Advanced Software Engineering

Ijaz Ahmed

January 30, 2020

Course Outline

- □ Design Patterns (4 weeks)
    - Creational Design Patterns
    - Structural Design Patterns
    - Behavioral Design Patterns
- □ Java Modeling Language JML (4 weeks)
    - Design by Contract
    - Pre and Post Conditions
    - Class Invariants
    - Static Analysis
- □ Advanced Topics (4 weeks)
    - Model Checking
    - Symbolic Execution
    - Delta Debugging

Covered Topics

- ☐ Factory Pattern
- ☐ Singleton Pattern
- ☐ Flyweight Pattern
- ☐ Adapter Pattern
- ☐ Decorator Pattern
- ☐ Facade Pattern
- ☐ MVC
- ☐ Observer Pattern
- ☐ Strategy Pattern

Presentations

- ☐ State Pattern
- ☐ Memento Pattern
- ☐ Multiton Pattern
- ☐ Interpreter Pattern
- ☐ Prototype Pattern
- ☐ Proxy Pattern
- ☐ Command Pattern
- ☐ Iterator Pattern
- ☐ Visitor Pattern
- ☐ Template Method
- ☐ Bridge Pattern

Assignment2

- ☐ Facade Pattern
- ☐ Strategy Pattern

Assignment2

- ☐ Adapter Pattern
- ☐ Decorator Pattern

Assignment3

- ☐ Observer Pattern
- ☐ Flyweight Pattern

Assignment4

- ☐ Factory Pattern
- ☐ Singleton Pattern

Presentations should include

- ☐ Intent, Motivation
- ☐ Application
- ☐ Class/Sequence Diagram
- ☐ Code

Teaching Methodology for This Course

- ☐ A teacher duty is to “explain concepts”
- ☐ A student duty is to explore concepts based on the “explained concepts”
- ☐ Students are also supposed to present and implement some solutions
- ☐ Roughly 3 to 4 classes will be invested on presentations

Design Patterns

- ☐ A solution to a problem that occurs repeatedly in a variety of contexts
- ☐ Each pattern has a name
- ☐ Use of each pattern has consequences

Design Patterns (Co.)

- Generally at a higher level of abstraction.
- Not about designs such as linked lists or hash tables
- Generally descriptions of communicating objects and classes

Different Types of Pattern

- Creational
  - Flyweight
  - Abstract Factory
  - Factory Method
- Structural:
  - Faade
- Behavioral
  - MVC
  - Observer
- Design Patterns book by Erich Gamma, et al., Addison-Wesley, 1994.

Structural Pattern

Pattern: Flyweight

A class that has only one instance for each unique state

Problem of redundant objects

- Existence of redundant objects can bog down system

  many objects have same state

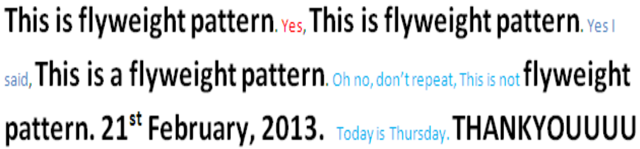
- Example: File objects that represent the same file on disk

  new File("test.txt")
  new File("test.txt")
  new File("test.txt")
  ...
  ...

Flyweight pattern

- an assurance that no more than one instance of a class will have identical state
- achieved by caching identical instances of objects to reduce object construction
- Objects for each character in a document editor
- similar to singleton, but has many instances, one for each unique-state object
- useful for cases when there are many instances of a type but many are the same
- can be used in conjunction with Factory pattern to create a very efficient object-builder
- examples in Java: String, Image / Toolkit, Formatter

character in a document


This is flyweight pattern. Yes, This is flyweight pattern. Yes I said, This is a flyweight pattern. Oh no, don't repeat, This is not flyweight pattern. 21st February, 2013. Today is Thursday. THANKYOUUUU

Flyweight pattern

- Java Strings are flyweighted by the compiler wherever possible
- can be flyweighted at runtime with the intern method

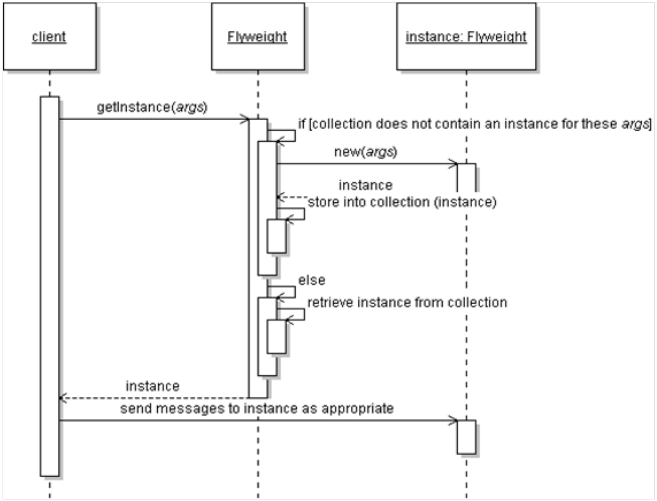
```
public class StringTest {
  public static void main(String[] args) {
    String fly  = "fly", weight  = "weight";
    String fly2 = "fly", weight2 = "weight";

    System.out.println(fly == fly2);          // true
    System.out.println(weight == weight2);  // true

    String distinctString = fly + weight;
    System.out.println(distinctString == "flyweight"); fa

    String flyweight = (fly + weight).intern();
    System.out.println(flyweight == "flyweight");// true
  }
}
```

Implementing a Flyweight

- ☐ flyweighting works best on immutable objects
- ☐ immutable: cannot be changed once constructed

```
public class Flyweighted {
static map or table of instances
private constructor
static method to get an instance
if we have created this type of instance before, get it
otherwise, make the new instance, store and return it
}
```

Flyweight sequence diagram


client
Flyweight
instance: Flyweight
getInstance(*args*)
if [collection does not contain an instance for these *args*]
new(*args*)
instance
store into collection (instance)
else
retrieve instance from collection
instance
send messages to instance as appropriate

Implementing a Flyweight

```
public class Flyweighted {
  Map or table of instances

  private Flyweighted() {}

  public static synchronized Flyweighted
  getInstance(Object key) {
    if (!myInstances.contains(key)) {
      Flyweighted fw = new Flyweighted(key);
      myInstances.put(key, fw);
      return fw;
    } else
      return (Flyweighted)myInstances.get(key);
  }
}
```

A class to be flyweighted

```
public class Point {
  private int x, y;

  public Point(int x, int y) {
    this.x = x;     this.y = y;
  }

  public int getX() { return this.x; }
  public int getY() { return this.y; }

  public String toString() {
    return "(" + this.x + ",␣" + this.y + ")";
  }
}
```

A class that has been flyweighted!

```
public class Point {
  private static Map instances = new HashMap();

  public static Point getInstance(int x, int y) {
    String key = x + ",␣" + y;
    if (instances.containsKey(key))  // re-use existing
      return (Point)instances.get(key);

    Point p = new Point(x, y);
    instances.put(key, p);
    return p;
  }

  private final int x, y;  // immutable

  private Point(int x, int y) {
```

Flyweight: Applicability

- □ An application uses a large number of objects
- □ Storage costs are high because of the sheer quantity of objects
- □ Many Groups of objects may be replaced by relatively few shared objects
- □ The application doesnt depend on object identity

Adapter Pattern

Adapter Pattern

- ☐ Convert the interface of a class into another interface clients expect.
- ☐ Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- ☐ Wrap an existing class with a new interface
- ☐ Also Known As Wrapper

Motivation

- Outlets in the US require a certain kind of plug.
- For example, a plug made in Europe outlet may not be used in USA.
- To use these appliances in USA or vice-versa we may need to purchase an adapter

Motivation (Co.)

- Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
- We can not change the library interface, since we may not have its source code
- Even if we did have the source code, we probably should not change the library for each domain-specific application

Examples

- ☐ Example 1- YYYYMMDD to MM/DD/YYYY or DD/MM/YYYY

Main Participants

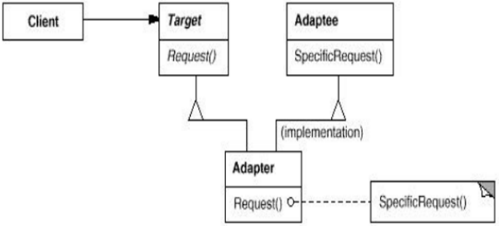
- Adapter
  - adapts the interface Adaptee to the Target interface.
- Adaptee
  - defines an existing interface that needs adapting.

Variations in Adapters

- Class Adapters
  - Use multiple inheritance to compose classes
- Object Adapters
  - Object adapters use a compositional technique to adapt one interface to another.

