

9. Guidelines, Idioms and Patterns

Roadmap

- > Idioms, Patterns and Frameworks
 - Programming style: Code Talks; Code Smells
- > Basic Idioms
 - Delegation, Super, Interface
- > Some Design Patterns
 - Adapter, Proxy, Template Method, Composite, Observer, Visitor, State



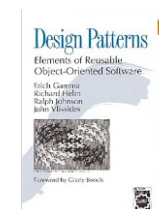
Roadmap

- > **Idioms, Patterns and Frameworks**
 - **Programming style: Code Talks; Code Smells**
- > **Basic Idioms**
 - Delegation, Super, Interface
- > **Some Design Patterns**
 - Adapter, Proxy, Template Method, Composite, Observer, Visitor, State

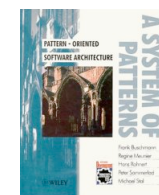


Sources

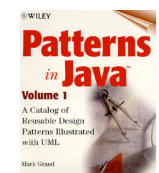
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides,
Design Patterns, Addison Wesley, Reading, MA, 1995.



Frank Buschmann, et al., *Pattern-Oriented Software Architecture
— A System of Patterns*, Wiley, 1996



Mark Grand, *Patterns in Java, Volume 1*, Wiley, 1998



Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997



“Code Smells”, <http://c2.com/cgi/wiki?CodeSmell>

OR <http://sis36.berkeley.edu/projects/streek/agile/bad-smells-in-code.html>



Style

Code Talks

- > Do the simplest thing you can think of (KISS)
 - Don't over-design
 - Implement things *once and only once*
 - *First do it, then do it right, then do it fast*
(don't optimize too early)

- > Make your intention clear
 - Write *small methods*
 - Each method should *do one thing only*
 - Name methods for *what they do*, not how they do it
 - Write to an *interface*, not an implementation



Refactoring

Redesign and refactor when the code starts to “smell”

Code Smells (<http://sis36.berkeley.edu/projects/streek/agile/bad-smells-in-code.html>)

- > Methods *too long* or too complex
 - decompose using helper methods
- > *Duplicated code*
 - factor out the common parts
(e.g., using a *Template method* Pattern)
- > Violation of *encapsulation*
 - redistribute responsibilities
- > Too much communication (*high coupling*)
 - redistribute responsibilities



Many idioms and patterns can help you improve your design ...

Refactoring Long Methods

2. From the selection's context menu in the editor, select **Refactor > Extract Method...**

short is good!

*If I need to comment then
extract as method*

```

Class superClass= theClass;
Vector names= new Vector ();
while (true) {
    ...
    From(superClass) {
        ...
        .length; i++) {
            ...
            i], names, theClass);
        ...
        Superclass ();
    }
}
...
found in "+theClass.getName ();

```

* Cons	Copy	Ctrl+C	
*/	Paste	Ctrl+V	
public	Source	Alt+Shift+S	▶
set	Refactor	Alt+Shift+T	▶
}	Surround With	Alt+Shift+Z	▶
/**	Local History		▶
* Adds	References		▶
*/	Declarations		▶
public	Move...	Alt+Shift+V	
	Change Method Signature...	Alt+Shift+C	
	Extract Method...	Alt+Shift+M	
	Extract Interface...		
	Use Supertype Where Possible...		

What are Idioms and Patterns?

Idioms	Idioms are <i>common programming techniques</i> and conventions. They are often <i>language-specific</i> . (http://c2.com/ppr/wiki/Javaldioms/Javaldioms.html)
Patterns	Patterns document <i>common solutions to design problems</i> . They are language-independent.
Libraries	Libraries are <i>collections of functions, procedures or other software components</i> that can be used in many applications.
Frameworks	Frameworks are open libraries that define the <i>generic architecture</i> of an application, and can be extended by adding or deriving new classes. (http://martinfowler.com/bliki/InversionOfControl.html)

Frameworks typically make use of common idioms and patterns.

Roadmap

- > Idioms, Patterns and Frameworks
 - Programming style: Code Talks; Code Smells
- > **Basic Idioms**
 - **Delegation, Super, Interface**
- > Some Design Patterns
 - Adapter, Proxy, Template Method, Composite, Observer, Visitor, State



Delegation

- ✎ How can an object share behaviour without inheritance?
- ✓ *Delegate some of its work to another object*

Inheritance is a common way to extend the behaviour of a class, but can be *an inappropriate way to combine features*.

Delegation reinforces encapsulation by keeping roles and responsibilities distinct.

Delegation

Example

- > When a `TestSuite` is asked to `run()`, it delegates the work to each of its `TestCases`.

Consequences

- > More *flexible, less structured* than inheritance.

Delegation is one of the most basic object-oriented idioms, and is used by almost all design patterns.

Delegation example

```
public class TestSuite implements Test {
    ...
    public void run(TestResult result) {
        for(Enumeration e = fTests.elements();
            e.hasMoreElements();)
        {
            if (result.shouldStop())
                break;
            Test test = (Test) e.nextElement();
            test.run(result);
        }
    }
}
```

delegate



Super

- ✎ How do you extend behavior inherited from a superclass?
- ✓ *Overwrite the inherited method, and send a message to “super” in the new method.*

Sometimes you just want to *extend* inherited behavior, rather than replace it.

