



7 - Design Patterns (Part II)

Computer Science Department
California State University, Sacramento

Announcement

- **Midterm exam is schedule on March 15.**
- **Allowed one page of note.**
- **Coverage: materials till end of this week.**

Overview

- **Background**
- **Types of Design Patterns**
 - **Creational vs. Structural vs. Behavioral Patterns**
- **Specific Patterns**

<i>Composite</i>	<i>Singleton</i>
<i>Iterator</i>	<i>Observer</i>
<i>Strategy</i>	<i>Command</i>
<i>Proxy</i>	<i>Factory Method</i>
- **MVC Architecture**

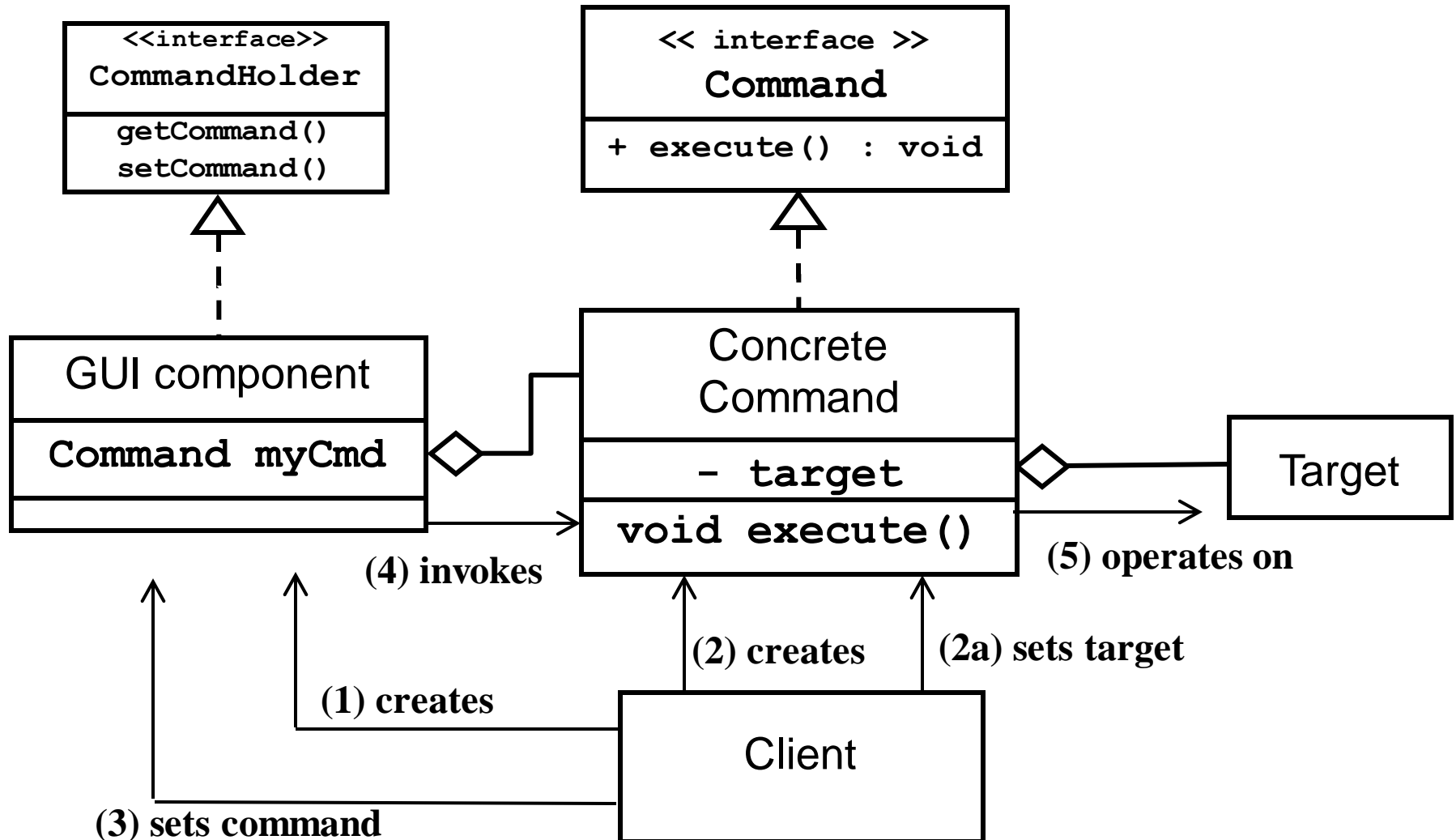
Part II – Design Pattern

The Command Pattern

Motivation

- Need to avoid having multiple copies of the code that performs the same operation invoked from different sources
- Desire to separate code implementing a command from the object which invokes it
- Need for maintaining *state information* about the command
 - Enabled or disabled?
 - Other data – e.g. invocation *count*

Command Pattern Organization



CN1 Command Class

- Implements **ActionListener** interface.
 - Provides empty body implementation for: `actionPerformed() == "execute()"`
 - We need to extend from **Command** and override `actionPerformed()` to perform the operation we would like to execute. In the constructor, do not forget to call `super("command name")`
- Also defines methods like: `isEnabled()` , `setEnabled()` , `getCommandName()`
- You can add a **command object** as a listener to a component using one of its `addXXXListener()` methods which takes **ActionListener** as a parameter (e.g. `addMouseListener()` in **Component**, `addActionListener()` in **Button**, `addKeyListener()` in **Form**)
- When activated (button pushed, pointer/key pressed etc), component calls `actionPerformed()` method of its listener/command

CN1 Command Class (cont.)

Using the `addKeyListener()` of `Form`, we can attach a listener (an object of a listener class which implements `ActionListener` or an object of subclass of `Command`) to a certain key.