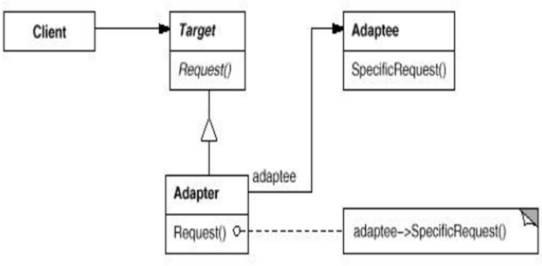
Structure

- A class adapter uses multiple inheritance to adapt one interface to another:

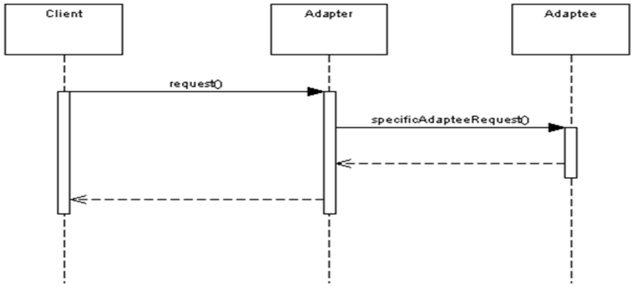


Structure

- An object adapter relies on object composition:



Collaboration



Applicability

- ☐ You want to use an existing class, and its interface does not match the one you need
- ☐ You want to create a reusable class that cooperates with unrelated classes with incompatible interfaces

Implementation

- How much adapting should be done?
  - Simple interface conversion that just changes operation names and order of arguments
  - Totally different set of operations???

Decorator Pattern

Decorator Pattern

- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing to extend flexibility
- Examples
  - Add borders or scrollbars to a GUI component
  - Add headers and footers to an advertisement
  - compressing a file before sending it over the wire

An Application

- Suppose there is a TextView GUI component and you want to add different kinds of borders and/or scrollbars to it
- You can add 3 types of borders
  - Plain, 3D, Fancy
- You can add 3 types of borders
  - Plain, 3D, Fancy
- and , 1, or 2 two scrollbars
  - Horizontal and Vertical
- An inheritance solution requires 15 classes for one view

So many classes

- ☐ TextView_Plain
- ☐ TextView_Fancy
- ☐ TextView_3D
- ☐ TextView_Horizontal
- ☐ TextView_Vertical
- ☐ TextView_Horizontal_Vertical
- ☐ TextView_Plain_Horizontal
- ☐ TextView_Plain_Vertical
- ☐ TextView_Plain_Horizontal_Vertical
- ☐ TextView_3D_Horizontal
- ☐ TextView_3D_Vertical
- ☐ TextView_3D_Horizontal_Vertical
- ☐ TextView_Fancy_Horizontal
- ☐ TextView_Fancy_Vertical
- ☐ TextView_Fancy_Horizontal_Vertical

Disadvantages

- ☐ Inheritance solution has an explosion of classes
- ☐ With another type of border added, many more classes would be needed with this design?
- ☐ Use the Decorator Pattern instead

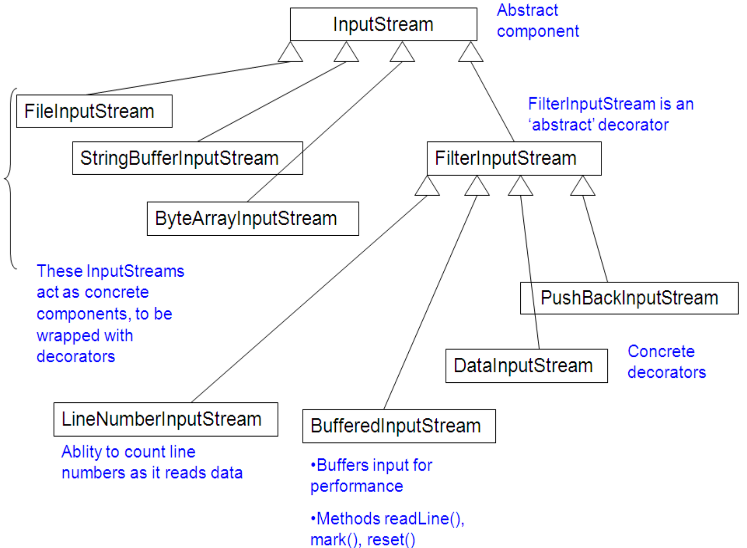
Motivation

- The more more flexible containment approach encloses the component in another object that adds the border
- The enclosing object is called the decorator
- The decorator conforms to the interface of the component so its presence is transparent to clients
- The decorator forwards requests to the component and may perform additional actions before or after any forwarding

Motivation

- InputStreamReader(InputStream in)
  - Bridge from byte streams to character streams: It reads bytes and translates them into characters using the specified character encoding.
- BufferedReader
  - Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

Decorator


InputStream
Abstract component
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
FilterInputStream is an 'abstract' decorator
These InputStreams act as concrete components, to be wrapped with decorators
PushBackInputStream
DataInputStream
Concrete decorators
LineNumberInputStream
Ablity to count line numbers as it reads data
BufferedInputStream
•Buffers input for performance
•Methods readLine(), mark(), reset()

Decorating FilterInputStream

```
public class LowerCaseInputStream extends FilterInputStre

public LowerCaseInputStream(InputStream in) {
                super(in);
}

public int read() throws IOException {
                int c = super.read();
                return (c == -1 ? c : Character.toLowerC
}
public int read(byte[] b, int offset, int len) throws IO
                int result = super.read(b, offset, len);
                for (int i = offset; i < offset+result;
                        b[i] = (byte)Character.toLowerCa
                }
                return result;
}
```

Decorating FilterInputStream

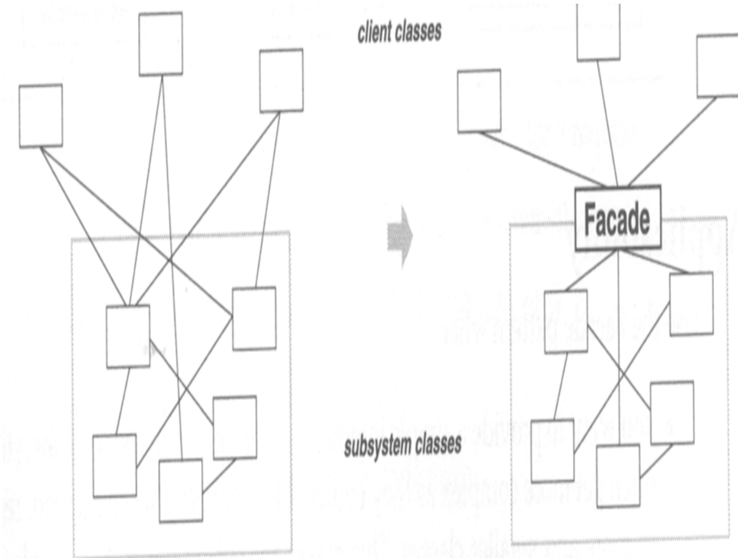
```
public class InputTest {
public static void main(String[] args) throws IOExceptio
int c;
 try {
        InputStream in =
            new LowerCaseInputStream(
                new BufferedInputStream(
                new FileInputStream("test.txt")));

        while((c = in.read()) >= 0) {
        System.out.print((char)c);
}
in.close();
}
catch (IOException e) {
        e.printStackTrace();
        }
}
```

Facade Pattern

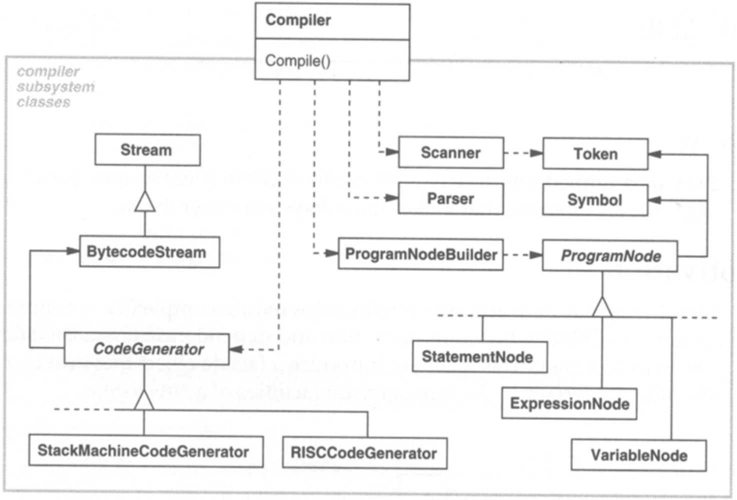
Facade Pattern

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem.
  - Faade defines a higher-level interface that makes the subsystem easier to use
- Motivation
  - Simplifying system architecture by unifying related but different interfaces via a Faade object that shield this complexity from clients

Faade Motivation



Faade Motivation



Faade Applicability

- ☐ Contrary to other patterns which decompose systems into smaller classes, Faade combines interfaces together to unified same one.
- ☐ Separate subsystems from clients via yet another unified interface to them. Levels architecture of a system, using Faade to separate the different subsystem layers of the application

Faade Participants

- Faade (Compiler)
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects
- Subsystem classes (Scanner, Parser, ProgramNode, etc.)
  - Implement subsystem functionality
  - Handle work assignment by the Faade object
  - Have no knowledge of the Faade I.e., no reference upward

Faade Collaboration

- Clients
  - Sending requests to the Faade, which forwards them appropriately to the subsystem components
- Separation
  - Clients do not need to know, or ever use the subsystem components directly

Faade Consequences

- Shielding Clients
  - Reduces the number of objects clients need to deal with
- Promotes weak coupling
  - Between subsystems and clients
  - components in the subsystem may be strongly coupled.
  - Help layer the system (also prevents circular dependencies)
- But permits direct use
  - In case individual components offer meaningful service to clients the Faade mediates, but does not block access.