Super

Examples

- > `Place.paint()` extends `Panel.paint()` with specific painting behaviour
- > Constructors for many classes, e.g., `TicTacToe`, invoke their superclass constructors.

Consequences

- > *Increases coupling* between subclass and superclass: if you change the inheritance structure, super calls may break!


Never use super to invoke a method different than the one being overwritten — use “this” instead!

Super examples

```
public class Place extends Panel {  
    ...  
    public void paint(Graphics g) {  
        super.paint(g);  
        Rectangle rect = g.getClipBounds();  
        int h = rect.height;  
        int w = rect.width;  
        int offset = w/10;  
        g.drawRect(0,0,w,h);  
        if (image != null) {  
            g.drawImage(image, offset, offset, w-2*offset, h-2*offset, this);  
        }  
    }  
    ...  
}
```

```
public class TicTacToe extends AbstractBoardGame {  
    public TicTacToe(Player playerX, Player playerO)  
    {  
        super(playerX, playerO);  
    }  
}
```

Interface

-  How do you keep a client of a service independent of classes that provide the service?
- ✓ *Have the client use the service through an interface rather than a concrete class.*

If a client *names a concrete class* as a service provider, then *only instances of that class* or its subclasses can be used in future.

By naming an interface, an instance of *any* class that implements the interface can be used to provide the service.

Interface

Example

- > Any object may be registered with an `Observable` if it implements the `Observer` interface.
- > ***Consequences***
- > Interfaces *reduce coupling* between classes.
- > They also *increase complexity* by adding indirection.

Interface example

```
public class GameGUI extends JFrame implements Observer {  
    ...  
    public void update(Observable o, Object arg) {  
        Move move = (Move) arg;  
        showFeedBack("got an update: " + move);  
        places_[move.col][move.row].setMove(move.player);  
    }  
    ...  
}
```

Roadmap

- > **Idioms, Patterns and Frameworks**
 - Programming style: Code Talks; Code Smells
- > **Basic Idioms**
 - Delegation, Super, Interface
- > **Some Design Patterns**
 - **Adapter, Proxy, Template Method, Composite, Observer, Visitor, State**



Adapter Pattern

- ✎ How do you use a class that provide the right features but the wrong interface?
- ✓ *Introduce an adapter.*

An adapter *converts the interface* of a class into another interface clients expect.

- > The client and the adapted object *remain independent.*
- > An adapter adds *an extra level of indirection.*

Also known as Wrapper

Adapter Pattern

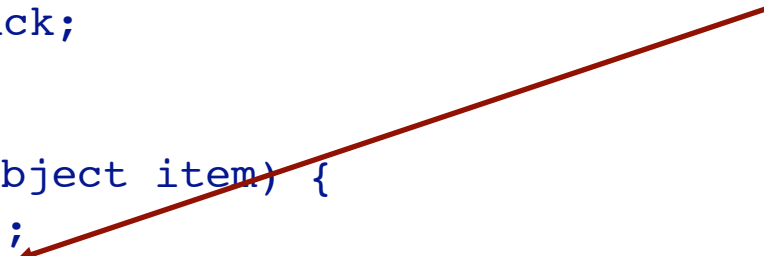
Examples

- > A `WrappedStack` adapts `java.util.Stack`, throwing an `AssertionException` when `top()` or `pop()` are called on an empty stack.
- > An `ActionListener` converts a call to `actionPerformed()` to the desired handler method.
- > ***Consequences***
- > The client and the adapted object *remain independent*.
- > An adapter adds *an extra level of indirection*.

Adapter Pattern example

```
public class WrappedStack implements StackInterface {  
  
    private java.util.Stack stack;  
  
    public WrappedStack() {  
        this(new Stack());  
    }  
  
    public WrappedStack(Stack stack) {  
        this.stack = stack;  
    }  
  
    public void push(Object item) {  
        stack.push(item);  
        assert this.top() == item;  
        assert invariant();  
    }  
}
```