This is called key binding: we are binding the listener/command (more specifically: the operation defined in its `actionPerformed()` method) to the key stroke, e.g:

```
/* Code for a form that uses key binding
//... [create a listener object called myCutCommand]
addKeyListener('c', myCutCommand);
//[when the 'c' key is hit, actionPerformed() method of CutCommand is called]
```


CN1 Button Class

Button is a “command holder”

- Defines methods like: `setCommand()` , `getCommand()`
- If you use `setCommand()` you do not need to also call `addActionListener()` since the command is automatically added as listeners on the button
- `setCommand()` changes the label of the button to the “command name” specified in command’s construction

To use the command design pattern properly on buttons, add the command object to the button using `setCommand()` (instead of `addActionListener()`).

Remember **CheckBox** is-a **Button** too!

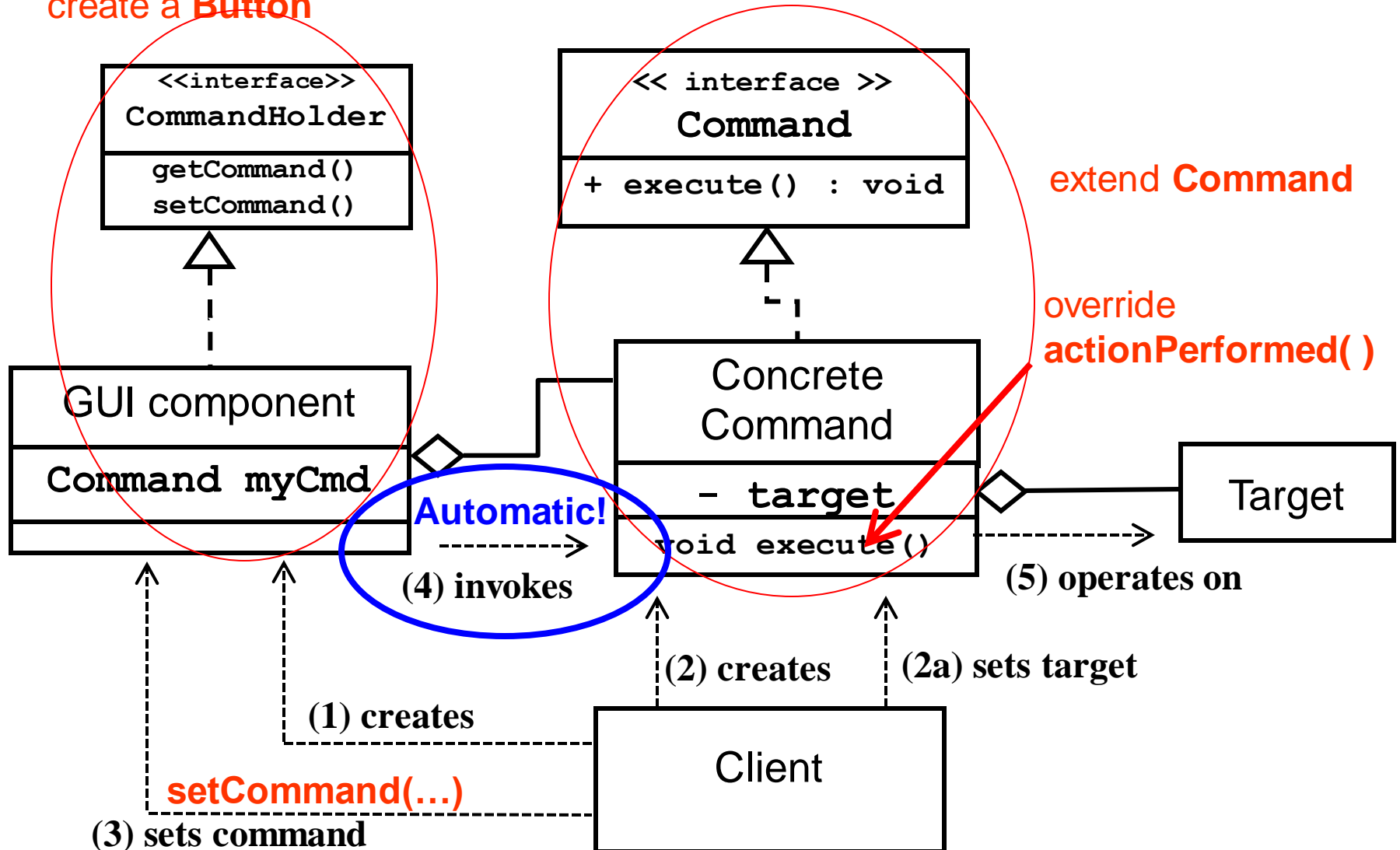
Adding Commands to Title Bar

When you add a regular command (e.g., without specifying a “SideComponent” property) to the title bar area using **Toolbar’s addCommandToXXX()** methods: (i.e. `myToolbar.addCommandToRightBar(myCommand)`)

- an item (side/overflow menu item or a title bar area item) is automatically generated and added to the title bar area
- The command (i.e `myCutCommand`) automatically becomes the listener of the item

Command Pattern – CN1

create a **Button**



Summary of Implementing Command Design Pattern in CN1

- **Define your command classes:**
 - Extend **Command** (which implements **ActionListener** interface and provides empty body implementation of **actionPerformed()**)
 - Override **actionPerformed()**
- **Add a Toolbar and buttons to your form**
- **Instantiate command objects in your form**
- **Add command objects to various entities:**
 - (1) buttons w/ **setCommand()** , (2) title bar area items w/ **Toolbar's addCommandToXXX()** methods, (3) key strokes w/ **Form's addKeyListener()**