Facade Implementation

```
class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}
```

Facade Implementation

```
class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
```

Facade Implementation

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

Adaptor, Decorator and Facade

Pattern	**Intent**
Adapter	Converts one interface to another so that it matches what the client is expecting
Decorator	Dynamically adds responsibility to the interface by wrapping the original code
Facade	Provides a simplified interface

Structural Pattern

- ☐ Class Structural patterns concern the aggregation of classes to form largest structures
- ☐ Object Structural pattern concern the aggregation of objects to form largest structures
- ☐ Ease the design by identifying a simple way to realize relationships between entities.

Memento Pattern

Intent

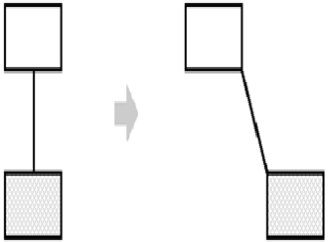
- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later

Motivation

- ☐ record the internal state of an object
- ☐ let users back out of tentative operations or recover from errors

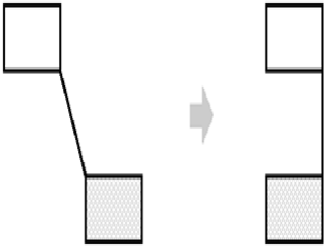
Motivation

- A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them



Motivation

- Supporting undo in this application isn't as easy as it may seem

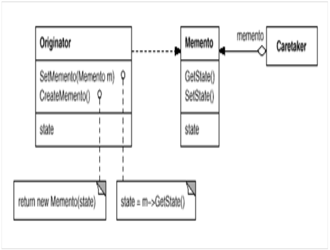


Applicability

- ☐ Snapshot
- ☐ Undo
- ☐ Redo
- ☐ History
- ☐ Saving and Loading

Participants

- ☐ Use the Memento pattern when
- ☐ It may be restored to that state later
- ☐ A direct interface would expose implementation



Memento

- ☐ stores internal state of the Originator object
- ☐ protects against access by objects other than the originator
- ☐ Two interfaces
- ☐ narrow interface
- ☐ wide interface

Originator

- ☐ creates a memento
- ☐ uses the memento

Caretaker

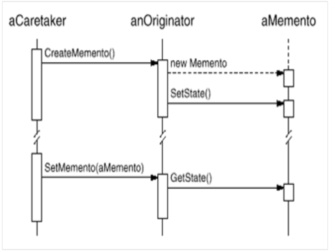
- ☐ is responsible for the mementos safekeeping.
- ☐ never operates on or examines the contents of a memento.

Memento

- stores internal state of the Originator object
- protects against access by objects other than the originator
- Two interfaces

Collaborations

- Mementos are passive. Only the originator that created a memento will assign or retrieve its state



Originator

```
class Originator {
    private String state;
    // The class could also contain additional data that is
    // state saved in the memento..

    public void set(String state) {
        System.out.println("Originator:␣Setting␣state␣to␣" +
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator:␣Saving␣to␣Memento.");
        return new Memento(state);
    }

    public void restoreFromMemento(Memento memento) {
        state = memento.getSavedState();
        System.out.println("Originator:␣State␣after␣restoring
    }
```

Memento

```
public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        public String getSavedState() {
            return state;
        }
    }
```

Caretaker

```
class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new ArrayList<

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

Output

```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```

Creational Pattern

- ☐ Abstract the instantiation process
- ☐ Make a system independent to its realization
- ☐ Class Creational use inheritance to vary the instantiated classes
- ☐ Object Creational delegate instantiation to an another object

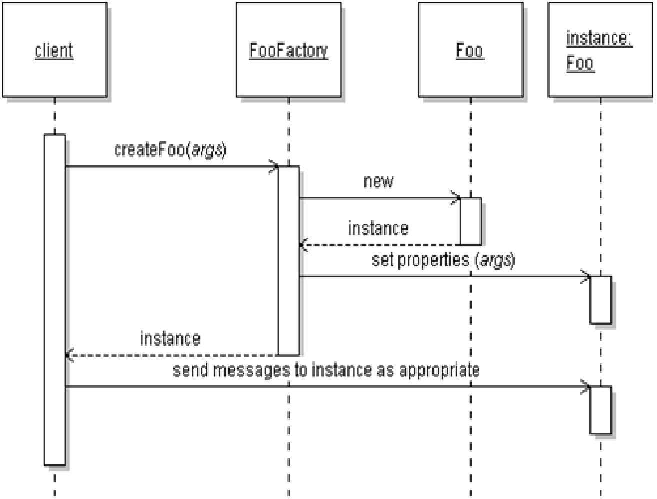
Factory pattern

- Factory: a class whose sole job is to easily create and return instances of other classes
- creational pattern; makes it easier to construct complex objects
- instead of calling a constructor, use a static method in a "factory" class to set up the object
- saves lines and complexity to quickly construct / initialize objects
- examples in Java: borders (BorderFactory), key strokes (KeyStroke), network connections (SocketFactory)

Factory implementation details

- ☐ the factory itself should not be instantiated
- ☐ make constructor private
- ☐ factory only uses static methods to construct components
- ☐ factory should offer as simple an interface to client code as possible
- ☐ don't demand lots of arguments; possibly overload factory methods to handle special cases that need more arguments
- ☐ factories are often designed for reuse on a later project or for general use throughout your system

Factory sequence diagram



Examples:

- The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.
- There can be many printers in a system but there should only be one printer spooler.
- There should be only one instance of a WindowManager.
- There should be only one instance of a filesystem.

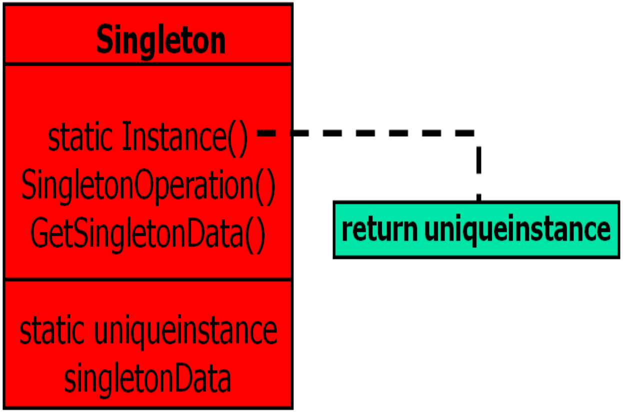
Singleton Pattern

- ☐ How do we ensure that a class has only one instance and that the instance is easily accessible?
- ☐ A global variable makes an object accessible, but does not keep you from instantiating multiple objects.
- ☐ A better solution is to make the class itself responsible for keeping track of its sole instance. The class ensures that no other instance can be created (by intercepting requests to create new objects) and it provides a way to access the instance.

Use Singleton Pattern

- ☐ There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- ☐ When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

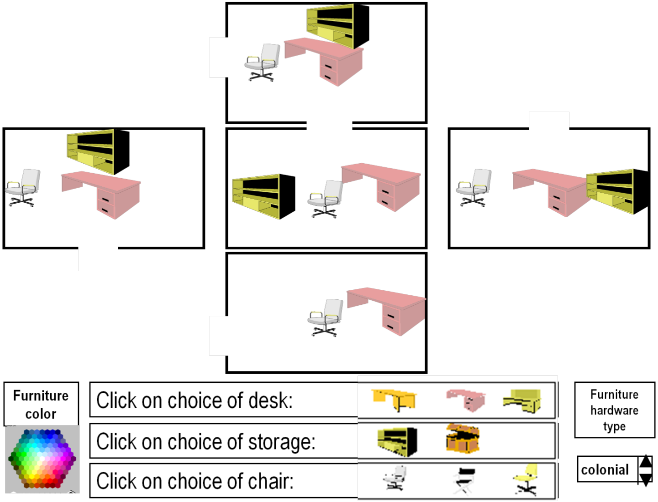
Singleton Pattern



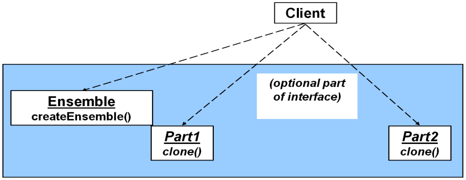
Prototype Pattern

- ☐ Create a set of almost identical objects whose type is determined at runtime
- ☐ Assume that a prototype instance is known; clone it whenever a new instance is needed.