delegate
request to
adaptee



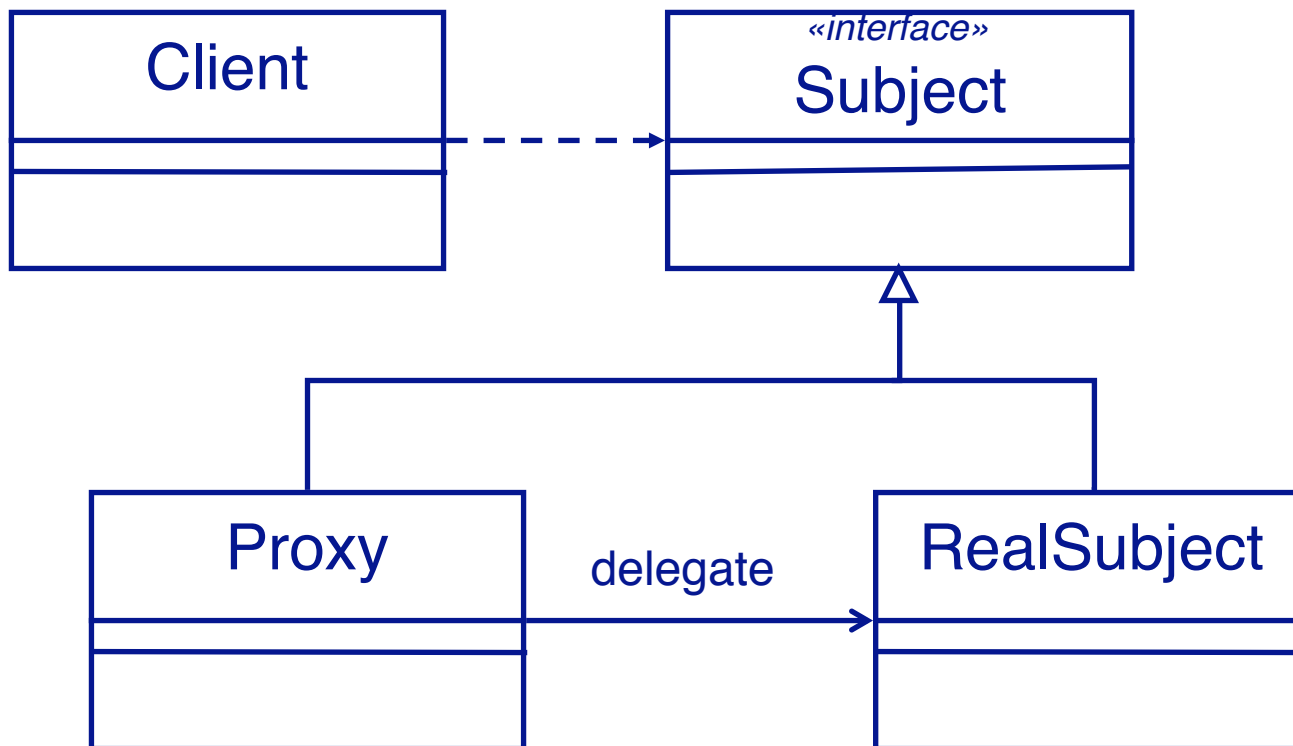
Proxy Pattern

-  How do you hide the complexity of accessing objects that require pre- or post-processing?
- ✓ *Introduce a proxy to control access to the object.*

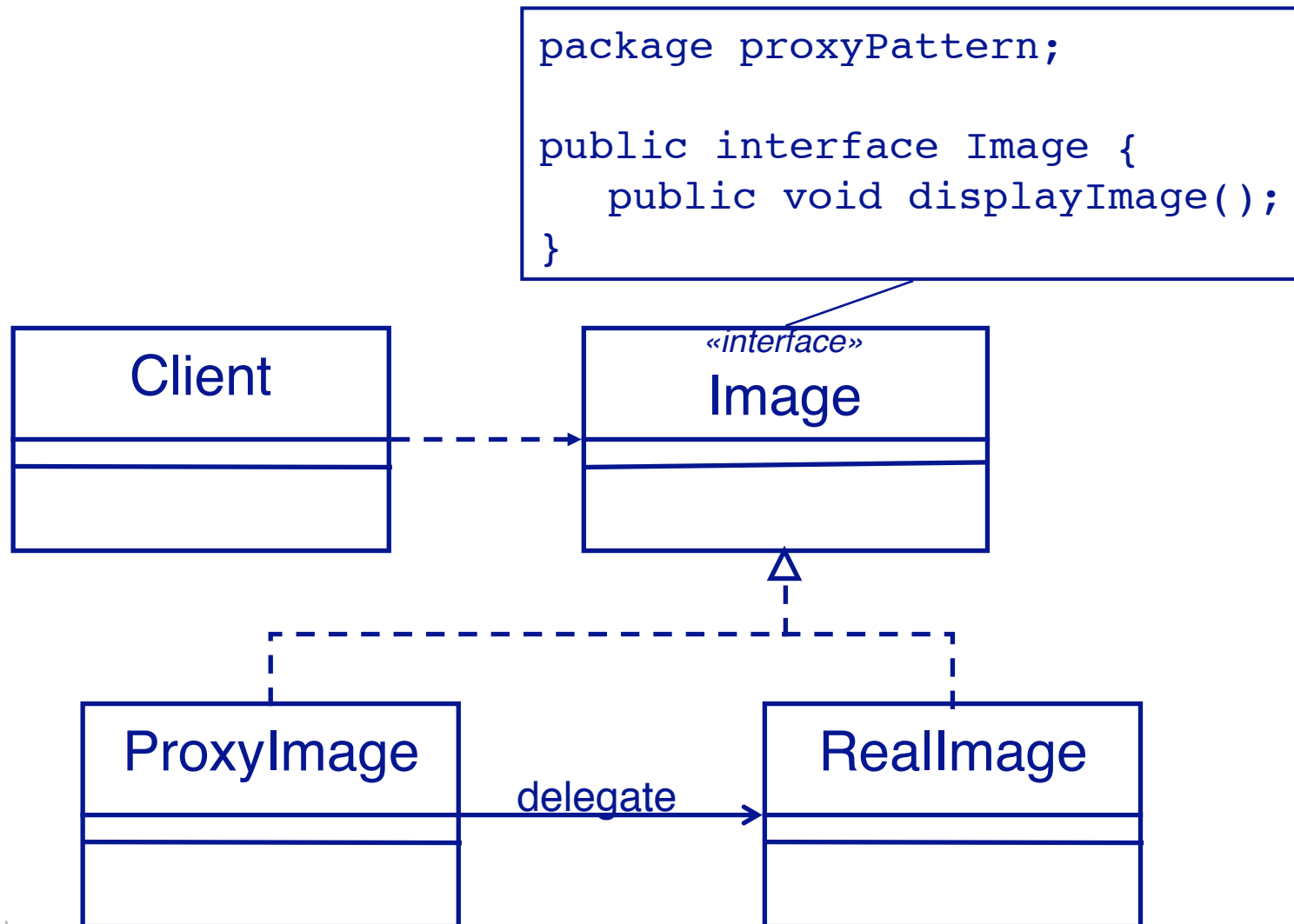
Some services require special pre or post-processing. Examples include objects that reside on a remote machine, and those with security restrictions.