Implementing Command Design Pattern in CN1

```
/** This class instantiates several command objects, creates several GUI  
 * components (button, side menu item, title bar item), and attaches the command objects  
 * to the GUI components and keys. The command objects then automatically get invoked  
 * when the GUI component or the key is activated.  
 */
```

```
public class CommandPatternForm extends Form {  
    public CommandPatternForm () {  
        //...[set a Toolbar to form]  
        Button buttonOne = new Button("Button One");  
        Button buttonTwo = new Button("Button Two");  
        //...[style and add two buttons to the form]  
        //create command objects and set them to buttons, notice that labels of buttons  
        //are set to command names  
        CutCommand myCutCommand = new CutCommand();  
        DeleteCommand myDeleteCommand = new DeleteCommand();  
        buttonOne.setCommand(myCutCommand);  
        buttonTwo.setCommand(myDeleteCommand);  
        //add cut command to the right side of title bar area  
        myToolbar.addCommandToRightBar (myCutCommand);  
        //add delete command to the side menu  
        myToolbar.addCommandToSideMenu (myDeleteCommand);  
        //bind 'c' ket to cut command and 'd' key to delete command  
        addKeyListener('c', myCutCommand);  
        addKeyListener('d', myDeleteCommand);  
        show();  
    }  
}
```

Implementing Command Design Pattern in CN1 (cont.)

```
/** These classes define a Command which perform "cut" and "delete" operations.
 * The commands are implemented as a subclass of Command, allowing it
 * to be added to any object supporting attachment of Commands.
 * This example does not show how the "Target" of the command is specified.
 */

public class CutCommand extends Command{
    public CutCommand() {
        super("Cut"); //do not forget to call parent constructor with command_name
    }
    @Override //do not forget @Override, makes sure you are overriding parent method
    //invoked to perform the 'cut' operation
    public void actionPerformed(ActionEvent ev) {
        System.out.println("Cut command is invoked...");
    }
}

public class DeleteCommand extends Command{
    public DeleteCommand() {
        super("Delete");
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Delete command is invoked...");
    }
}
```

The Strategy Pattern

Motivation

- A variety of algorithms exists to perform a particular operation
- The client needs to be able to select/change the choice of algorithm *at run-time*.

The Strategy Pattern (cont.)

Examples where different *strategies* might be used:

- Save a file in different formats (plain text, PDF, PostScript...)
- Compress a file using different compression algorithms
- Sort data using different sorting algorithms
- Capture video data using different encoding algorithms
- Plot the same data in different forms (bar graph, table, ...)
- Have a game's non-player character (NPC) change its AI
- Arrange components in an on-screen window using different layout algorithms

Example: NPC AI Algorithms

Typical client code sequence:

```
void attack() {  
  
    switch (characterType) {  
    case WARRIOR:    fight();           break;  
    case HUNTER:     fireWeapon();      break;  
    case PRIEST:     castDisablingSpell(); break;  
    case SHAMAN:     castMagicSpell();  break;  
    }  
}
```

Problem with this approach?

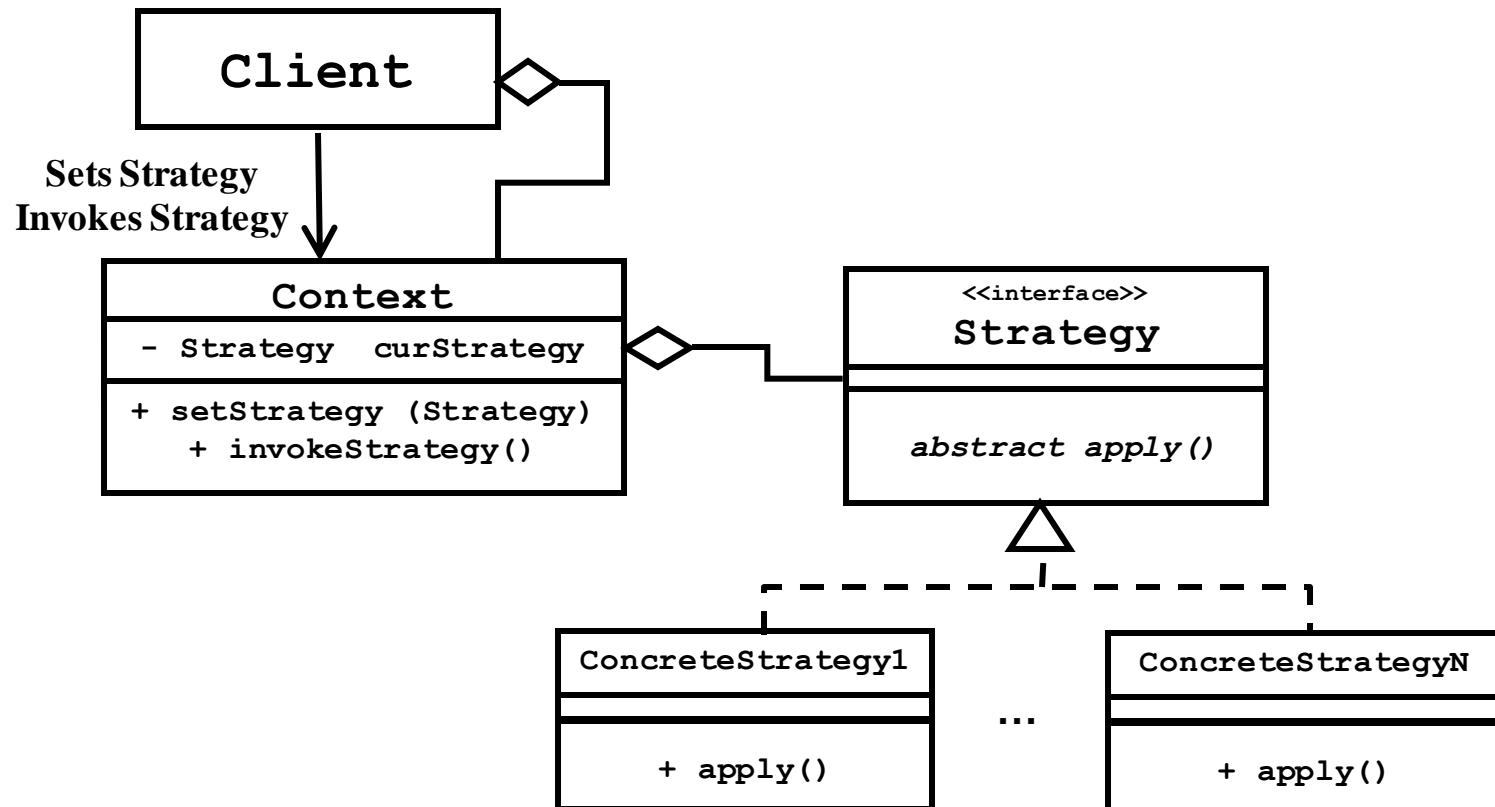
Changing or adding a plan requires changing the client!