Motivation


Furniture color
Click on choice of desk:
Click on choice of storage:
Click on choice of chair:
Furniture hardware type
colonial

Motivation

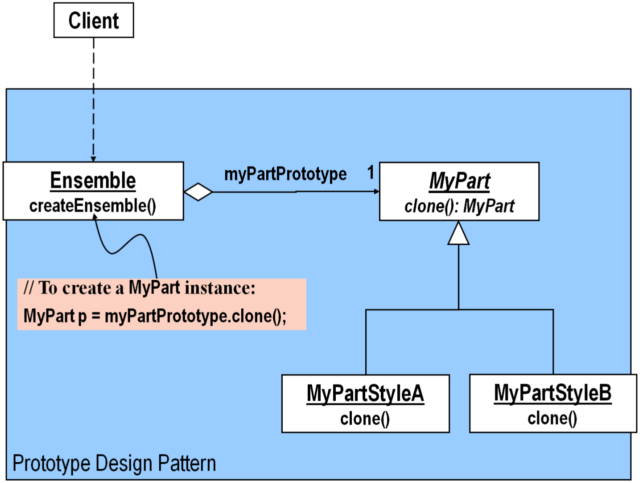


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Code Example

```
OfficeSuite myOfficeSuite =
    OfficeSuite.createOfficeSuite( myDesk,
                myChair, myStorage);

myGUI.add(myOfficeSuite);
myOfficeSuite.setBackground( pink );
```

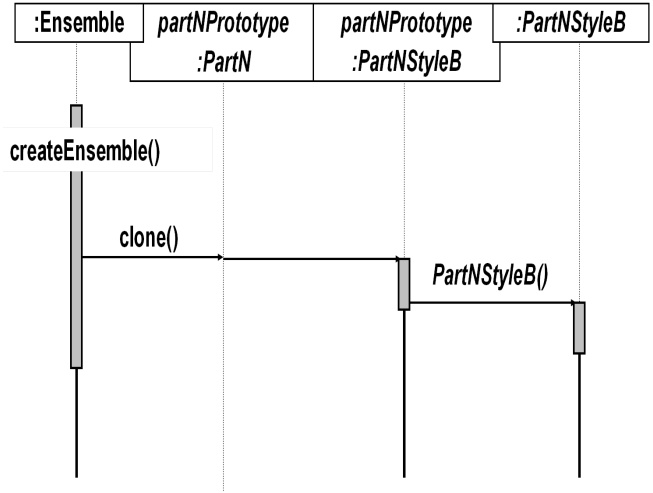
The Prototype Idea



```
// To create a MyPart instance:
MyPart p = myPartPrototype.clone();
```

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Sequence Diagram for Prototype



Code Example

```
Ensemble EnsembleA= Ensemble.createEnsemble(a,b,c);

Ensemble EnsembleB = Ensemble.createEnsemble(a,b,c);

MyPart anotherMyPart = MyPartPrototype.clone();
MyPart yetAnotherMyPart = MyPartPrototype.clone();
```

Factory and Prototype Pattern

- Prototype allows the client to select any chair style, any desk style, and any cabinet style
- This is all done separately rather than have to select an overall office style
- Nevertheless, the client wants to keep a single style of chair and a single style of desk throughout the office suite

Copying objects

- In other languages (common in C++), to enable clients to easily make copies of an object, you can supply a copy constructor
- Java has some copy constructors but also has a different way

```
Point p1 = new Point(-3, 5);
Point p2 = new Point(p1);      // make p2 a copy of p1


// in Point.java
public Point(Point blueprint) {   // copy constructor
    this.x = blueprint.x;
    this.y = blueprint.y;
```

Copying objects

```
protected Object clone() throws CloneNotSupportedExcepti
x.clone() != x
x.clone().equals(x)
x.clone().getClass() == x.getClass()
```

- ☐ The Object class's clone method makes a "shallow copy" of the object, but by convention, the object returned by this method should be independent of this object (which is being cloned).

Copying objects

```
protected Object clone() throws CloneNotSupportedExcepti
x.clone() != x
x.clone().equals(x)
x.clone().getClass() == x.getClass()
```

- The Object class's clone method makes a "shallow copy" of the object, but by convention, the object returned by this method should be independent of this object (which is being cloned).

Copying objects

- protected Object **clone**() throws CloneNotSupportedException
- **protected**: Visible only to the class itself, its subclasses, and any other classes in the same package.
- In other words, for most classes you are not allowed to call clone .
- If you want to enable cloning, you must override clone .
- You should make it public so clients can call it.
- You can also change the return type to your class's type. (good)
- You can also not throw the exception. (good)
- You must also make your class implement the Cloneable interface to signify that it is allowed to be cloned.

Flawed clone method 1

```
public class Point implements Cloneable {
            private int x, y;
            ...
            public Point clone() {
          Point copy = new Point(this.x, this.y);
          return copy;
            }
        }
```

- What's wrong with the above method?

The flaw

```
public class Point3D extends Point {
                private int z;
                ...
        }
```

- The above Point3D class's clone method produces a Point!
- This is undesirable and unexpected behavior.
- The only way to ensure that the clone will have exactly the same type as the original object (even in the presence of inheritance) is to call the clone method from class Object with super.clone()

Proper clone method

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        try {
            Point copy = (Point) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            // this will never happen
            return null;
        }
    }
}
```

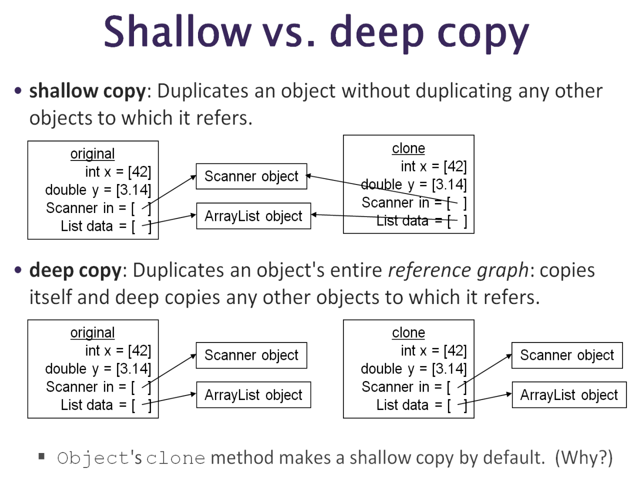
- To call Object's clone method, you must use try/catch.
- But if you implement Cloneable, the exception will not be thrown

Flawed clone method 2

```
        public class BankAccount implements Cloneable {
    private String name;
    private List<String> transactions;
    ...
    public BankAccount clone() {
        try {
            BankAccount copy = (BankAccount) super.clone
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;    // won't ever happen
        }
    }
}
```

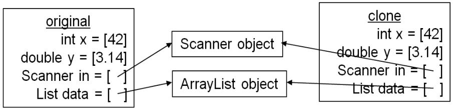
- ☐ What's wrong with the above method?

Deep versus Shallow Clone

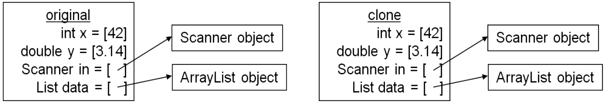


Shallow vs. deep copy

- **shallow copy**: Duplicates an object without duplicating any other objects to which it refers.



- **deep copy**: Duplicates an object's entire *reference graph*: copies itself and deep copies any other objects to which it refers.



  - `Object`'s `clone` method makes a shallow copy by default. (Why?)

Proper clone method 2

```
      public class BankAccount implements Cloneable {
   private String name;
   private List<String> transactions;
   ...
   public BankAccount clone() {
      try {                    // deep copy
         BankAccount copy = (BankAccount) super.clone();
         copy.transactions = new ArrayList<String>(trans
         return copy;
      } catch (CloneNotSupportedException e) {
         return null;
      }
   }
}
```

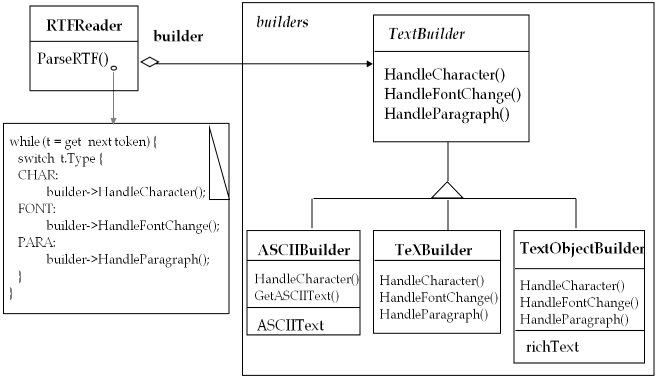
- Copying the list of transactions (and any other modifiable reference fields) produces a deep copy that is independent of the original

Builder Pattern

Intent / Applicability

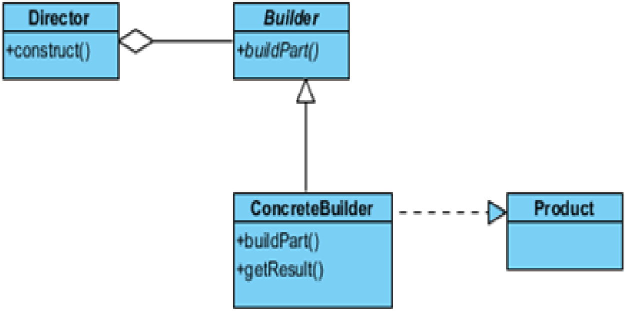
- ☐ Separate the construction of a complex object from its representation
- ☐ Same construction process can create different representations
- ☐ Algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
- ☐ the construction process must allow different representations for the object that is constructed

RTF Reader Example

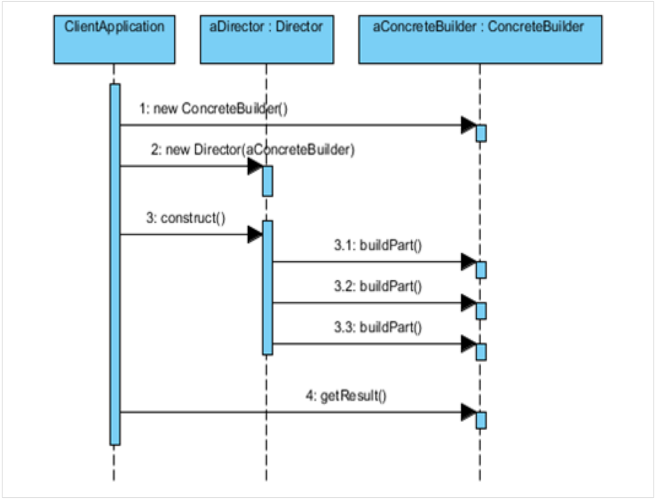