A proxy provides the same interface as the object that it controls access to.

Proxy Pattern - UML



Proxy Pattern Example (1)



Proxy Pattern Example (2)

```
public class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename){
        this.filename = filename;
    }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); //load only on demand
        }
        image.displayImage();
    }
}
```



delegate request to real subject

Proxy Pattern Example (3)

```
public class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        System.out.println("Loading "+filename);
    }

    public void displayImage() {
        System.out.println("Displaying "+filename);
    }
}
```

Proxy Pattern Example (4) - the Client

```
public class ProxyExample {
    public static void main(String[] args) {

        ArrayList<Image> images = new ArrayList<Image>();
        images.add(new ProxyImage("HiRes_10MB_Photo1"));
        images.add(new ProxyImage("HiRes_10MB_Photo2"));
        images.add(new ProxyImage("HiRes_10MB_Photo3"));

        images.get(0).displayImage();
        images.get(1).displayImage();
        images.get(0).displayImage(); // already loaded
    }
}
```

Proxies are used for remote object access

Example

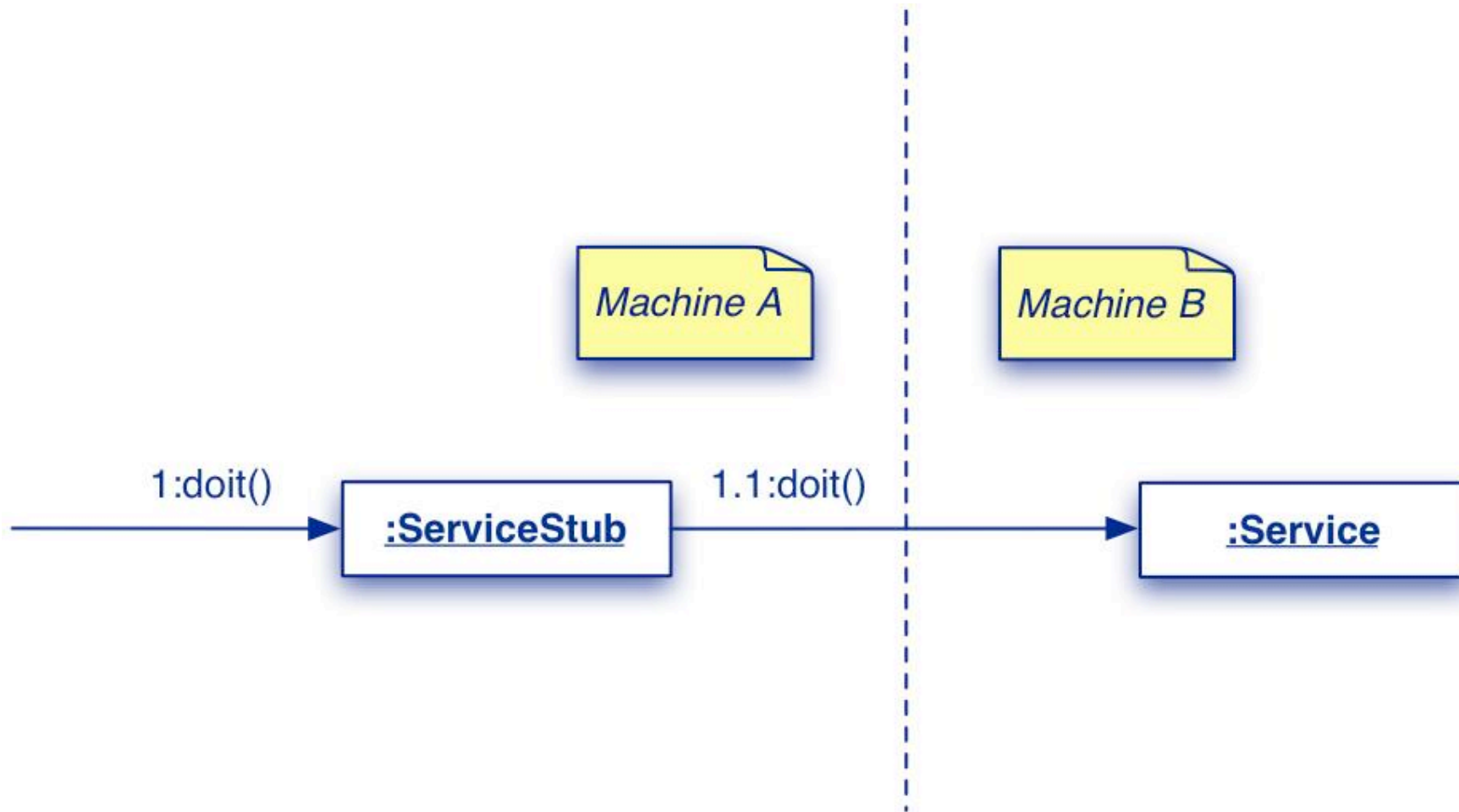
- > A Java “stub” for a remote object accessed by Remote Method Invocation (RMI).