Solution Approach

- Provide various objects that know how to “apply strategy” (e.g. apply fight, fireWeapon, or castMagicSpell strategies)
 - Each in a different way, but with a uniform interface
- The context (e.g. NPC) maintains a “current strategy” object
- Provide a mechanism for the client (e.g. Game) to *change* and *invoke* the current strategy object of a context

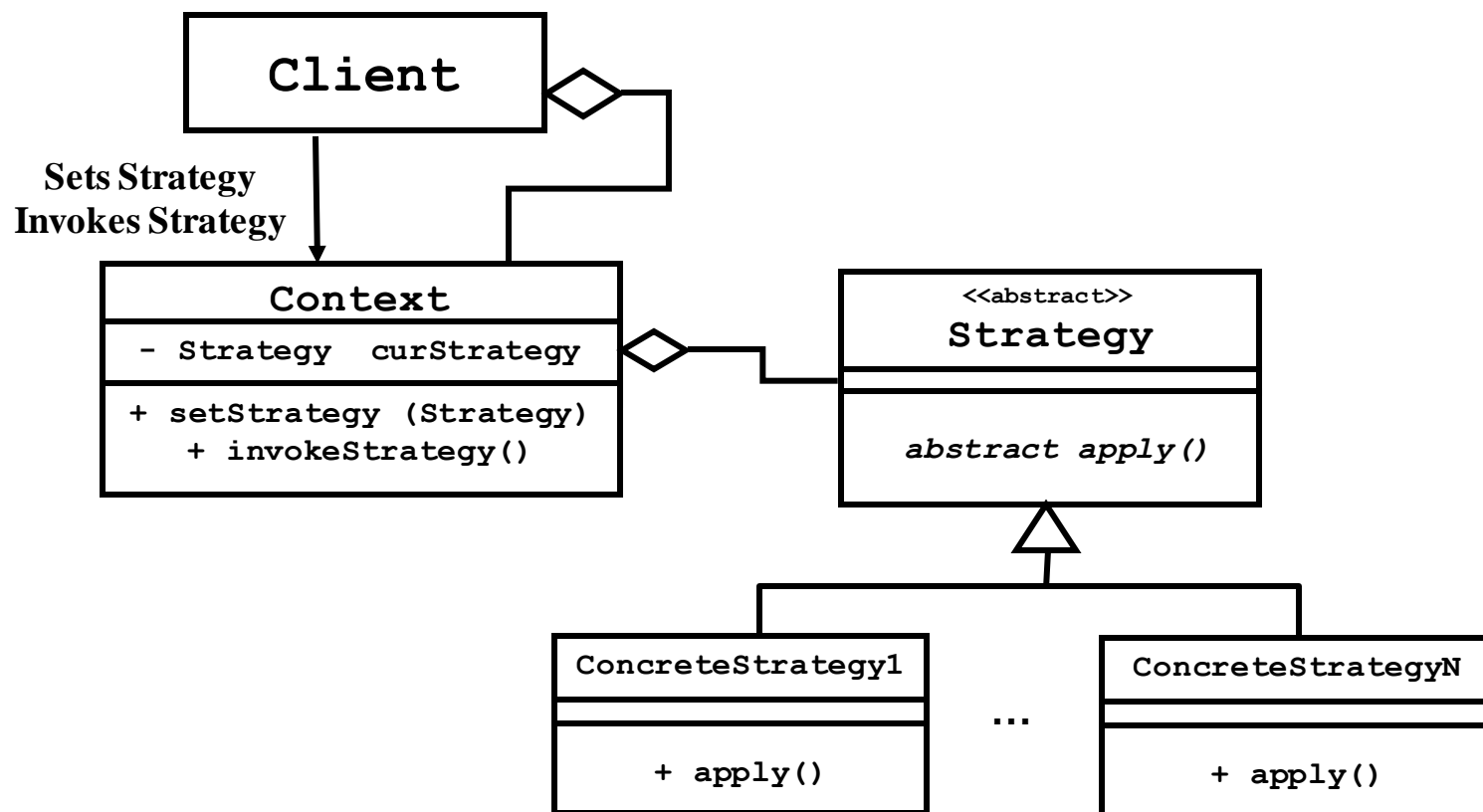
Strategy Pattern Organization

- Using Interfaces

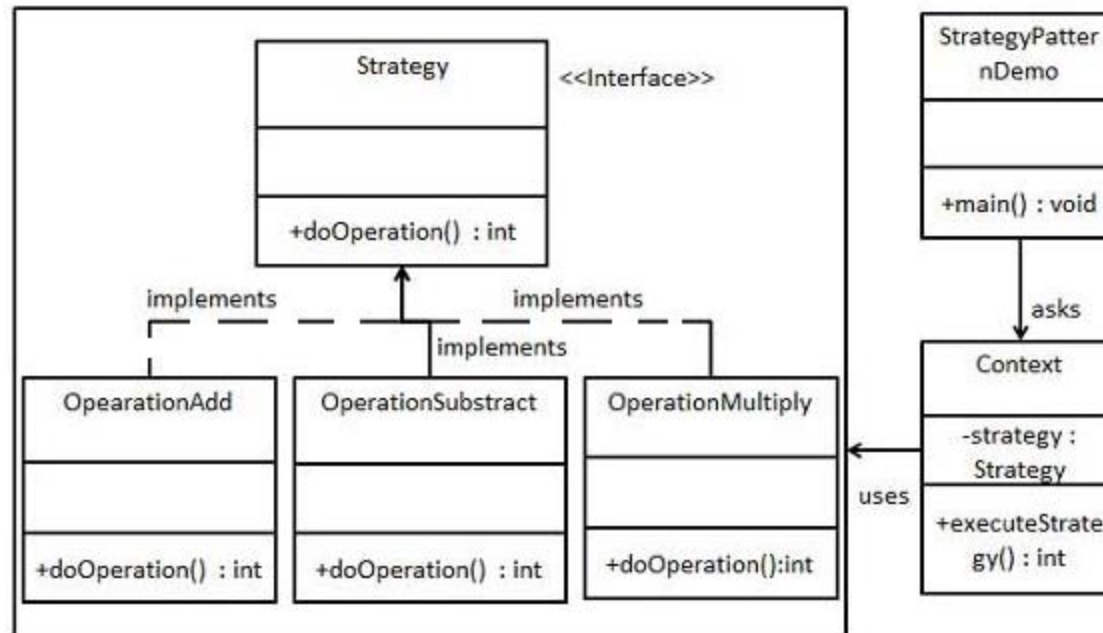


Strategy Pattern Organization (cont.)

- Using subclassing



Example: StrategyPatternDemo (Using Interface)



StrategyPatternDemo will use Context and strategy objects to demonstrate change in Context behavior based on strategy it deploys or uses.

Source: https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

Example: StrategyPatternDemo (Cont)

Step 1: Create an interface

```
public interface Strategy {  
    public int doOperation(int  
num1, int num2);  
}
```

Step 2: Create concrete classes implementing the same interface.

```
public class OperationAdd implements Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 + num2;  
    }  
}
```

```
public class OperationSubstract implements  
Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 - num2;  
    }  
}
```

```
public class OperationMultiply implements  
Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 * num2;  
    }  
}
```

Example: StrategyPatternDemo

(Cont)

Step 3: Create context class

```
public class Context {
    private Strategy strategy;

    public Context(Strategy
strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int
num1, int num2) {
        return
strategy.doOperation(num1, num2);
    }
}
```

Step 5: Verify result:

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

Step 4: Use the Context to see change in behavior when it changes its Strategy..

```
public class StrategyPatternDemo {
    public static void main(String[] args) {
        Context context = new Context(new OperationAdd());

        System.out.println("10 + 5 = " +
context.executeStrategy(10, 5));

        context = new Context(new OperationSubtract());
        System.out.println("10 - 5 = " +
context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " +
context.executeStrategy(10, 5));
    }
}
```

Example: CN1 Layouts

- Strategy abstract super class:

`Layout`

- Client is the `Form`
- Context: `Container` (e.g., `ContentPane` of `Form`)
- Context methods:

```
public void setLayout (Layout lout)
public void revalidate()
```

- Concrete strategies (`extends Layout`):

```
class FlowLayout()
class BorderLayout()
class GridLayout()
...
```

- “Apply” method (declared in the `Layout` super class):

```
abstract void layoutContainer(Container parent)
```


Example: NPC's in a Game

```
public interface Strategy {
    public void apply();
}

public class FightStrategy implements Strategy {
    public void apply() {
        //code here to do "fighting"
    }
}

public class FireWeaponStrategy implements Strategy {
    private Hunter hunter;
    public FireWeaponStrategy(Hunter h) {
        this.hunter = h; //record the hunter to which this strategy applies
    }
    public void apply() {
        //tell the hunter to fire a burst of 10 shots
        for (int i=0; i<10; i++) {
            hunter.fireWeapon();
        }
    }
}

public class CastMagicSpellStrategy implements Strategy {
    public void apply() {
        //code here to cast a magic spell
    }
}
```

NPC's in a Game (cont.)

“Contexts” :

```
public class Character {  
    private Strategy curStrategy;  
    public void setStrategy(Strategy s) {  
        curStrategy = s;  
    }  
    public void invokeStrategy() {  
        curStrategy.apply();  
    }  
}
```

```
public class Warrior extends Character {  
    //code here for Warrior specific methods  
}
```

```
public class Shaman extends Character {  
    //code here for Shaman specific methods  
}
```

```
public class Hunter extends Character {  
    private int bulletCount ;  
  
    public boolean isOutOfAmmo() {  
        if (bulletCount <= 0) return true;  
        else return false;  
    }  
    public void fireWeapon() {  
        bulletCount -- ;  
    }  
  
    //code here for other Hunter specific  
    //methods  
}
```

Assigning / Changing Strategies

```

/** This Game class demonstrates the use of the Strategy Design Pattern
* by assigning attack response strategies to each of several game characters.
*/
public class Game {
    //the list of non-player characters in the game
    ArrayList<Character> npcList = new ArrayList<Character>();

    public Game() { //construct some characters, assigning each a starting strategy
        Warrior w1 = new Warrior();
        w1.setStrategy(new FightStrategy());
        npcList.add(w1);

        Hunter h1 = new Hunter();
        h1.setStrategy(new FireWeaponStrategy(h1));
        npcList.add(h1);

        Shaman s1 = new Shaman();
        s1.setStrategy(new CastSpellStrategy());
        npcList.add(s1);
    }

    public void attack() { //force each character to execute its attack response
        for (Character c : npcList) {
            c.invokeStrategy();
        }
    }

    public void updateCharacters() { //update any strategies that need changing
        for (Character c : npcList) {
            if(c instanceof Hunter) {
                if ( ((Hunter)c).isOutOfAmmo() ) {
                    //change the character's strategy
                    c.setStrategy(new FightStrategy());
                }
            }
        }
    }
}