```
while (t = get  next token) {
  switch  t.Type {
  CHAR:
        builder->HandleCharacter();
  FONT:
        builder->HandleFontChange();
  PARA:
        builder->HandleParagraph();
  }
}
```

UML


Director
+construct()
Builder
+buildPart()
ConcreteBuilder
+buildPart()
+getResult()
Product

Collaborators



Example: building different types of airplanes

- AerospaceEngineer (director)
- AirplaneBuilder (abstract builder)
- Airplane (product)
- Sample concrete builders:
  - CropDuster
  - FighterJet
  - Glider
  - Airliner

Director

```
public class AerospaceEngineer {

        private AirplaneBuilder airplaneBuilder;

        public void setAirplaneBuilder(AirplaneBuilder a
                airplaneBuilder = ab;
        }

        public Airplane getAirplane() {
                return airplaneBuilder.getAirplane();
        }

        public void constructAirplane() {
                airplaneBuilder.createNewAirplane();
                airplaneBuilder.buildWings();
                airplaneBuilder.buildPowerplant();
                airplaneBuilder.buildAvionics();
```

AbstractBuilder

```
public abstract class AirplaneBuilder {

        protected Airplane airplane;
        protected String customer;
        protected String type;

        public Airplane getAirplane() {
                return airplane;
        }
        public void createNewAirplane() {
                airplane = new Airplane(customer, type);
        }
        public abstract void buildWings();
        public abstract void buildPowerplant();
        public abstract void buildAvionics();
        public abstract void buildSeats();
}
```

Product

```
public class Airplane {
        private String type;
        private float wingspan;
        private String powerplant;
        private int crewSeats;
        private int passengerSeats;
        private String avionics;
        private String customer;

        Airplane (String customer, String type){
                this.customer = customer;
                this.type = type;
        }

        public void setWingspan(float wingspan) {
                this.wingspan = wingspan;
        }
```

Product (Continued)

```
    public void setPowerplant(String powerplant) {
                    this.powerplant = powerplant;
            }
            public void setAvionics(String avionics) {
                    this.avionics = avionics;
            }
            public void setNumberSeats(int crewSeats, int pa
                    this.crewSeats = crewSeats;
                    this.passengerSeats = passengerSeats;
            }
    public String getCustomer() {
                    return customer;
            }
    public String getType() {
                    return type;
            }
}
```

ConcreteBuilder 1

```
public class CropDuster extends AirplaneBuilder {
        CropDuster (String customer){
                super.customer = customer;
                super.type = "Crop␣Duster␣v3.4";
        }
        public void buildWings() {
                airplane.setWingspan(9f);
        }
        public void buildPowerplant() {
                airplane.setPowerplant("single␣piston");
        }
    public void buildAvionics() {}

        public void buildSeats() {
                airplane.setNumberSeats(1,1);
        }
}
```

ConcreteBuilder 2

```
public class FighterJet extends AirplaneBuilder {

        FighterJet (String customer){
                super.customer = customer;
                super.type = "F-35␣Lightning␣II";
        }
    public void buildWings() {
                airplane.setWingspan(35.0f);
        }
    public void buildPowerplant() {
                airplane.setPowerplant("dual␣thrust␣vect
        }

        public void buildAvionics() {
                airplane.setAvionics("military");
        }
    public void buildSeats() {
```

ConcreteBuilder 3

```
public class Glider extends AirplaneBuilder {

        Glider (String customer){
                super.customer = customer;
                super.type = "Glider␣v9.0";
        }

        public void buildWings() {
                airplane.setWingspan(57.1f);
        }

        public void buildPowerplant() {}
    public void buildAvionics() {}
    public void buildSeats() {
                airplane.setNumberSeats(1,0);
        }
}
```

ConcreteBuilder 4

```
public class Airliner extends AirplaneBuilder {

        Airliner (String customer){
                super.customer = customer;
                super.type = "787␣Dreamliner";
        }

        public void buildWings() {
                airplane.setWingspan(197f);
        }

        public void buildPowerplant() {
                airplane.setPowerplant("dual␣turbofan");
        }

        public void buildAvionics() {
                airplane.setAvionics("commercial");
```

Client Application

```
public class BuilderExample {
        public static void main(String[] args) {
                // instantiate the director (hire the en
                AerospaceEngineer aero = new AerospaceEn

                // instantiate each concrete builder (ta
                AirplaneBuilder crop = new CropDuster("F
                AirplaneBuilder fighter = new FighterJet
                AirplaneBuilder glider = new Glider("Tim
                AirplaneBuilder airliner = new Airliner(

                // build a CropDuster
                aero.setAirplaneBuilder(crop);
                aero.constructAirplane();
                Airplane completedCropDuster = aero.getA
                System.out.println(completedCropDuster.g
                                "␣is␣completed␣and␣ready
                                completedCropDuster.getC
```

Builder: Advantages / Disadvantages

- Advantages
  - Allows you to vary a products internal representation
  - Encapsulates code for construction and representation
  - Provides control over steps of construction process
- Disadvantages
  - Requires creating a separate ConcreteBuilder for each different type of Product

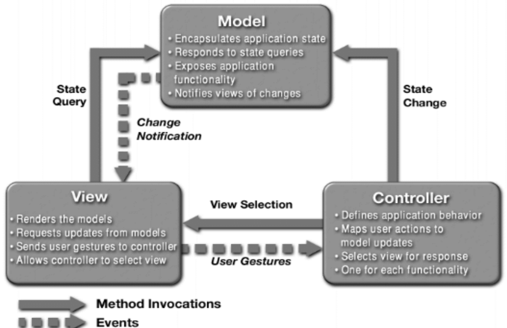
Behavioral Pattern

- Concern with algorithms and assignment of responsibilities between objects
- Describe the patterns of communication between classes or objects
- Behavioral class pattern use inheritance to distribute behavior between classes
- Behavioral object pattern use object composition to distribute behavior between classes

The MVC Architectural Pattern

The MVC Architectural Pattern

- MVC was first introduced by Trygve Reenskaug at the Xerox Palo Alto Research Center in 1979.
- Part of the basic of the Smalltalk programming environment.
- Widely used for many object-oriented designs involving user interaction.
- A three-tier architectural model:


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Model

- Manages the behavior and data of the application domain,
- Responds to requests for information about its state (usually from the view),
- Responds to instructions to change state (usually from the controller).
- In event-driven systems, the model notifies observers (usually views) when the information changes so that they can react.
- In enterprise software, a model often serves as a software approximation of a real-world process.
- In a game, the model is represented by the classes defining the game entities, which are embedding their own state and actions.

View

- Renders the model into a form suitable for interaction, typically a user interface element.
- Multiple views can exist for a single model for different purposes.
- The view renders the contents of a portion of the models data.
- If the model data changes, the view must update its presentation as needed. This can be achieved by using:
- a push model
  - in which the view registers itself with the model for change notifications
- a pull model
  - in which the view is responsible for calling the model when it needs to retrieve the most current data.

Controller

- Receives user input and initiates a response by making calls on appropriate model objects.
- Accepts input from the user and instructs the model to perform actions based on that input.
- The controller translates the user's interactions with the view it is associated with, into actions that the model will perform.
- A controller may also spawn new views upon user demand

Interactions between Model, View and Controller

- The view registers as an observer on the model. Any changes to the underlying data of the model immediately result in a broadcast change notification, which all associated views receives (in the push back model). Note that the model is not aware of the view or the controller – it simply broadcasts change notifications to all interested observers.
- The controller is bound to the view and can react to any user interaction provided by this view. This means that any user actions that are performed on the view will invoke a method in the controller class.
- The controller is given a reference to the underlying model

Interactions between Model, View and Controller

- Once a user interacts with the view, the following actions occur:
- The view recognizes that a GUI action – for example, pushing a button or dragging a scroll bar – has occurred. In the listener method, the view calls the appropriate method on the controller.
- The controller translates this signal into an appropriate action in the model, which will in turn possibly be updated in a way appropriate to the user's action.
- If the model has been altered, it notifies interested observers, such as the view, of the change.

Observer Pattern

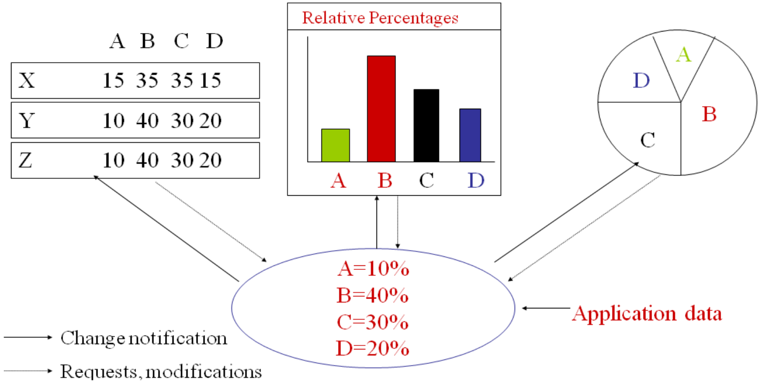
Motivation

- The cases when certain objects need to be informed about the changes occurred in other objects are frequent.
- The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- This pattern is a cornerstone of the Model-View-Controller architectural design, where the Model implements the mechanics of the program, and the Views are implemented as Observers.

Application

- ☐ Subscribers of mobile-communication provider services
- ☐ Subscribers of an email-service
- ☐ Etc

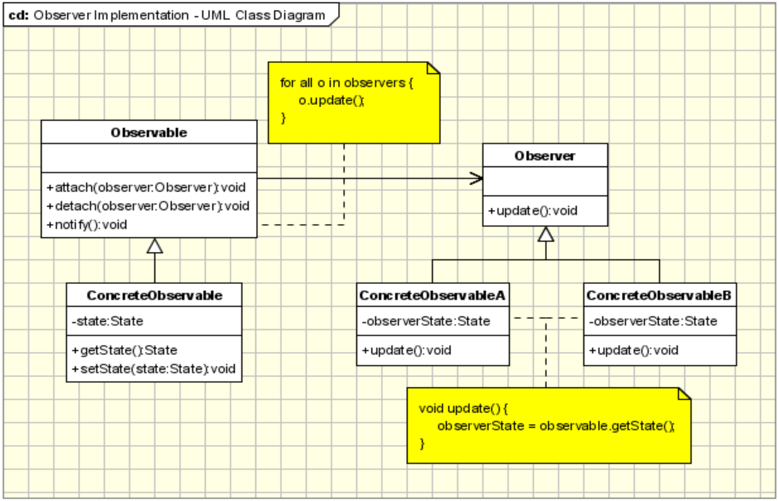
Application (Co.)



The Participants Classes

- Observable
  - interface or abstract class defining the operations for attaching and de-attaching observers to the client. known as Subject.
- ConcreteObservable - concrete Observable class.
  - It maintain the state of the observed object and when a change in its state occurs it notifies the attached Observers.
- Observer
  - interface or abstract class defining the operations to be used to notify the Observer object.
- ConcreteObserverA, ConcreteObserverB -
  - concrete Observer implementations.

Definition


cd: Observer Implementation - UML Class Diagram
for all o in observers {
o.update();
}
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() {
observerState = observable.getState();
}

Implementation

- The client class instantiates the ConcreteObservable subject object.
- Then it instantiate concrete observers and attaches the concrete observers to subject.
- Each time the state of the subject is changing, it notifies all the attached Observers using the methods defined in the Observer class.
- When a new Observer is added to the application, all we need to do is to instantiate it in the client class and to add attach it to the Observable object.

Example