Consequences


- > A Proxy decouples clients from servers. A Proxy introduces a level of indirection.

Proxy differs from Adapter in that it does not change the object's interface.

Proxy remote access example



Template Method Pattern

-  How do you implement a generic algorithm, deferring some parts to subclasses?
- ✓ *Define it as a Template Method.*

A Template Method *factors out the common part of similar algorithms*, and delegates the rest to:

- *hook methods* that subclasses *may extend*, and
- *abstract methods* that subclasses *must implement*.

Template Method Pattern (2)

Example

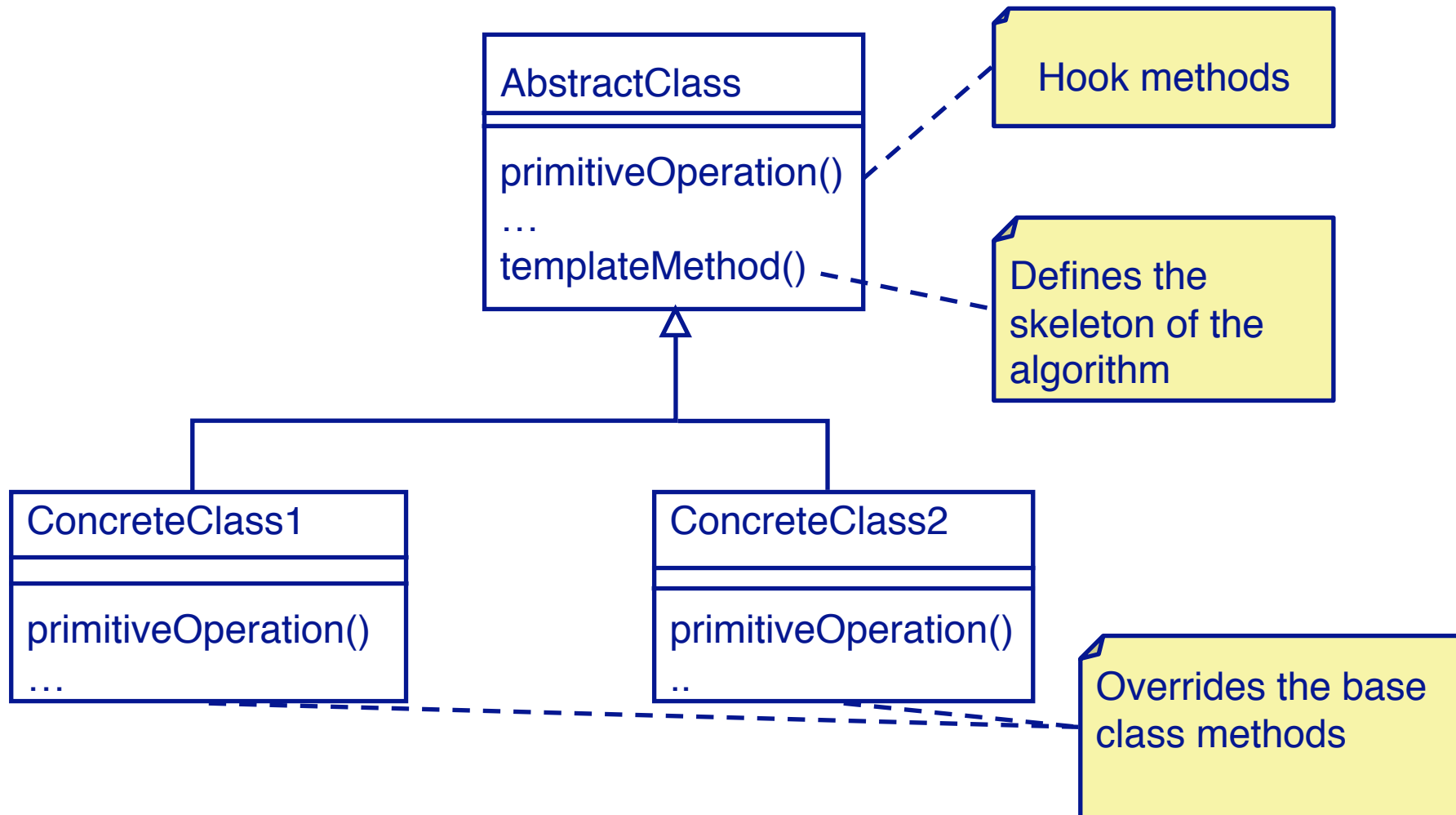
- > `TestCase.runBare()` is a template method that calls the hook method `setUp()`.
- > `AbstractBoardGame`'s constructor defers initialization to the abstract `init()` method

Consequences

- > Template methods lead to an *inverted control structure* since a parent class calls the operations of a subclass and not the other way around.

Template Method is used in most frameworks to allow application programmers to easily extend the functionality of framework classes.

Template Method Pattern - UML




Template Method Pattern Example

Subclasses of `TestCase` are expected to *override hook method* `setUp()` and possibly `tearDown()` and `runTest()`.

```
public abstract class TestCase implements Test {
    ...
    public void runBare() throws Throwable {
        setUp();
        try { runTest(); }
        finally { tearDown(); }
    }
    protected void setUp() { }           // empty by default
    protected void tearDown() { }
    protected void runTest() throws Throwable { ... }
}
```

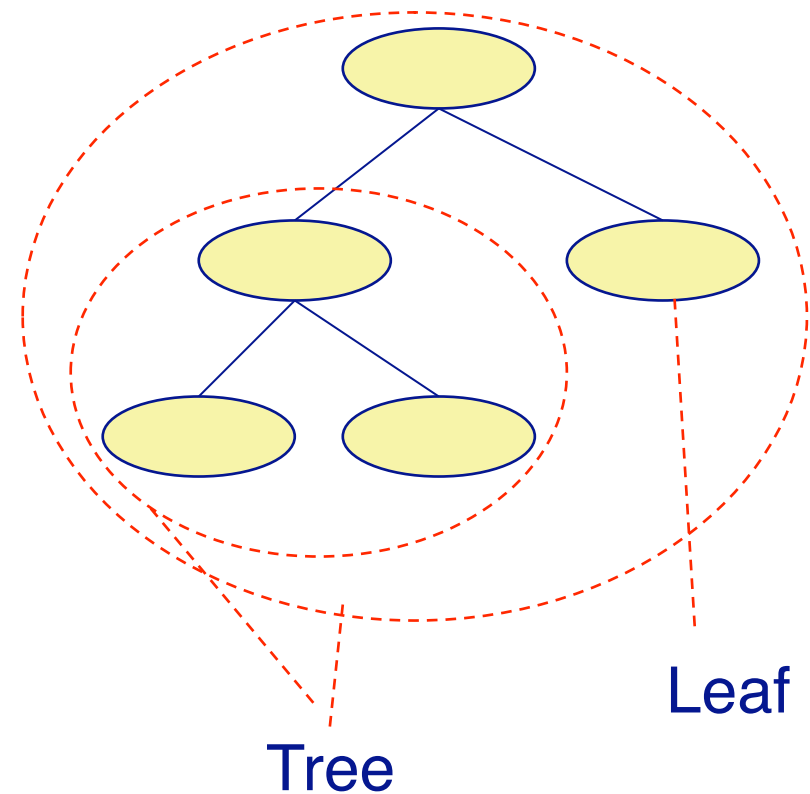
Composite Pattern

-  How do you manage a part-whole hierarchy of objects in a consistent way?
- ✓ *Define a common interface that both parts and composites implement.*

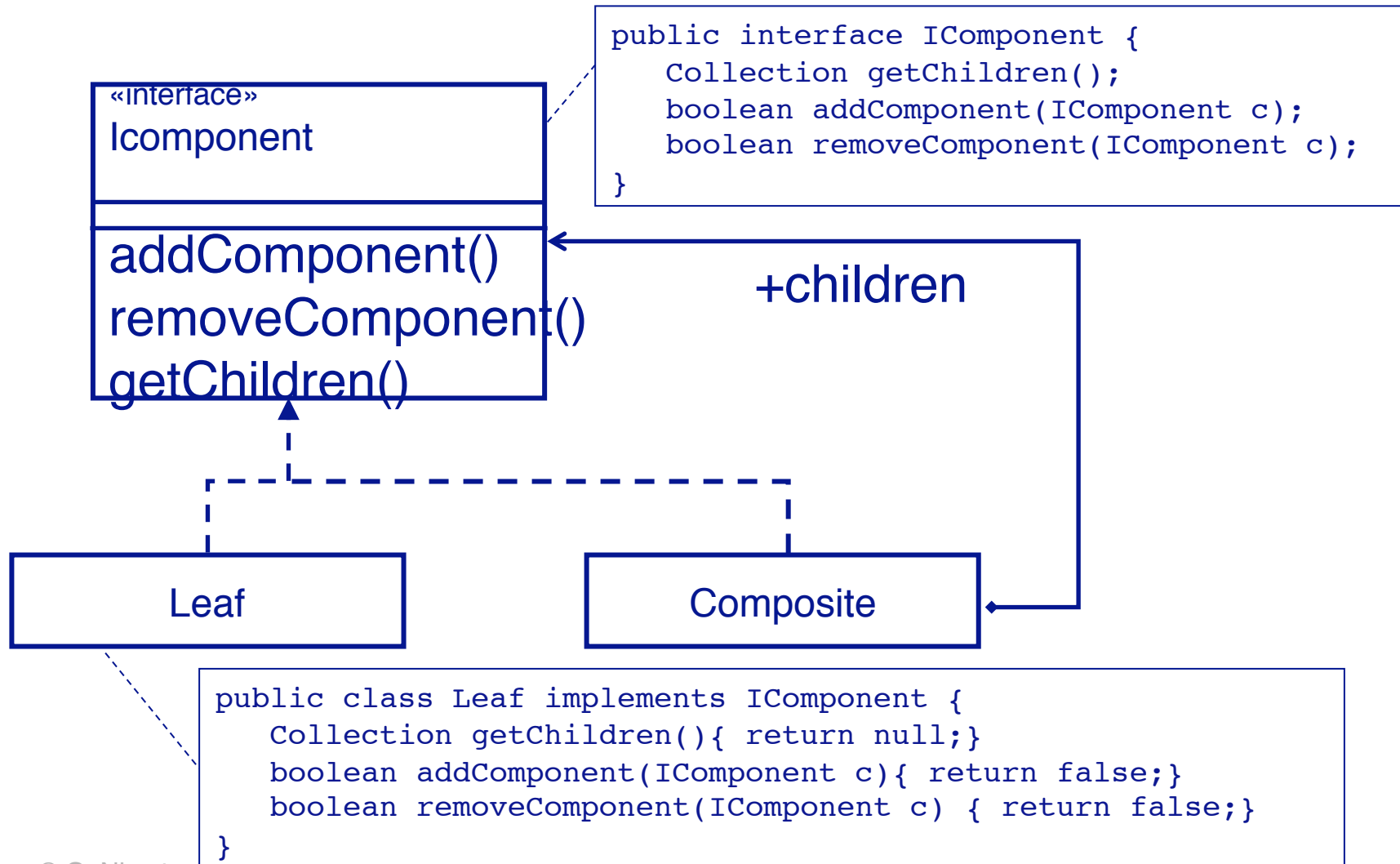
Typically composite objects will implement their behavior by *delegating to their parts.*

Composite Pattern Example

- > *Composite* allows you to treat a single instance of an object the same way as a *group* of objects.
- > Consider a *Tree*. It consists of Trees (subtrees) and *Leaf* objects.



Composite Pattern Example (2)




Composite Pattern Example (3)

```
public class Composite implements IComponent {
    private String id;
    private ArrayList<IComponent> list = new ArrayList<IComponent> ();
    public boolean addComponent(IComponent c) {
        return list.add(c);
    }
    public Collection getChildren() {
        return list;
    }
    public boolean removeComponent(IComponent c) {
        return list.remove(c);
    }
    ...
}
```

Composite Pattern Example — Client Usage (4)

```
public class CompositeClient {
    public static void main(String[] args) {
        Composite switzerland = new Composite("Switzerland");
        Leaf bern = new Leaf("Bern");
        Leaf zuerich = new Leaf("Zuerich");
        switzerland.addComponent(bern);
        switzerland.addComponent(zuerich);
        Composite europe = new Composite("Europe");
        europe.addComponent(switzerland);
        System.out.println(europe.toString());
    }
}
```

Observer Pattern

-  How can an object inform arbitrary clients when it changes state?
- ✓ *Clients implement a common Observer interface and register with the “observable” object; the object notifies its observers when it changes state.*

An observable object *publishes* state change events to its *subscribers*, who must implement a common interface for receiving notification.

Observer Pattern (2)


Example

- > ***See GUI Lecture***
- > A Button expects its observers to implement the ActionListener interface.
(see the Interface and Adapter examples)

Consequences

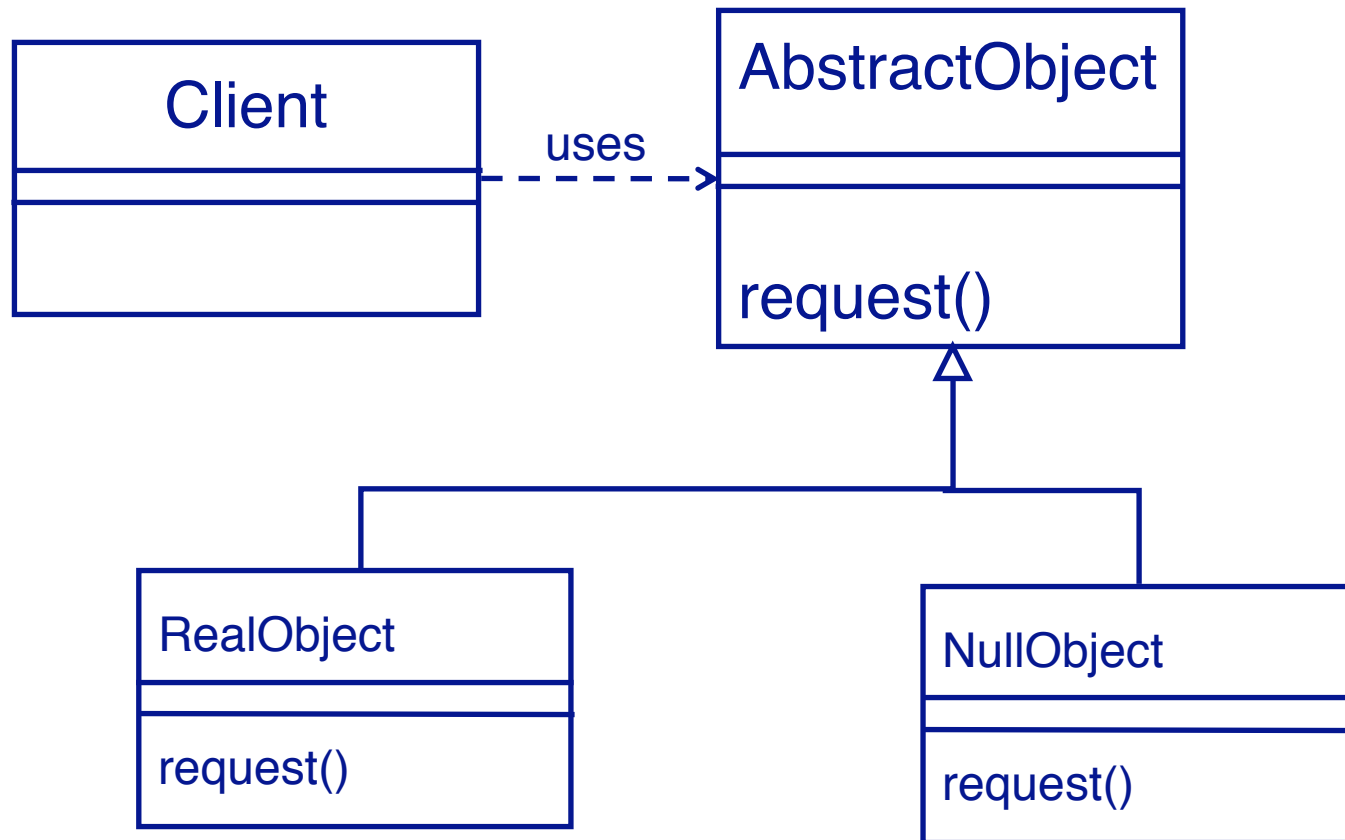
- > Notification can be *slow* if there are many observers for an observable, or if observers are themselves observable!

Null Object Pattern

-  How do you avoid cluttering your code with tests for null object pointers?
- ✓ *Introduce a Null Object that implements the interface you expect, but does nothing.*

Null Objects may also be Singleton objects, since you never need more than one instance.

Null Object Pattern – UML



Null Object

Examples

- > `NullOutputStream` extends `OutputStream` with an empty `write()` method

Consequences

- > Simplifies client code
- > Not worthwhile if there are only few and localized tests for null pointers








What Problems do Design Patterns Solve?

Patterns:






- > document *design experience*
- > enable widespread *reuse of software architecture*
- > *improve communication* within and across software development teams
- > *explicitly capture knowledge* that experienced developers already understand implicitly
- > arise from *practical experience*
- > help *ease the transition* to object-oriented technology
- > *facilitate training* of new developers
- > help to transcend “programming language-centric” viewpoints

Doug Schmidt, CACM Oct 1995

What you should know!

-  *What's wrong with **long methods**? How long should a method be?*
-  *What's the difference between a **pattern** and an **idiom**?*
-  *When should you use **delegation** instead of **inheritance**?*
-  *When should you call "**super**"?*
-  *How does a **Proxy** differ from an **Adapter**?*
-  *How can a **Template Method** help to eliminate duplicated code?*
-  *When do I use a **Composite Pattern**? Do you know any examples from the Frameworks you know?*

Can you answer these questions?

-  *What **idioms** do you regularly use when you program?
What patterns do you use?*
-  *What is the difference between an **interface** and an **abstract class**?*
-  *When should you use an **Adapter** instead of modifying the interface that doesn't fit?*
-  *Is it good or bad that `java.awt.Component` is an abstract class and not an interface?*
-  *Why do the Java libraries use different interfaces for the Observer pattern (`java.util.Observer`, `java.awt.event.ActionListener` etc.)?*

License

<http://creativecommons.org/licenses/by-sa/2.5/>



You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:



Attribution. You must attribute the work in the manner specified by the author or licensor.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.