```

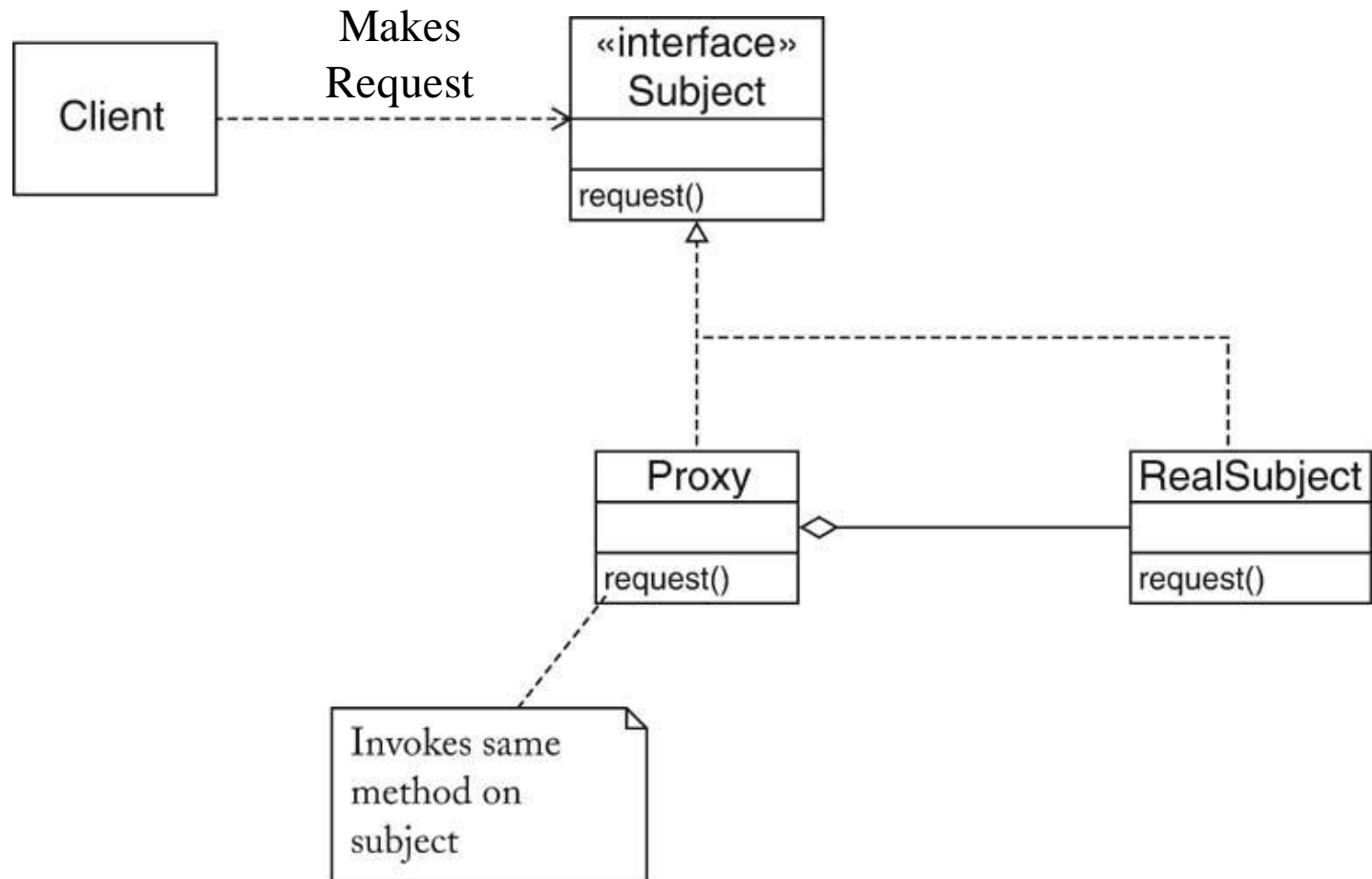
The Proxy Pattern

- Motivation
 - Undesirable target object manipulation
 - Access required, but not to all operations
 - Expensive target object manipulation
 - Lengthy image load time
 - Significant object creation time
 - Large object size
 - Inaccessible target object
 - Resides in a different address space
 - E.g. another JVM or a machine on a network