```
public class Subject {

private:List<Observer*> *_observers;

public void Attach(Observer* o){
        _observers->Insert(o);
}
public void Detach(Observer* o){
        _observers->remove(o);
}
public void Notify(){
// assign i the address of the _observers
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);}
    }
}
```

Example

```
public class ClockTimer extends Subject {

    int GetHour(){return hour};
    int GetMinute(){return minute};
    int GetSecond(){return second};

public void Tick(){
           Notify();
   };

  private:
    int hour;
    int minute;
    int second;
};
```

Example

```
//Observer Class
public interface Observer {
  public void Update(Subject* theChangeSubject);
}
```

Example

```
public class DigitalClock implements Observer {
    ClockTimer _subject;

    DigitalClock(ClockTimer s){
      _subject = s; _subject->Attach(this);
      }

    void Update(Subject theChangedSubject){
      if(theChangedSubject == _subject)
        draw();
     }
   public void draw(){
     int hour   = _subject->GetHour();
     int minute = _subject->GetMinute();
     int second = _subject->GetSecond();
     // draw operation
     };
}
```

Example

```
int main(void)
{

  ClockTimer timer = new ClockTimer();       //Subject
  DigitalClock digitalClock = new DigitalClock(timer);
   //Observer
  timer->Tick(); // Subject changes
  return 0;
}
```

Chain of Responsibility Pattern

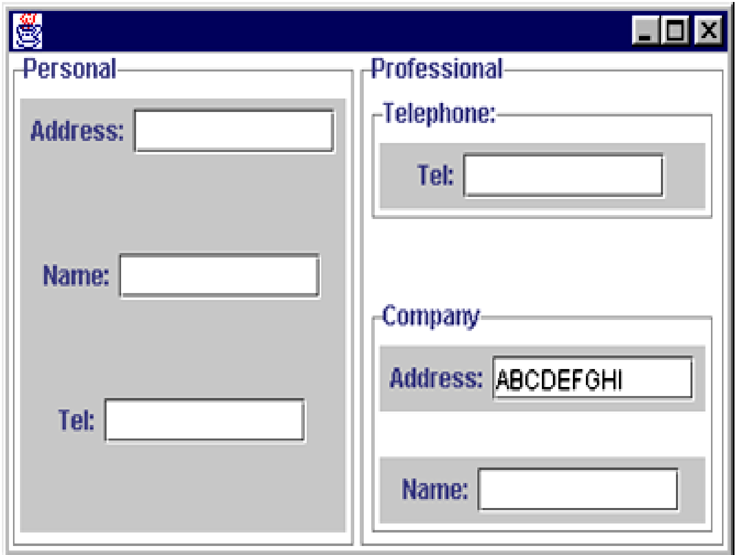
Intent

- ☐ Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- ☐ Chain the receiving objects and pass the request along the chain until an object handles it

Motivation

- ☐ Consider a context-sensitive help system for a GUI
- ☐ The object that ultimately provides the help isn't known explicitly to the object (e.g., a button) that initiates the help request
- ☐ So use a chain of objects to decouple the senders from the receivers. The request gets passed along the chain until one of the objects handles it.
- ☐ Each object on the chain shares a common interface for handling requests and for accessing its successor on the chain

GUI For Customer Information Application



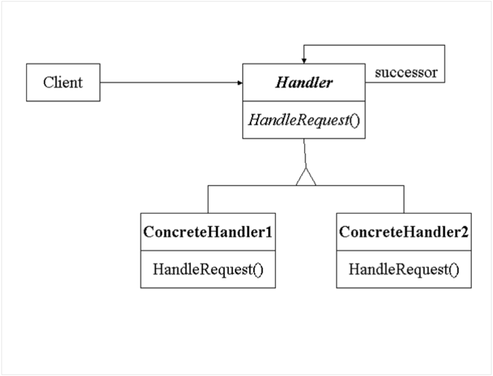
Applicability

- When more than one object may handle a request and the actual handler is not know in advance
- When requests follow a "handle or forward" model - that is, some requests can be handled where they are generated while others must be forwarded to another object to be handled

Consequences

- Reduced coupling between the sender of a request and the receiver - the sender and receiver have no explicit knowledge of each other
- Receipt is not guaranteed - a request could fall off the end of the chain without being handled
- The chain of handlers can be modified dynamically

Class Structure



Participants Classes

- Handler defines interface for handling requests. Can also implement successor link
- ConcreteHandler handles requests it is responsible for; otherwise forwards requests to successor.
- Client initiates request to a ConcreteHandler in the chain.

Abstract Window Toolkit

- Java 1.0 AWT (Abstract Window Toolkit) event handler
- AWT package- Contains all of the classes for creating user interfaces and for painting graphics and images

```
public boolean action(Event event, Object obj) {
 if (event.target == test_button)
 doTestButtonAction();
 else if (event.target == exit_button)
 doExitButtonAction();
 else
 return super.action(event,obj);
 return true; // Return true to indicate the event has b
 // handled and should not be propagated further.
 }
```

Implementation

```
public interface Handler {
 public void handleRequest();
 }
```

Implementation

