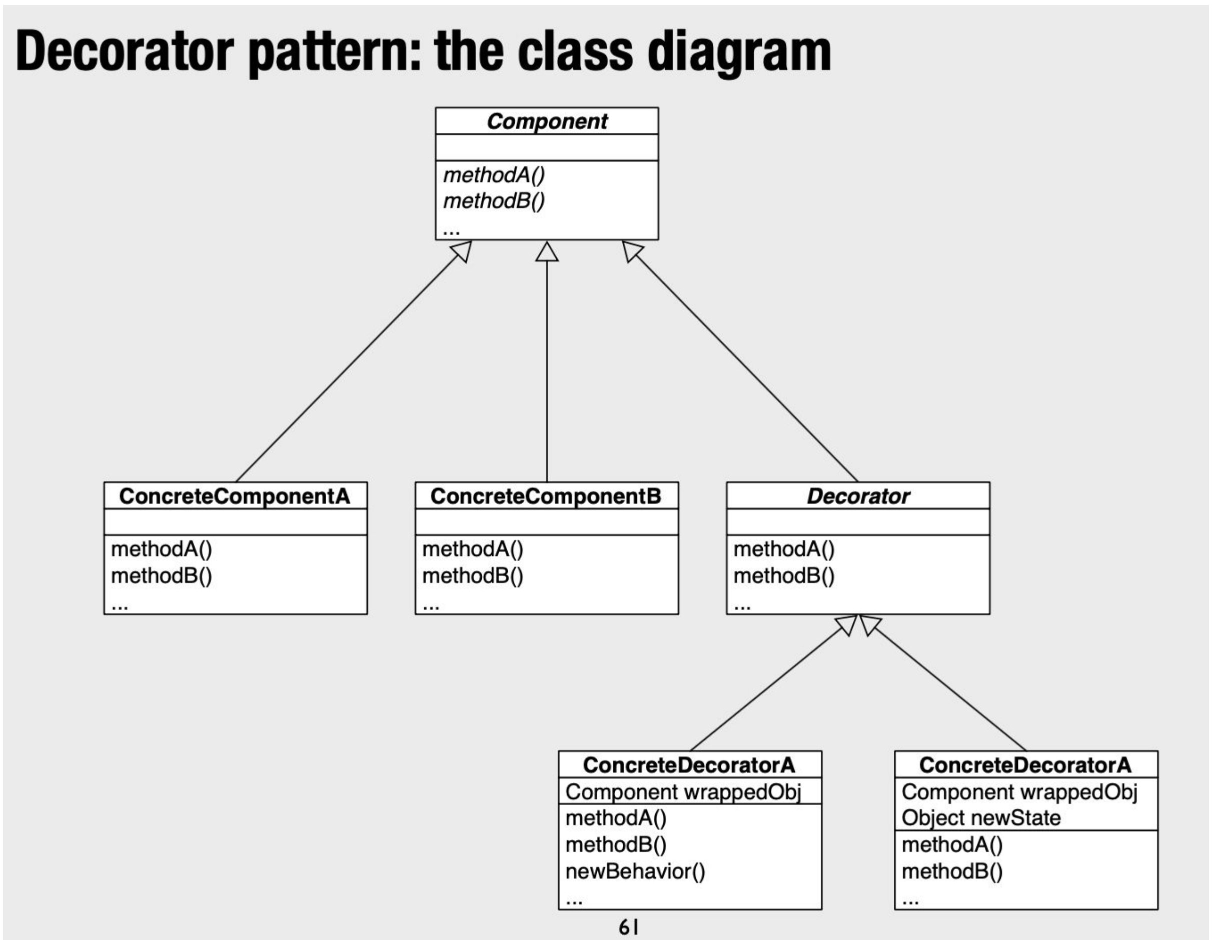
The strategy Pattern

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Classes should be open for extension, but closed for modification.

The Decorator Pattern

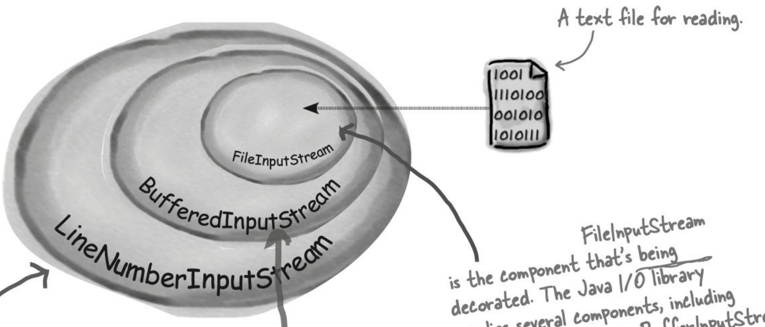
The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Inheritance vs Composition

- We are using inheritance to achieve the type matching, but we are not using inheritance to get behavior.
- We are acquiring new behavior not by inheriting it from a superclass, but by composing objects together. With composition, we can mix and match decorators any way we like at runtime.
- Traditionally the Decorator Pattern does specify an abstract component, but in Java, obviously, we could use an interface.

Real world decorators: Java I/O



LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

FileInputStream is the component that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

12 - Responsibility-Driven Design

Why use Responsibility-Driven Design?

Functional Decomposition: Decompose according to the **functions** a system is supposed to perform:

- Good ina “waterfall” approach: stable requirements and one monolithic function
- However
  - Naive: modern systems perform more than one function
  - Maintainability: System functions evolve >> redesign affects the whole system
  - Interoperability: Interfacing with other systems is difficult

Object-Oriented Decomposition: Decompose according to the **objects** a system is supposed to manipulate

- Better for complex and evolving systems
- However: how to find the objects?

- The result of the design process is not a final product
  - Design decisions may be revisited, even after implementation
  - Design is not linear but iterative

- The design process is not algorithmic
  - A design method provides guidelines, not fixed rules
  - A good sense of style often helps to produce clean, elegant designs, designs that make sense from the engineering standpoint

Responsibility-Driven Design is a design (and analysis) technique that works well in combination with various methods and notations

Finding Classes

Flow:

Left: Write the guidelines

Middle: Pre ideas of concrete classes

Right: Concrete Class Ideas

Initial Exploration

- Find the **classes** in your system
- Determine the **responsibilities** of each class
- Determine how objects **collaborate** with each other to fulfill their responsibilities

The Detailed Analysis

- **Factor** common responsibilities to build class hierarchies

- **Streamline** collaborations between objects
    - Is message traffic heavy in parts of the system?
    - Are there classes that collaborate with everybody?
    - Are there classes that collaborate with nobody?
    - Are there groups of classes that can be seen as subsystems?
- Turn class responsibilities into fully specified signatures

Start with requirements specification: What are the goals of the system being designed, its expected inputs and desired responses?

Look for noun phrases
- Separate into obvious classes, uncertain candidates, and nonsense

Refine to a list of **candidate classes**. Some guidelines:
- Model **physical objects** - e.g., disks, printers
- Model **conceptual entities** - e.g., windows, files
- Choose **one word for one concept** - what does it mean in the system?
- Be wary of **adjectives** - is it really a separate class?
- Be wary of **missing or misleading subjects** - rephrase in active voice
- Model **categories of classes** - delay modeling of inheritance
- Model **interfaces** to the system - e.g., user interface, program interfaces
- Model attribute **values**, not attributes - e.g., Point vs. Centre

Class Selection Rationale

Choose one word for one concept
- Drawing Editor ~ ~~editor, interactive graphics editor~~

Be vary of adjectives
- Group them accordingly

Be wary of sentences with missing or misleading subjects

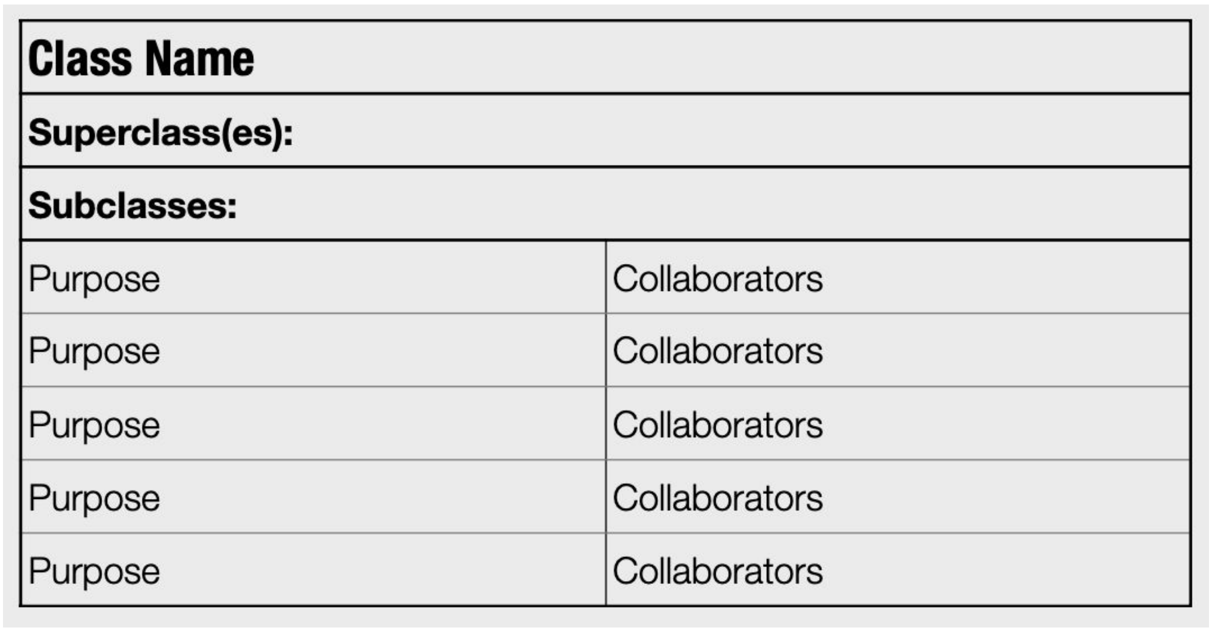
Model Categories

Model interfaces to the system
- ~~user~~ - don't need to model user explicitly
- ~~cursor~~ - cursor motion handled by operating system

Model values of attributes, not attributes themselves
- ~~height of the rectangle, width of the rectangle~~
- ~~major radius, minor radius~~
- ~~position~~ - of first text character, probably Point attribute
- ~~mode of operation~~ - attribute of Drawing Editor
- ~~shape of the cursor, I-beam, crosshair~~ - attributes of Cursor

- ~~corner~~ - attribute of Rectangle
- ~~time~~ - an implicit attribute of the system

CRC (Class-responsibility-collaboration) Sessions



Class Name	
Superclass(es):	
Subclasses:	
Purpose	Collaborators
Purpose	Collaborators
Purpose	Collaborators
Purpose	Collaborators
Purpose	Collaborators

- Record the candidate Class Name and superclass (if known)
- Record each Responsibility and the Collaborating Classes
  - compact, easy to manipulate, easy to modify or discard!
  - easy to arrange, reorganize
  - easy to retrieve discarded classes

CRC Cards are **not** a specification of a design

They are a tool to **explore** possible designs
- Prepare a CRC Card for **each candidate class**
- Get a team of developers to **sit around a table** and distribute the cards to the team
- The team **walks through scenarios**, playing the roles of the classes

This exercise will uncover
- **unneeded** classes and responsibilities
- **missing** classes and responsibilities
  - Cut Buffer class

Identifying Responsibilities

Responsibilities are meant to convey a **sense of the purpose** of an
object and its **place** in the system

What are responsibilities in practice?

- the <u>knowledge</u> an object maintains and provides
- the <u>actions</u> it can perform

Responsibilities represent the **public services** an object may provide to clients (but not the way in which those services may be implemented)

- specify **what** an object does, not **how** it does it
- don't describe the interface yet, only **conceptual responsibilities**

Identifying Responsibilities
Each class exists because of an implied responsibility

Study the requirements specification

- highlight **verbs** and determine which represent responsibilities
- perform a **walk-through** of the system
  - **Explore as many scenarios as possible**
  - **Identify actions resulting from input to the system**

Study the candidates classes

- class names >> roles >> responsibilities
- recorded purposes on class cards >> responsibilities

How to Assign Responsibility
"Don't do anything you can push off to someone else"

"Don't let anyone else play with you"

Evenly distribute system intelligence

- avoid centralization of responsibilities
  - Clear control flow, but hard-wired system's behavior

State responsibilities as **generally** as possible

- "draw yourself" vs. "draw a line/rectangle"
- leads to sharing

Keep **behavior** together with any **related information**

- Known as principle of **Encapsulation**

Keep information about one thing in one place

- If multiple objects need access to the same information (e.g., location in Drawing and Drawing Element)
  - A new object may be introduced to manage the information
  - One object may be introduce to manage the information
  - One object may be an obvious candidate
  - The multiple objects may need to be collapsed into one

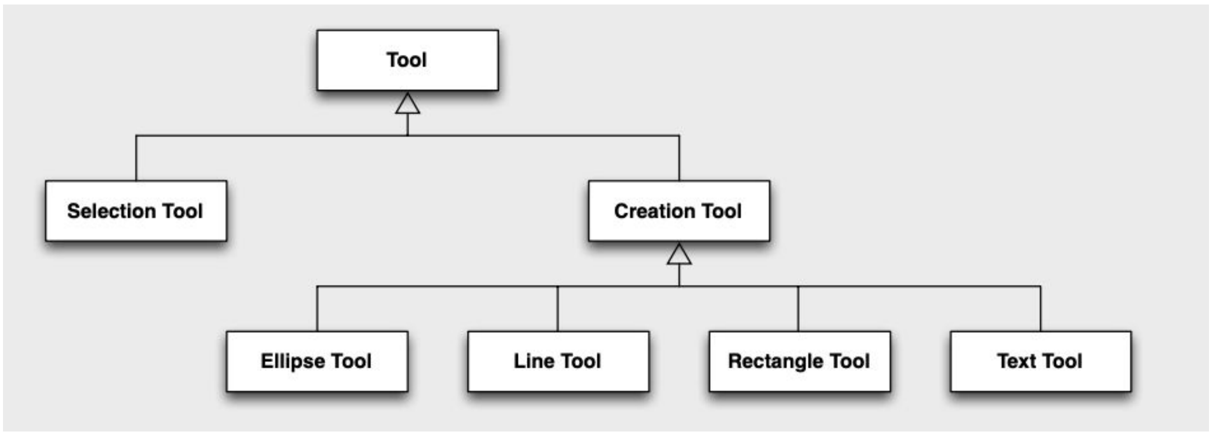
Share responsibilities among related objects

- Break down complex responsibilities
- A Drawing Editor knows when the drawing has changed; the Drawing knows which elements to display; each Drawing Element knows how and where its presentation should be drawn …

Relationships Between Classes

Additional responsibilities can be uncovered by examining relationships between classes, especially

- The **“Is-Kind-Of”** Relationship:
    - classes sharing a **common attribute** often share a **common superclass**
    - common superclasses suggest **common responsibilities**
    - for example: To create a new Drawing Element, a Creation Tool must
        - accept user input - **implemented in subclass**
        - determine location to place it - **generic**
        - instantiate the element - **implement in subclass**



The “**Is-Analogous-To”** Relationship: similarities between classes suggest as-yet-undiscovered superclasses

The **“Is-Part-Of”** Relationship: distinguish (don’t share) responsibilities of part and of whole

EXAMPLE:

- Drawing element is-part-of Drawing
- Drawing Element has-knowledge-of Control Points
- Rectangle Tool is-kind-of Creation Tool

Difficulties in assigning responsibilities suggest:

- **missing classes** in design - e.g., Group Element in selection
- **free choice** between multiple classes

Finding Collaborations

What are Collaborations?
Collaborations are client requests to servers needed to fulfill responsibilities

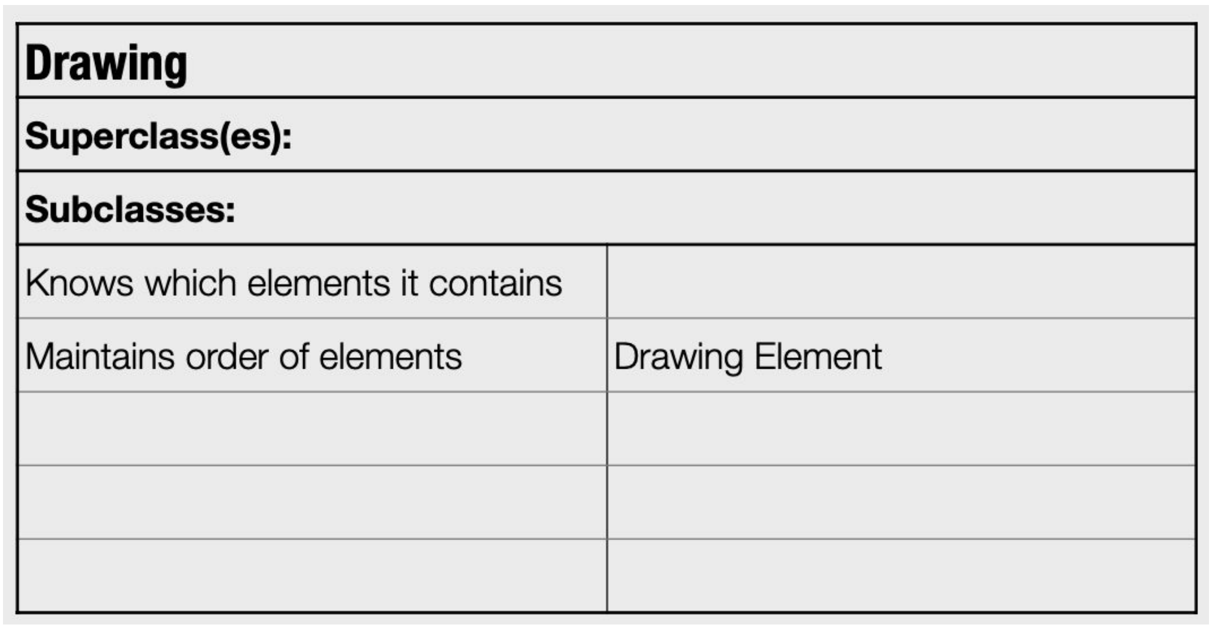
Collaborations reveal control and information flow, and ultimately subsystems

Collaborations can uncover missing responsibilities

Analysis of communication patterns can reveal wrongly assigned responsibilities

Finding Collaborations
For each responsibility
- Can the class **fulfill** the responsibility by **itself**?
- If not, **what does it need**, and from what other class can it obtain what it needs?

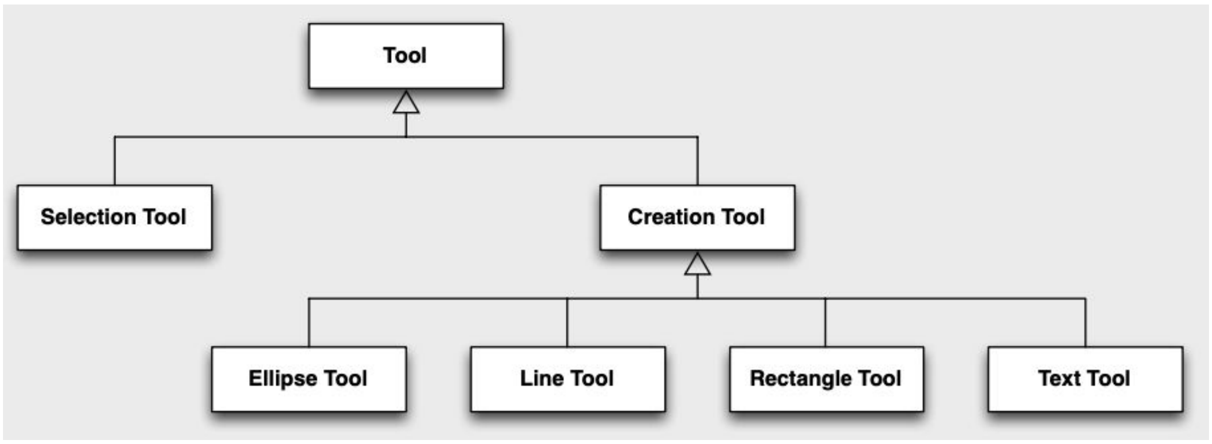
For each class
- What does this class **know**?
- What **other classe**s need its information or results? Check for collaborations
- Classes that **do not interact** with others should be **discarded**, but check carefully



Drawing	
Superclass(es):	
Subclasses:	
Knows which elements it contains	
Maintains order of elements	Drawing Element

Structuring Inheritance Hierarchies

Finding Abstract Classes
Abstract classes factor out common behavior shared by other classes
- group related classes with common attributes
- introduce abstract superclasses to represent the group
- “categories” are good candidates for abstract classes
- Warning: beware of premature classification; your hierarchy will evolve!



Building Good Hierarchies

Model a "kind-of" hierarchy

- Subclasses should support all inherited responsibilities, possibly more

Factor common responsibilities as high as possible

- Classes that share common responsibilities should inherit from a common abstract superclass; introduce any that are missing

Make sure that abstract classes do not inherit from concrete classes

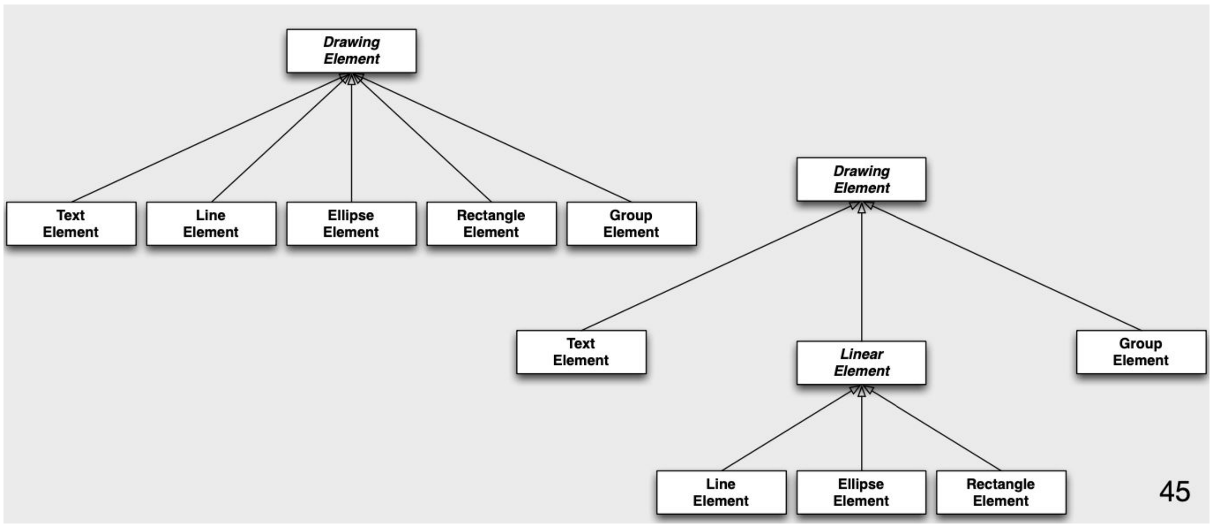
- Eliminate by introducing common abstract superclass: abstract classes should support responsibilities in an implementation-independent way

Eliminate classes that do not add functionality

- Classes should either add new responsibilities, or a particular way of implementing inherited ones

Refactoring Responsibilities

Lines, Ellipses, and Rectangles are responsible for keeping track of the width and color of the lines they are drawn with - this suggests a common superclass



Protocols

A protocol is a **set of signatures** (i.e., an **interface**) to which a class will respond

- Generally, protocols are defined for **public responsibilities**
- Protocols for ~~private~~ **protected** responsibilities should be specified if they will be used or implemented by **subclasses**

Therefore,

- Construct protocols for each class
- Write a design specification for each class and subsystem
- Write a design specification for each contract

What you should know!

What criteria can you use to identify potential classes?

How can CRC cards help during analysis and design?

How can you identify abstract classes?

What are class responsibilities, and how can you identify them?

How can identification of responsibilities help in identifying classes?

What are collaborations, and how do they relate to responsibilities?

What criteria can you use to design a good class hierarchy?

How can refactoring responsibilities help to improve a class hierarchy?

When should an attribute be promoted to a class?

Why is it useful to organize classes into a hierarchy?

How can you tell if you have captured all the responsibilities and collaborations?

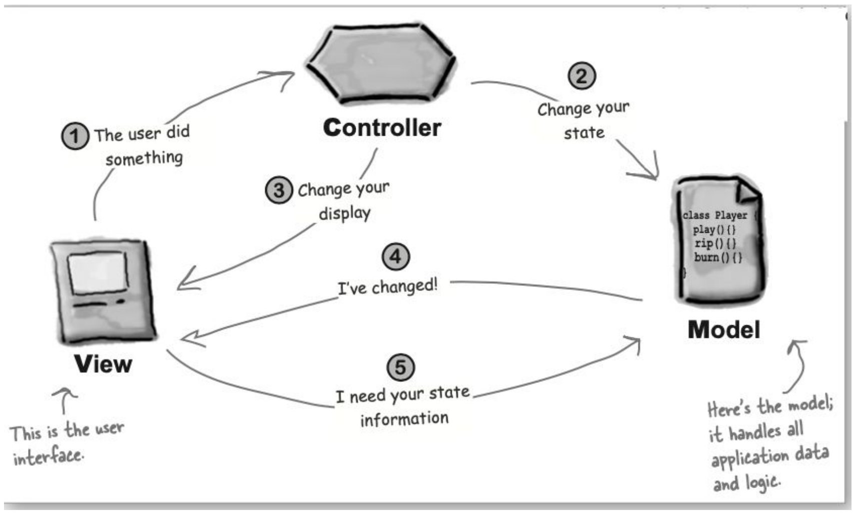
What use is multiple inheritance during design if your programming language does not support it?

Lecture 13 - Model View Controller

Model View Controller is a **Compound Pattern**

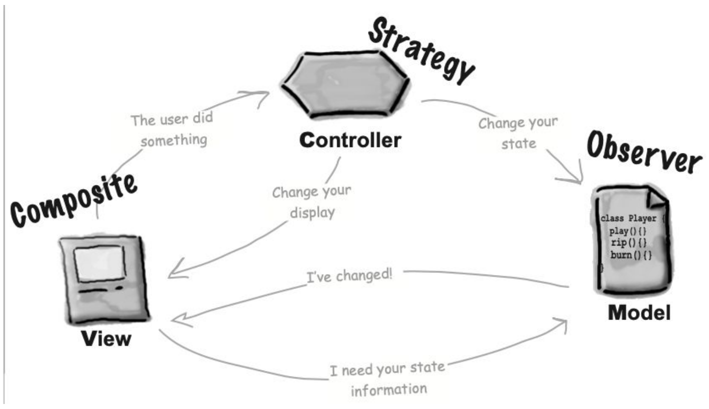
- You can have as many views for your Model using the Observer Pattern

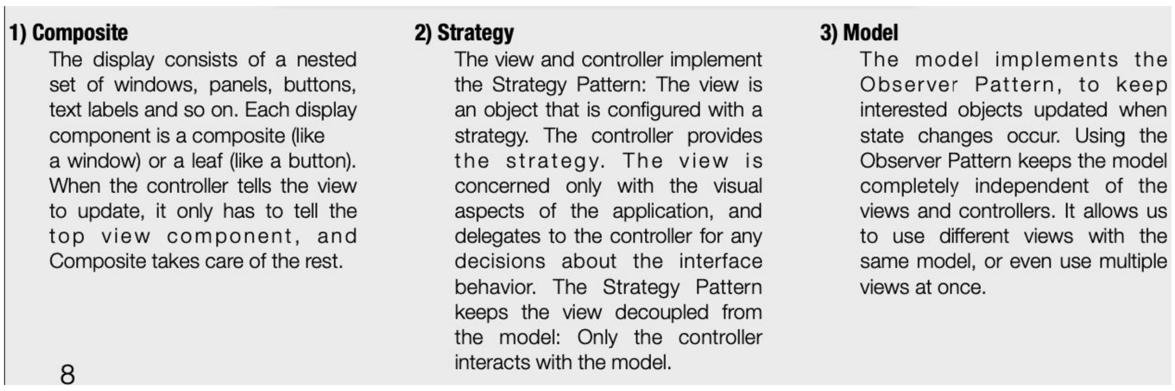
MVC Pattern



Controller
1 The user did something
2 Change your state
3 Change your display
4 I've changed!
5 I need your state information
View
Model
class Player play() {} rip() {} burn() {}
This is the user interface.
Here's the model; it handles all application data and logic.

Looking at MVC with design patterns





1) Composite

The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

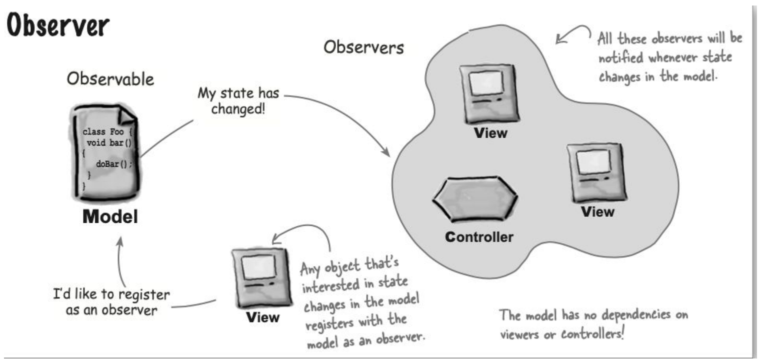
2) Strategy

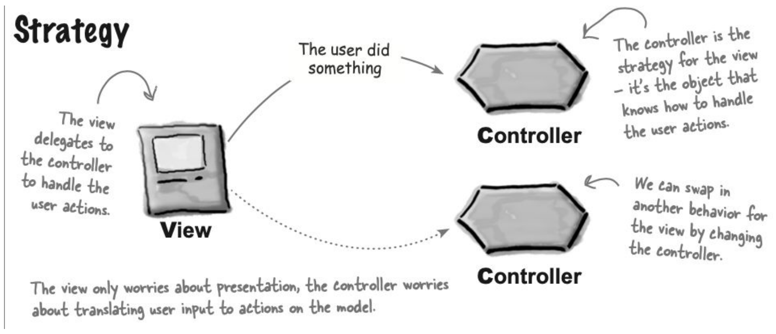
The view and controller implement the Strategy Pattern: The view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. The Strategy Pattern keeps the view decoupled from the model: Only the controller interacts with the model.

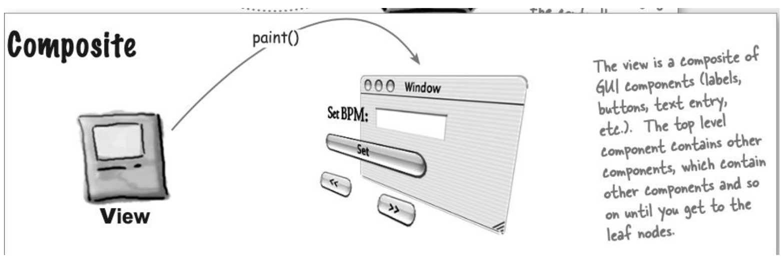
3) Model

The model implements the Observer Pattern, to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.

8


Observer
Observers
All these observers will be notified whenever state changes in the model.
Observable
My state has changed!
class Foo { void bar() { doBar(); } }
Model
View
View
Controller
I'd like to register as an observer
View
Any object that's interested in state changes in the model registers with the model as an observer.
The model has no dependencies on viewers or controllers!


Strategy
The user did something
The controller is the strategy for the view – it's the object that knows how to handle the user actions.
The view delegates to the controller to handle the user actions.
Controller
View
We can swap in another behavior for the view by changing the controller.
Controller
The view only worries about presentation, the controller worries about translating user input to actions on the model.


Composite
paint()
Window
Set BPM:
Set
<<
>>
View
The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.

Primitives

What is a primitive: Something that is provided by the language and may not reflect the normal paradigm of this language

Example: Boolean in Java

Implement and operation for boolean

Create abstract class boolean
Create inherited classes true and false with an and operation
The And operation for false returns this (false)
The And operation for true returns the argument boolean passed to the method

Not, Xor, or, …

No if statements needed