Proxy Types

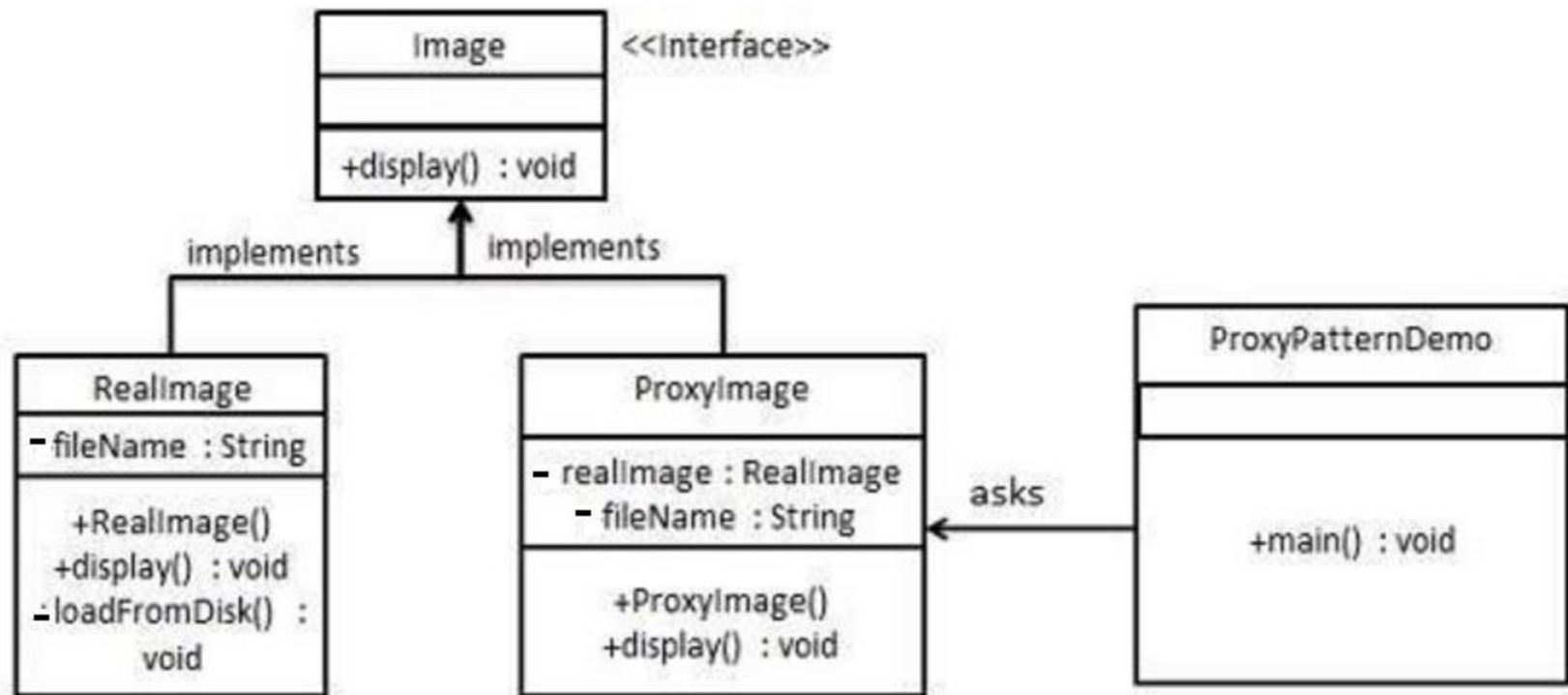
- **Protection Proxy – controls access**
- **Virtual Proxy – acts as a stand-in**
- **Remote Proxy – local stand-in for object in another address space**
- **This type of design pattern comes under structural pattern.**

Proxy Pattern Organization



Proxy Example 1

ProxyImage is a proxy class to reduce memory footprint of RealImage object loading. ProxyPatternDemo, our demo class, will use ProxyImage to get an Image object to load and display as it needs.

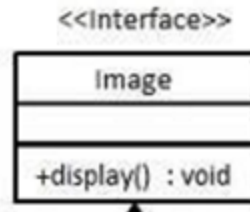


Source: https://www.tutorialspoint.com/design_pattern/proxy_pattern.htm

Proxy Example 1 (Cont)

Step 1: Create an interface

```
public interface Image {
    void display();
}
```



Step 2: Create concrete classes implementing the same interface.

public class RealImage implements Image {

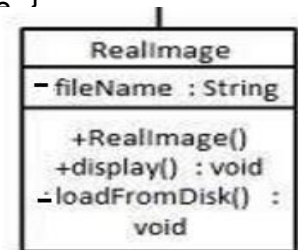
```
    private String fileName;
```

```
    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }
```

```
@Override
```

```
public void display() {
    System.out.println("Displaying " + fileName);
}
```

```
private void loadFromDisk(String fileName) {
    System.out.println("Loading " + fileName);
}
```



Step 2: Create concrete classes implementing the same interface.

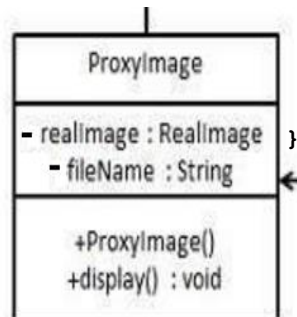
public class ProxyImage implements Image{

```
    private RealImage realImage;
    private String fileName;
```

```
    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }
```

```
@Override
```

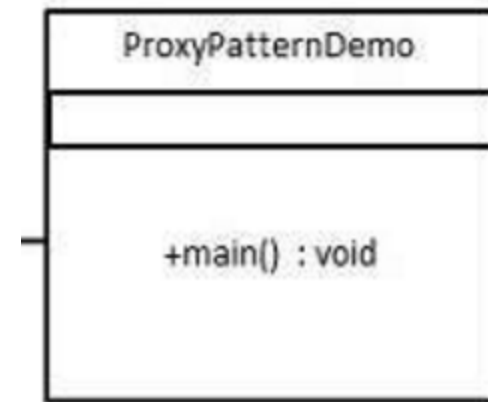
```
public void display() {
    if(realImage == null){
        realImage = new RealImage(fileName);
    }
    realImage.display();
}
}
```



Proxy Example 1 (Cont)

Step 3: Use the ProxyImage to get object of RealImage class when required..

```
public class ProxyPatternDemo {  
  
    public static void main(String[] args) {  
        Image image = new ProxyImage("test_10mb.jpg");  
  
        //image will be loaded from disk  
        image.display();  
        System.out.println("");  
  
        //image will not be loaded from disk  
        image.display();  
    }  
}
```



Step 4: Verify result.

```
Loading test_10mb.jpg  
Displaying test_10mb.jpg  
  
Displaying test_10mb.jpg
```

Proxy Example 2

```
interface IGameWorld {  
    Iterator getIterator();  
    void addGameObject(GameObject o);  
    boolean removeGameObject (GameObject o);  
}
```

```
/**A proxy which prohibits removal of GameWorldObjects from the GameWorld*/  
public class GameWorldProxy implements IObservable, IGameWorld {  
    private GameWorld realGameWorld ;  
  
    public GameWorldProxy (GameWorld gw)  
        { realGameWorld = gw; }  
  
    public Iterator getIterator ()  
        { return realGameWorld.getIterator(); }  
  
    public void addGameObject(GameObject o)  
        { realGameWorld.addGameObject(o) ; }  
  
    public boolean removeGameObject (GameObject o)  
        { return false ; }  
    //...[also has methods implementing IObservable]  
}
```

Proxy Example 2 (cont.)

```

/** This class defines a Game containing a GameWorld with a ScoreView Observer. */
public class Game {
    public Game() {
        GameWorld gw = new GameWorld();    //construct a GameWorld
        ScoreView sv = new ScoreView();    //construct a ScoreView
        gw.addObserver(sv);                //register ScoreView as a GameWorld Observer
    }
}

-----

/** This class defines a GameWorld which is an Observable and maintains a list of
 * Observers; when the GameWorld needs to notify its Observers of changes it does so
 * by passing a GameWorldProxy to the Observers. */
public class GameWorld implements IObservable, IGameWorld {
    private Vector<GameObject> myGameObjectList = new Vector<GameObject>();
    private Vector<IObserver> myObserverList = new Vector<IObserver>();
    public Iterator<GameObject> getIterator() { ... }
    public void addGameObject(GameObject o) { ... }
    public boolean removeGameObject(GameObject o) {
        //code here to remove the specified GameObject from the GameWorld...
    }
    public void addObserver(IObserver o) { myObserverList.add(o); }

    //Pass a GameWorldProxy to Observers, thus prohibiting Observer removal of GameObjects
    public void notifyObservers() {
        GameWorldProxy proxy = new GameWorldProxy(this);
        for (IObserver o : myObserverList) {
            o.update((IObservable)proxy, null);
        }
    }
}

```

The Factory Method Pattern

- **Motivation**
 - We create object without exposing the creation logic to the client and refer to newly created object using a common interface.
 - Sometimes a class can't anticipate the class of objects it must create
 - It is sometimes better to delegate specification of object types to subclasses
 - It is frequently desirable to avoid binding application-specific classes into a set of code

Example: Maze Game

```
public class MazeGame {
```

```
    // This method creates a maze for the game, using a hard-coded structure for the  
    // maze (specifically, it constructs a maze with two rooms connected by a door).
```

```
    public Maze createMaze () {
```

```
        Maze theMaze = new Maze() ;    //construct an (empty) maze
```

```
        Room r1 = new Room(1) ;        //construct components for the maze
```

```
        Room r2 = new Room(2) ;
```

```
        Door theDoor = new Door(r1, r2);
```

```
        r1.setSide(NORTH, new Wall()); //set wall properties for the rooms
```

```
        r1.setSide(EAST,  theDoor);
```

```
        r1.setSide(SOUTH, new Wall());
```

```
        r1.setSide(WEST,  new Wall());
```

```
        r2.setSide(NORTH, new Wall());
```

```
        r2.setSide(EAST,  new Wall());
```

```
        r2.setSide(SOUTH, new Wall());
```

```
        r2.setSide(WEST,  theDoor);
```

```
        theMaze.addRoom(r1); //add the rooms to the maze
```

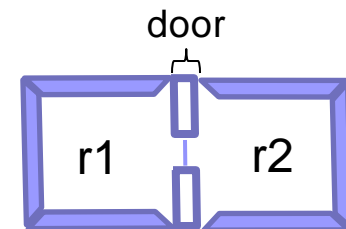
```
        theMaze.addRoom(r2);
```

```
        return theMaze ;
```

```
    }
```

```
    //other MazeGame methods here (e.g. a main program which calls createMaze())...
```

```
}
```



Problems with `createMaze()`

- Inflexibility; lack of “reusability”
- Reason: it “hardcodes” the maze types
 - Suppose we want to create a maze with (e.g.)
 - Magic Doors
 - Enchanted Rooms
 - Possible solutions:
 - Subclass `MazeGame` and override `createMaze()` (i.e., create a whole new version with new types)
 - Hack `createMaze()` apart, changing pieces as needed

createMaze () Factory Methods

```
public class MazeGame {  
  
    //factory methods - each returns a MazeComponent of a given type  
    public Maze makeMaze()          { return new Maze() ; }  
    public Room makeRoom(int id)    { return new Room(id) ; }  
    public Wall makeWall()          { return new Wall() ; }  
    public Door makeDoor(Room r1, Room r2) { return new Door(r1,r2) ; }  
  
    // Create a maze for the game using factory methods  
    public Maze createMaze () {  
        Maze theMaze = makeMaze() ;  
        Room r1 = makeRoom(1) ;  
        Room r2 = makeRoom(2) ;  
        Door theDoor = makeDoor(r1, r2);  
        r1.setSide(NORTH, makeWall());  
        r1.setSide(EAST,  theDoor);  
        r1.setSide(SOUTH, makeWall());  
        r1.setSide(WEST,  makeWall());  
        r2.setSide(NORTH, makeWall());  
        r2.setSide(EAST,  makeWall());  
        r2.setSide(SOUTH, makeWall());  
        r2.setSide(WEST,  theDoor);  
        theMaze.addRoom(r1);  
        theMaze.addRoom(r2);  
        return theMaze ;  
    }  
    ...  
}
```

Overriding Factory Methods

*//This class shows how to implement a maze made of different types of rooms. Note
// in particular that **we can call exactly the same (inherited) createMaze() method**
// to obtain a new "EnchantedMaze".*

```
public class EnchantedMazeGame extends MazeGame {

    //override MakeRoom to produce "EnchantedRooms"
    @Override
    public Room makeRoom(int id) {

        //create the spell necessary to enter the enchanted room
        Spell spell = makeSpell() ;

        //construct and return an EnchantedRoom requiring a spell to be entered
        return new EnchantedRoom(id, spell);
    }

    //override MakeDoor to produce a door requiring a spell
    @Override
    public Door makeDoor(Room r1, Room r2) {

        //construct and return a Door requiring a spell to be entered
        return new DoorNeedingSpell(r1, r2);
    }

    //new factory method for making spells
    public Spell makeSpell() { return new Spell(); }
    ...
}
```