```
public class ConcreteHandler implements Handler {
 private Handler successor;
 public ConcreteHandler(Handler successor) {
 this.successor = successor;
 }
 public void handleRequest(String request) {
 if (request.equals("Help")) {
 // We handle help ourselves, so help code is here.
 }
 else
 // Pass it on!
 successor.handle(request);
 }
 }
```

Design by Contract

- Design by Contract and the language that implements the Design by Contract principles (called Eiffel) was developed in Santa Barbara by Bertrand Meyer (he was a UCSB professor at the time, now he is at ETH)
- Bertrand Meyer won the 2006 ACM Software System Award for the Eiffel programming language!
  - Award citation: " *For designing and developing the Eiffel programming language, method and environment, embodying the Design by Contract approach to software development and other features that facilitate the construction of reliable, extendible and efficient software*."
- The company which supports the Eiffel language is located in Santa Barbara:
  - Eiffel Software (http://www.eiffel.com)
- The material in the following slides is mostly from the following paper:
  - "Applying Design by Contract," B. Meyer, IEEE Computer,

Dependability and Object-Orientation

- An important aspect of object oriented design is reuse
  - For reusable components correctness is crucial since an error in a module can effect every other module that uses it
- Main goal of object oriented design and programming is to improve the quality of software
  - The most important quality of software is its dependability
- Design by contract presents a set of principles to produce dependable and robust object oriented software
  - Basic design by contract principles can be used in any object oriented programming language

What is a Contract?

- There are two parties:
  - Client which requests a service
  - Supplier which supplies the service
- Contract is the agreement between the client and the supplier
- Two major characteristics of a contract
  - Each party expects some *benefits* from the contract and is prepared to incur some *obligations* to obtain them
  - These benefits and obligations are documented in a contract document
- Benefit of the client is the obligation of the supplier, and vice versa.

What is a Contract?

- As an example let's think about the contract between a tenant and a landlord

What is a Contract?

- A contract document between a client and a supplier protects both sides
  - It protects the client by specifying how much should be done to get the benefit. The client is entitled to receive a certain result.
  - It protects the supplier by specifying how little is acceptable. The supplier must not be liable for failing to carry out tasks outside of the specified scope.
- If a party fulfills its obligations it is entitled to its benefits
  - *No Hidden Clauses Rule*: no requirement other than the obligations written in the contract can be imposed on a party to obtain the benefits

How Do Contracts Relate to Software Design?

- You are not in law school, so what are we talking about?
- Here is the basic idea
  - One can think of pre and post conditions of a procedure as obligations and benefits of a contract between the client (the caller) and the supplier (the called procedure)
- Design by contract promotes using pre and post-conditions (written as assertions) as a part of module design
- Eiffel is an object oriented programming language that supports design by contract
  - In Eiffel the pre and post-conditions are written using require and ensure constructs, respectively

Contracts

- The pre and postconditions are assertions, i.e., they are expressions which evaluate to true or false
  - The precondition expresses the requirements that any call must satisfy
  - The postcondition expresses the properties that are ensured at the end of the procedure execution
- If there is no precondition or postcondition, then the precondition or postcondition is assumed to be true (which is equivalent to saying there is no pre or postcondition)

Assertion Violations

- What happens if a precondition or a postcondition fails (i.e., evaluates to false)
  - The assertions can be checked (i.e., monitored) dynamically at run-time to debug the software
  - A **precondition violation** would indicate a bug at the **caller**
  - A **postcondition violation** would indicate a bug at the **callee**
- Our goal is to prevent assertion violations from happening
  - The pre and postconditions are not supposed to fail if the software is correct
    - hence, they differ from exceptions and exception handling
  - By writing the contracts explicitly, we are trying to avoid contract violations, (i.e, failed pre and postconditions)

Defensive Programming vs. Design by Contract

- Defensive programming is an approach that promotes putting checks in every module to detect unexpected situations
- This results in redundant checks (for example, both caller and callee may check the same condition)
  - A lot of checks makes the software more complex and harder to maintain
- In Design by Contract the responsibility assignment is clear and it is part of the module interface
  - prevents redundant checks
  - easier to maintain
  - provides a (partial) specification of functionality

Design by Contract in Eiffel

Design by Contract in Eiffel

The put_child Contract

- The put_child contract in English would be something like the table below.
  - Eiffel language enables the software developer to write this contract formally using require and ensure constructs

Class Invariants

- A class invariant is an assertion that holds for all instances (objects) of the class
  - A class invariant must be satisfied after creation of every instance of the class
  - The invariant must be preserved by every method of the class, i.e., if we assume that the invariant holds at the method entry it should hold at the method exit
  - We can think of the class invariant as conjunction added to the precondition and postcondition of each method in the class
- For example, a class invariant for a binary tree could be (in Eiffel notation)

Design by Contract and Inheritance

- Inheritance enables declaration of subclasses which can redeclare some of the methods of the parent class, or provide an implementation for the abstract methods of the parent class
- Polymorphism and dynamic binding combined with inheritance are powerful programming tools provided by object oriented languages
  - How can the Design by Contract can be extended to handle these concepts?

Inheritance: Preconditions

- If the precondition of the ClassB.someMethod is stronger than the precondition of the ClassA.someMethod, then this is not fair to the Client
- The code for ClassB may have been written after Client was written, so Client has no way of knowing its contractual requirements for ClassB

Inheritance: Postconditions

- If the postcondition of the ClassB.someMethod is weaker than the postcondition of the ClassA.someMethod, then this is not fair to the Client
- Since Client may not have known about ClassB, it could have relied on the stronger guarantees provided by the ClassA.someMethod

Inheritance: Invariants

- If the class invariant for the ClassB is weaker than the class invariant for the ClassA, then this is not fair to the Client
- Since Client may not have known about ClassB, it could have relied on the stronger guarantees provided by the ClassA

Behavioral Subtyping

- ☐ These inheritance rules in design-by-contract is related to the concept of *behavioral subtyping*
  - Given a program that has a type T, and a type S where S is a subtype of T, if you change the type of objects with type T in the program to the type S, then the behavior of the program should not change
- ☐ This is not enforced in object-oriented programming languages
  - In general it would be undecidable to check if a program conforms to behavioral subtyping
- ☐ The inheritance rules in design-by-contract ensure that the contracts follow the behavioral subtyping principle

Inheritance in Eiffel

- Eiffel enforces the following
  - the precondition of a derived method to be weaker
  - the postcondition of a derived method to be stronger
- In Eiffel when a method overwrites another method the new declared precondition is combined with previous precondition using disjunction
- When a method overwrites another method the new declared postcondition is combined with previous postcondition using conjunction
- Also, the invariants of the parent class are passed to the derived classes
  - invariants are combined using conjunction

Dynamic Design-by-Contract Monitoring

- Enforce contracts at run-time
- A contract
  - Preconditions of modules
    - What conditions the module requests from the clients
  - Postconditions of modules
    - What guarantees the module gives to clients
  - Invariants of the objects
- Precondition violation, the client is to blame
  - Generate an error message blaming the client (caller)
- Postcondition violation, the server is to blame
  - Generate an error message blaming the server (callee)
- Eiffel compiler supports dynamic design-by-contract monitoring. You can run the program with design-by-contract monitoring on, and it will report any contract violations are runtime

Design-by-Contract Java

- There are dynamic design-by-contract monitoring tools for Java
  - preconditions, postconditions and class invariants are written as Java predicates (Java methods with no side effects, that return a boolean result)
  - Tool: JContractor (http://jcontractor.sourceforge.net/) developed by Murat Karaorman from UCSB
- Given the precondition, postcondition and class invariant methods, dynamic design-by-contract monitoring tools instrument the program to track contract violations and report any contract violations at runtime
- A different approach to writing design-by-contract specifications is to use an annotation language
  - An annotation language is a language which has a formal syntax and semantics but written as a part of the comments in a program
    - So it does not interfere with the program execution and can

Java Modeling Language (Java Modelling Language JML)

- JML is a behavioral interface specification language
- The Application Programming Interface (API) in a typical programming language (for example consider the API of a set of Java classes) provides very little information
  - The method names and return types, argument names and types
- This type of API information is not sufficient for figuring out what a component does
- JML is a specification language that allows specification of the behavior of an API
  - not just its syntax, but its semantics
- JML specifications are written as *annotations*
  - As far as Java compiler is concerned they are comments but a JML compiler can interpret them

JML Project(s) and Materials

- Information about JML and JML based projects are available at Gary Leavens' website:
  - http://www.cs.ucf.edu/l̃eavens/JML/
- My lecture notes are based on:
  - Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212-232, June 2005
  - Slides by Yoonsik Cheon
  - JML tutorials by Joe Kiniry

JML

- One goal of JML is to make it easily understandable and usable by Java programmers, so it stays close to the Java syntax and semantics whenever possible
- JML supports design by contract style specifications with
  - Pre-conditions
  - Post-conditions
  - Class invariants
- JML supports quantification (5́cforall, 5́cexists), and specification-only fields and methods
  - Due to these features JML specifications are more expressive than Eiffel contracts and can be made more precise and complete compared to Eiffel contracts

JMLAnnotations

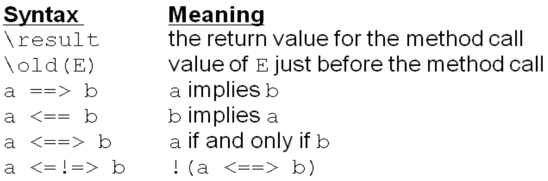
- JML assertions are added as comments to the Java source code
  - either between /*@ . . . @*/
  - or after //@
    - These are **annotations** and they are ignored by the Java compiler
- In JML properties are specified as Java boolean expressions
  - JML provides operators to support design by contract style specifications such as 5́cold and 5́cresult
  - JML also provides quantification operators (5́cforall, 5́cexists)
- JML also has additional keywords such as
  - requires, ensures, signals, assignable, pure, invariant, non null, . . .

Design by Contract in JML

- In JML constracts:
  - Preconditions are written as a requires clauses
  - Postconditions are written as ensures clauses
  - Invariants are written as invariant clauses

JML assertions

- JML assertions are written as Java expressions, but:
  - Cannot have side effects
    - No use of $=$, $++$, –, etc., and
    - Can only call *pure* methods (i.e., methods with no side effects)



Syntax	**Meaning**
`\result`	the return value for the method call
`\old(E)`	value of `E` just before the method call
`a ==> b`	`a` implies `b`
`a <== b`	`b` implies `a`
`a <==> b`	`a` if and only if `b`
`a <=!=> b`	`!(a <==> b)`

JML quantifiers

- JML supports several forms of quantifiers
  - Universal and existential ($\forall and \exists$)
  - (∀ Student s; class272.contains(s); s.getProject() != null)
  - (∀ Student s; class272.contains(s) ==> s.getProject() != null)
- Without quantifiers, we would need to write loops to specify these types of constraints

JML Quantifiers

- Quantifier expressions
  - Start with a decleration that is local to the quantifier expression
    - (∀ Student s;)
  - Followed by an optional range predicate
    - class272.contains(s);
  - Followed by the body of the quantifier
    - s.getProject() ! = null)

JML Quantifiers

- \ sum, \ product, \ min, \ max return the sum, product, min and max of the values of their body expression when the quantified variables satisfy the given range expression
- For example,
  - (\ sum int x; 1 $<=$ x && x $<=$ 5; x) denotes the sum of values between 1 and 5 inclusive
- The numerical quantifier, \ num_of, returns the number of values for quantified variables for which the range and the body predicate are true

JML Example: Purse

```
public class Purse {
  final int MAX_BALANCE;     int balance;
  //@ invariant 0 <= balance && balance <= MAX_BALANCE;
  byte[] pin;
  /*@ invariant pin != null && pin.length == 4
    @             && (\forall int i; 0 <= i && i < 4;
    @                       0 <= pin[i] && pin[i] <= 9);@*/


  /*@ requires 0 < mb && 0 <= b && b <= mb
  @            && p != null && p.length == 4
  @            && (\forall int i; 0 <= i && i < 4;
  @                                  0 <= p[i] && p[i] <= 9);
  @ assignable MAX_BALANCE, balance, pin;
  @ ensures MAX_BALANCE == mb && balance == b
  @  && (\forall int i; 0 <= i && i < 4; p[i] == pin[i]);@*/
Purse(int mb, int b, byte[] p) {
  MAX_BALANCE = mb; balance = b; pin = (byte[]) p.clone();}
```

JML Example: Purse

```
/*@ requires p != null && p.length >= 4;
  @ assignable \nothing;
  @ ensures \result <==> (\forall int i; 0 <= i && i < 4;
  @                                          pin[i] == p[i]);@*/
public boolean checkPin(byte[] p) {
  boolean res = true;
  for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
  return res;
}
/*@ requires amount >= 0;
  @ assignable balance;
  @ ensures balance == \old(balance) - amount
  @                 && \result == balance;
  @ signals (PurseException) balance == \old(balance);@*/
public int debit(int amount) throws PurseException {
  if (amount <= balance) { balance -= amount; return balance;
  else { throw new PurseException("overdrawn by " + amount);
}
```

JML Invariants

- Invariants (i.e., class invariants) must be maintained by all the methods of the class
  - Invariants must be preserved even when an exception is thrown
- Invariants are implicitly included in all pre and post-conditions
  - For constructors, invariants are only included in the post-condition not in the pre-condition. So, the constructors ensure the invariants but they do not require them.
- Invariants document design decision and makes understanding the code easier

Invariants for non-null references

- Many invariants, pre- and post-conditions are about references not being null.
  - The non_null keyword is a convenient short-hand for these.
  - public class Directory (
  - private /*@ non null @*/ File[] files;
  - void createSubdir(/*@ non null @*/ String name)(
  - ...
  - Directory /*@ non null @*/ getParent()(
  - ...
  -)

JML Example: Purse, Cont'd

- The assignable clause indicates that balance is the only field that will be assigned
  - This type of information is very useful for analysis and verification tools
  - The default assignable clause is: assignable \everything

JML post conditions

- ☐ The keyword \ old can be used to refer to the value of a field just before the execution of the method
- ☐ The keyword \ result can be used to refer to the return value of the method
- ☐ Both of these keywords are necessary and useful tools for specifying post conditions

Exceptions in JML

- In addition to normal post-conditions, JML also supports exceptional postconditions
  - Exceptional postconditions are written as signals clauses
- Exceptions mentioned in throws clause are allowed by default, i.e. the default signals clause is
  - signals (Exception) true;
  - To rule them out, you can add an explicit
  - signals (Exception) false;
  - or use the keyword normal_behavior
  - /*@ normal_behavior
  - @ requires ...
  - @ ensures ...
  - @*/

Class Exercise

- ☐ Think about the invariants in a set class
- ☐ Think about the invariants of a ration number of the form p/q
- ☐ Think about the invariants on a national-id-card number, IBAN number etc.

BoundedStack

```
public class BoundedStack {
  private Object[] elems;  private int size = 0;

  public BoundedStack(int n) { elems = new Object[n];}

  public void push(Object x) {
    elems[size] = x;
    size++;
  }
  public void pop() {
   size--;
   elems[size] = null;
  }
  public void resize(){
   int s[] = new int[2*elems.length+1];
   for (int i=0; i<elems.length; i++)
    s[i] = elems[i];
    elems = s;
    }
```

BoundedStack

```
public class BoundedStack {

        private /*@ spec_public non_null*/ Object[] elems;
        private /* spec_public*/ int size = 0;

    /*@ public invariant size >= 0 && size < elems.length
      @ && elems.length > 0
      @*/

    /*@ requires n >= 0
      @ ensures elems.length == n &&
      @*/
        public BoundedStack(int n) ;

}
```

BoundedStack

```
public class BoundedStack {


  /*@ requires x != null
    @ ensures size == \old(size + 1) &&
    elem[\old(size)] == x
    @*/
       public void push(Object x) ;

  /*@ requires size > 0 && elems.length > 0
    @ ensures size == \old(size) - 1 && elem[size] == null
    @*/
       public void pop() ;

  /*@ ensures elems.length == \old(2*elems.length + 1)
    @ \forall int i; (i >= 0 && i < elems.length-1)
    @ ==> (elems[i] == \old(elems[i]))
    @*/
       public void resize() ;
```

Model variables

- In JML one can declare and use variables that are only part of the specification and are not part of the implementation
- For example, instead of a Purse assume that we want to specify a PurseInterface
  - We could introduce a model variable called balance in order to specify the behavioral interface of a Purse
  - Then, a class implementing the PurseInterface would identify how its representation of the balance relates to this model variable

JML Libraries

- JML has an extensive library that supports concepts such as sets, sequences, and relations.
- These can be used in JML assertions directly without needing to re-specify these mathematical concepts

JML & Side-effects

- The semantics of JML forbids side-effects in assertions.
  - This both allows assertion checks to be used safely during debugging and supports mathematical reasoning about assertions.
- A method can be used in assertions only if it is declared as pure, meaning the method does not have any side-effects and does not perform any input or output.
- For example, if there is a method getBalance() that is declared as
  - /*@ pure @*/ int getBalance() (...)
  - then this method can be used in the specification instead of the field balance.
- Note that for pure methods, the assignable clause is implicitly
  - assignable 5́cnothing

Assert clauses

- The requires clauses are used to specify conditions that should hold just before a method execution, i.e., preconditions
- The ensures clauses are used to specify conditions that should hold just after a method execution, i.e., postconditions
- An assert clause can be used to specify a condition that should hold at some point in the code (rather than just before or right after a method execution)
- if (i ¡= 0 —— j ¡ 0) (
  - ...
-) else if (j ¡ 5) (
  - //@ assert i ¿ 0 && 0 ¡ j && j ¡ 5;
  - ...
-) else (
  - //@ assert i ¿ 0 && j ¿ 5;
  - ...
-)

Assert in JML

- Although assert is also a part of Java language now, assert in JML is more expressive
- for (n = 0; n ¡ a.length; n++)
  - if (a[n]==null) break;
- /*@ assert (5́cforall int i; 0 ¡= i && i ¡ n;
- @ a[i] != null);
- @*/

JML Tools

- There are tools for parsing and type-checking Java programs and their JML annotations
  - JML compiler (**jmlc**)
- There are tools for supporting documentation with JML
  - HTML generator (**jmldoc**)
- There are tools for runtime assertion checking:
  - Test for violations of assertions (pre, postconditions, invariants) during execution
  - Tool: **jmlrac**
- There are testing tools based on JML
  - JML/JUnit unit test tool: **jmlunit**
- Automated verification:
  - Automatically prove that contracts are never violated at any execution
  - Automatic verification is done statically (i.e., at compile time) using theorem proving
  - Tool: **ESC/ Java**
- Automatically inferring specifications: