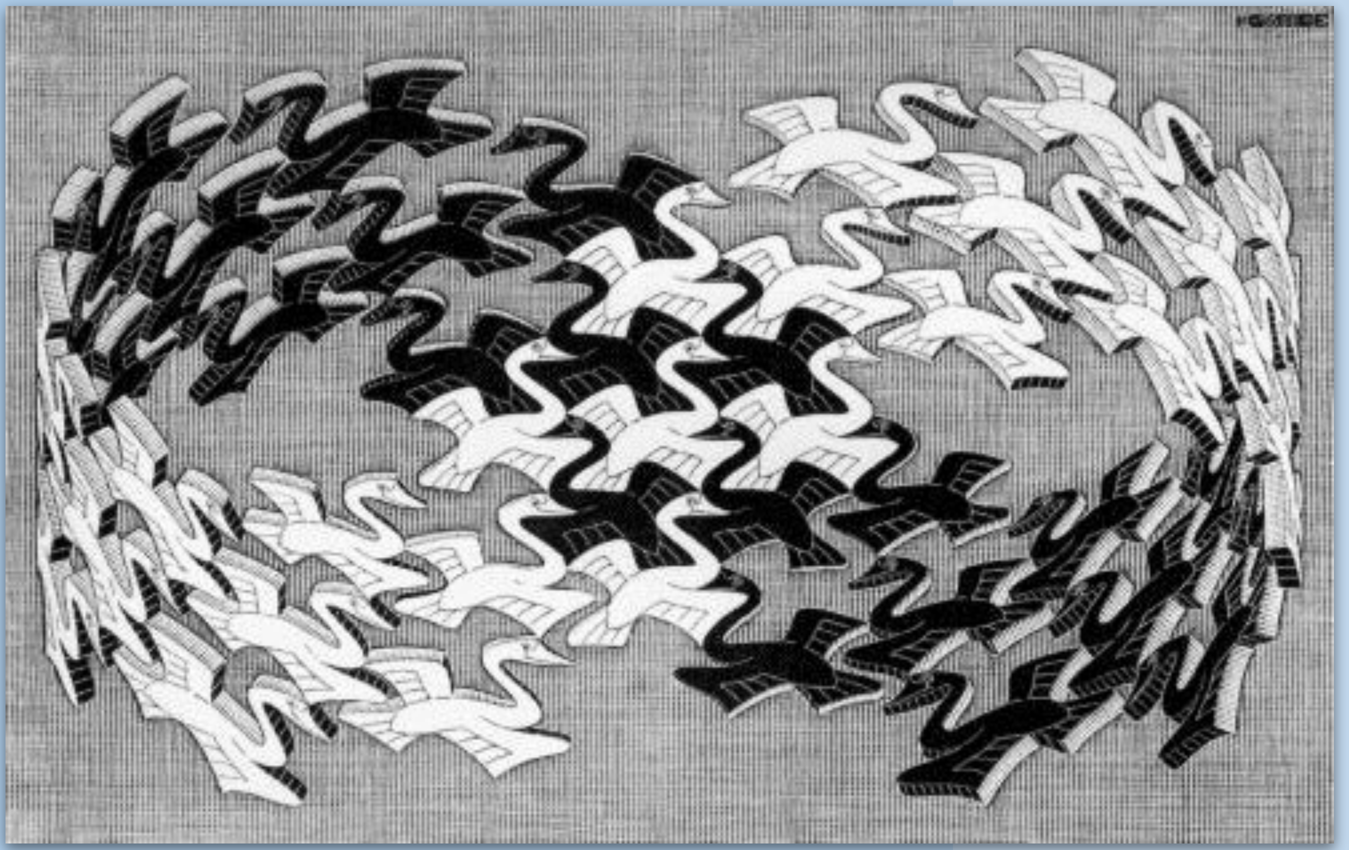
10. Guidelines, Idioms and Patterns

Oscar Nierstrasz



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

The “Design Patterns” book by Gamma et al. (also known as the “Gang of Four”, or GOF) was the first book to systematically document design patterns. The book describes patterns encountered by the four authors during their experience working on various projects in different object-oriented languages. The patterns are language-agnostic, and discuss trade-offs in different languages.

Many other authors have documented other design patterns since then, both in books and in conferences, particularly the Pattern Languages of Programs (PLoP) series. There exist also books dedicated to how design patterns can be implemented in specific languages, like Java.

Design patterns are *common solutions to recurring design problems*.

Design patterns are not just about structure, they also cover interaction between objects — *communication patterns*.

Also deal with *strategies for inheritance and containment.*

The GOF book presents three groups of patterns:

Creational patterns create objects for you. You can decide which objects are created given a specific case.

Structural patterns help you to compose objects into larger structures.

Behavioral patterns help you to define communication between objects and how the flow is controlled in a complex program.

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

Simplicity in design should be a key goal (used in the Apollo project). Simplicity denotes beauty purity and clarity. Or if code speaks and no one is around to hear it, does it still make a sound?

Extreme Programming encourages starting with the simplest solution and refactoring to better ones. The difference between this approach and more conventional system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month. Proponents of XP acknowledge the disadvantage that this can sometimes entail more effort tomorrow to change the system; their claim is that this is more than compensated for by the advantage of not investing in possible future requirements that might change before they become relevant. Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed.

Related to the "communication" value, simplicity in design and coding should improve the (quality of) communication. A simple design with very simple code could be easily understood by most programmers in the team.

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 ...

Object-oriented programs live best and longest with *short methods*. The payoffs of indirection — explanation, sharing and choosing — are supported by little methods. Everybody knows short is good.

99% of the time, just use Extract Method to shorten a method. Find a part that goes together and make a new method.

Parameters and temporary variables get in the way of extracting methods. Use Replace Temp with Query to remove the temporaries and Introduce Parameter Object or Preserve Whole Object to slim down the parameter lists.

If you still have too many temps and parameters, it's time for the heavy artillery: Replace Method with Method Object

Look for comments. A block of code with a comment should be replaced by a method named to match the comment. *Even a single line is worth extracting if it needs explanation.*

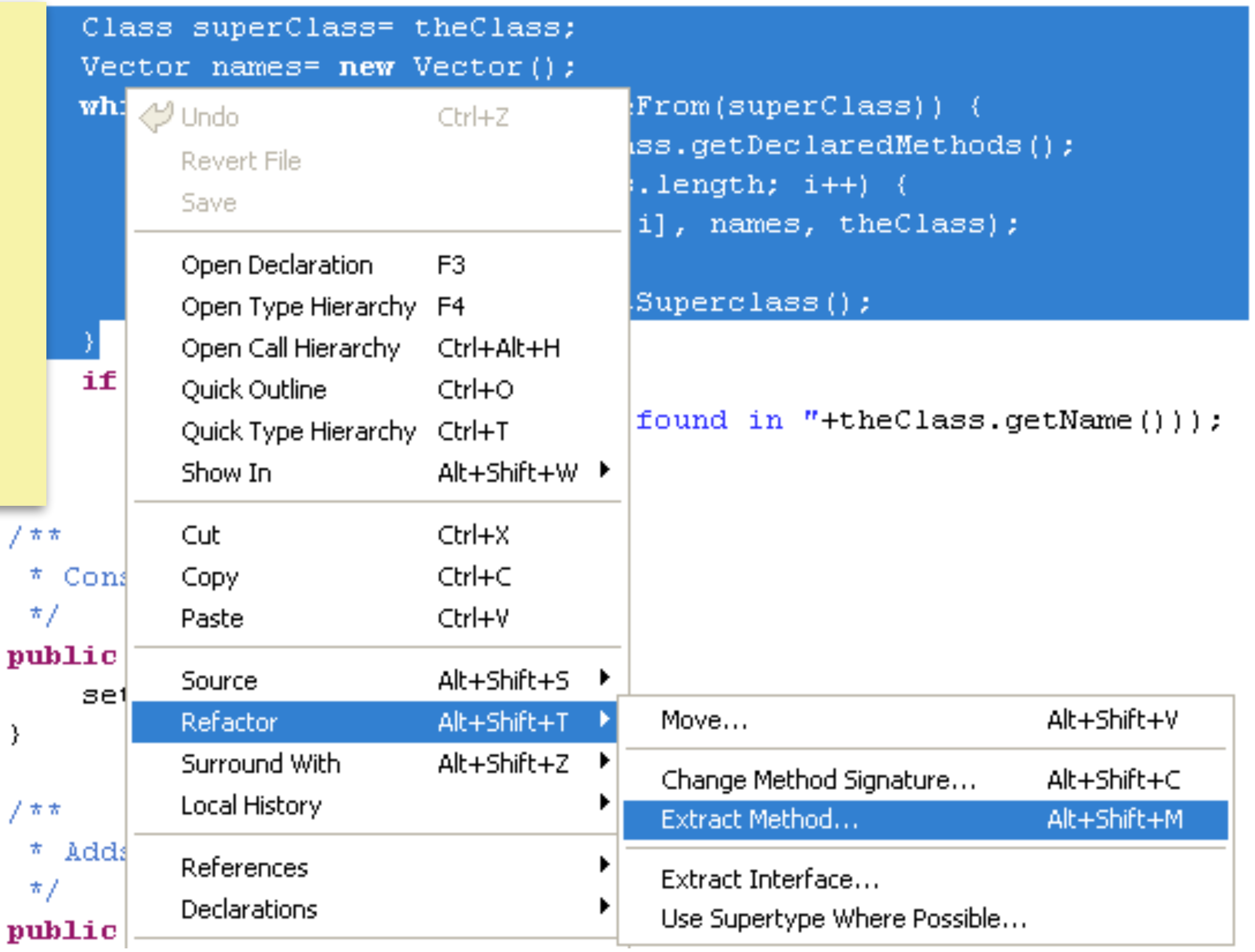
Look for conditionals and loops. Use Decompose Conditional. Extract each loop into its own method.

Refactoring Long Methods

Short is good!

If you need to comment then Extract as Method.

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



```
Class superClass= theClass;
Vector names= new Vector();
wh                                     From(superClass)) {
                                      ss.getDeclaredMethods();
                                      .length; i++) {
                                       i], names, theClass);

                                      Superclass();
}
if
                                      found in "+theClass.getName()));

/**
 * Cons
 */
public
    se
}

/**
 * Adds
 */
public
```

What are Idioms and Patterns?

Idioms	Idioms are *common programming techniques* and conventions. They are often language-specific. (http://c2.com/ppr/wiki/JavaIdioms/JavaIdioms.html)
Design Patterns	Patterns document *common solutions to design problems*. They are language-independent.
Libraries	Libraries are *collections of functions, procedures or other software components* that can be used in many applications.
Frameworks	Frameworks are open libraries that define the *generic architecture* 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.

Idiom: the peculiar character or genius of a language — a distinct style or character, in music, art, e.g., the idiom of Bach.

E.g., Interface in Java

A Library is about reusable behavior, a framework is about reusable architecture.

A library is something you call or inherit from your code.

A framework is something that calls your code to provide services for your code.

A framework is an abstract design that embodies how an application works and it has hooks where you can inject your module or component.

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 is the handing of a task to another part of the program. One object, the *delegator*, defers a task to another object, known as the *delegate*.

Delegation is used to separate concerns and distribute responsibilities. The opposite of delegation is concentration of responsibilities (as seen in the *God class* anti-pattern).

Some programming languages, like JavaScript (JS), are fully based on delegation. JS is based on prototypes rather than classes: to create a new object, you clone an existing object (a *prototype*) instead of instantiating a class. Class-based inheritance is simulated by *delegation*: if an object does not understand a message (i.e., a request) it delegates it to its prototype (and so on).

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

Instead of a test suite directly running tests, it delegates the task to its individual test cases. The form of delegation is also fundamental to the *Composite pattern* (discussed later): when a composite object is asked to perform a task, it typically delegates the tasks to its components. Here the test suite is the composite and the test cases are the components.

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.

Instead of *replacing* behavior from the superclass, you would like to *extend* it. The `super` construct allows you to override the inherited method and extend it by invoking that behavior by calling `super`. The extended behavior will occur *before*, *after* or *around* the old behavior.

```
public T foo() {
     T result;
     // new behavior before
     result = super.foo();
     // new behavior after
     return result;
}
```

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!

NB: It is bad practice to use `super` to invoke an inherited method when this will do. Consider the following classes `A` and `B`:

```
class A {
  void m1() { ... }
}
class B extends A {
  void m2() { ... super.m1(); ... } // should use this.m1()
}
```

Now, if we later override m1() in B, then m2 will call the wrong method!

```
class B extends A {
  void m2() { ... super.m1(); ... } // calls wrong m1() !
  void m1() { ... } // overrides A.m1()
}
```

The correct thing to do is to call `this.m1()`.

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);
   }
```

In the first example we extend the behavior of the `paint()` method with our specific painting behavior. In general, whenever you override a method inherited from a framework, *it is good practice to explicitly call the overridden method using `super`*. If you don't then you may break the framework behavior. Here, if you fail to call `super.paint()`, nothing will work.

In the second example, we explicitly call the constructor of the superclass. By default classes in Java only inherit the default constructor of the superclass.

NB: Subclass constructors should *always* explicitly call a constructor of the superclass to ensure that the invariant of inherited state is established.

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.

Interfaces reduce coupling by removing the need to inherit from a specific class. If you declare a variable `x` to be of type `A`, where `A` is a class, then `x` *must* be an instance of `A` or one of its subclasses. If, on the other hand, you declare `x` to be of type I, where I is an interface, then `x` can be an instance of *any* class `X` that implements the interface `X`. In the latter case, you reduce coupling between the class where `x` is declared and other class hierarchies.

On the other hand, complexity is increased by adding a new interface `I` to the system. If the flexibility offered by the interface is not needed, it is pointless to add it.

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

This is one of the most basic and common design patterns.

By defining an adapter around an existing class, you *avoid the need to modify directly the interface* of that class. Sometimes you may not even have access to the source code of the class to be adapted, or there may be a great deal of existing code that depends on the old interface. Writing an adapter may be the only way to avoid rewriting the source code of the adapted class.

The down side is that the adapter introduces an *extra level of indirection*. Every access to the old class must go through the adapter, possibly impacting performance.

Adapter Pattern

Examples

> A `WrappedStack` adapts `java.util.Stack`, throwing an `AssertionException` when `top()` or `pop()` are called on an empty stack.

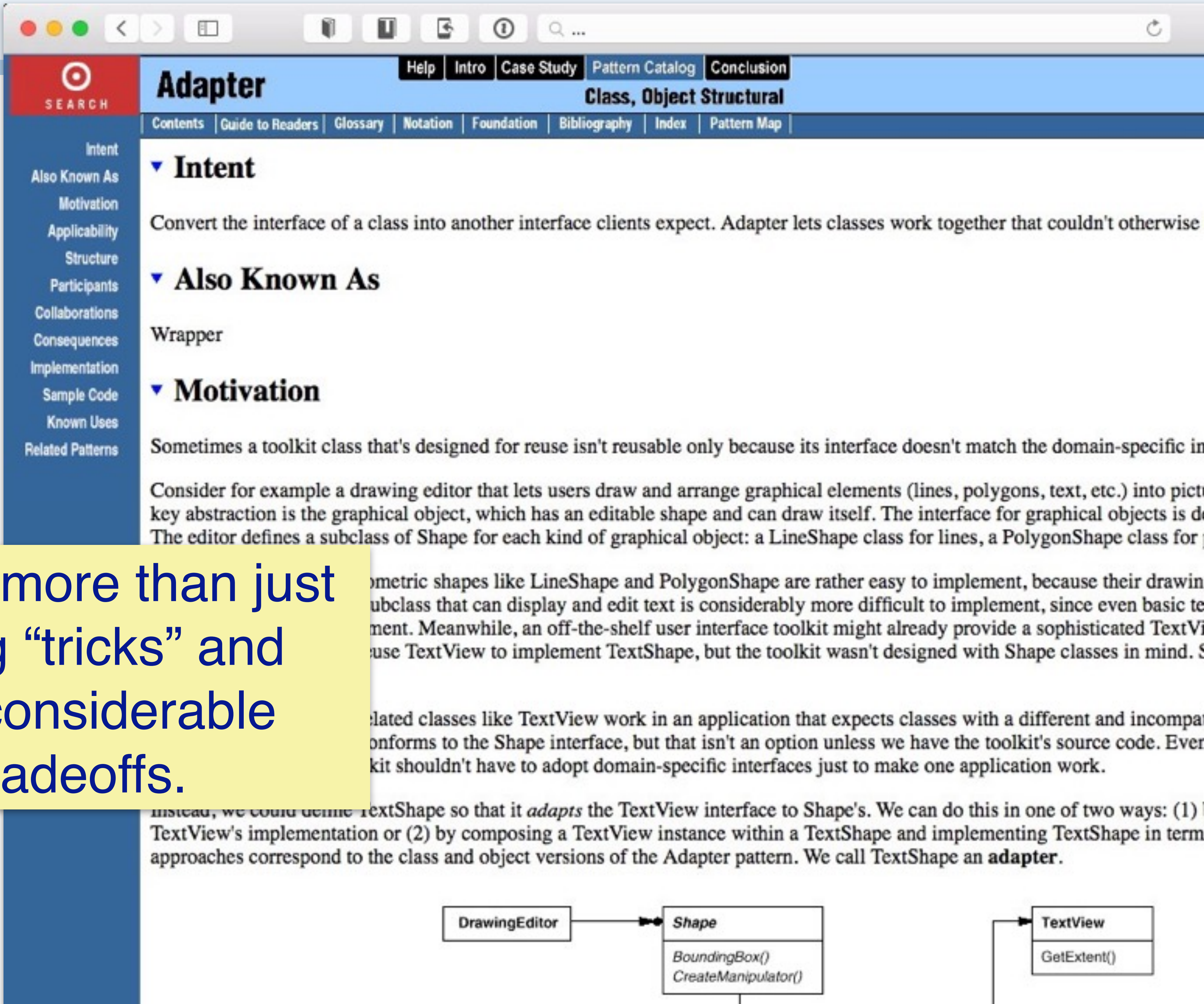
> An `ActionListener` converts a call to `actionPerformed()` to the desired handler method.

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

Pattern description format



Adapter

Class, Object Structural

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise

Also Known As

Wrapper

Motivation

Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific in

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pict
key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is d
The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for

ometric shapes like LineShape and PolygonShape are rather easy to implement, because their drawin
ubclass that can display and edit text is considerably more difficult to implement, since even basic te
ment. Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextVi
use TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind.

lated classes like TextView work in an application that expects classes with a different and incompa
onforms to the Shape interface, but that isn't an option unless we have the toolkit's source code. Eve
kit shouldn't have to adopt domain-specific interfaces just to make one application work.

Instead, we could define TextShape so that it *adapts* the TextView interface to Shape's. We can do this in one of two ways: (1)
TextView's implementation or (2) by composing a TextView instance within a TextShape and implementing TextShape in term
approaches correspond to the class and object versions of the Adapter pattern. We call TextShape an **adapter**.

Patterns are more than just programming “tricks” and can involve considerable details and tradeoffs.

The basic idea of any given design pattern may be quite simple, but to fully discuss the pros and cons, implementation strategies and other tradeoffs may require several pages of documentation.

Each of the design patterns in the original GOF book increases flexibility of your code at the expense of increased complexity. Understanding the tradeoffs is critical to applying design patterns judiciously.

Erich Gamma has a story of a design pattern "fan" who proudly told him that in his last project he successfully applied all the design patterns in the book. Erich, of course, was horrified.

Proxy Pattern

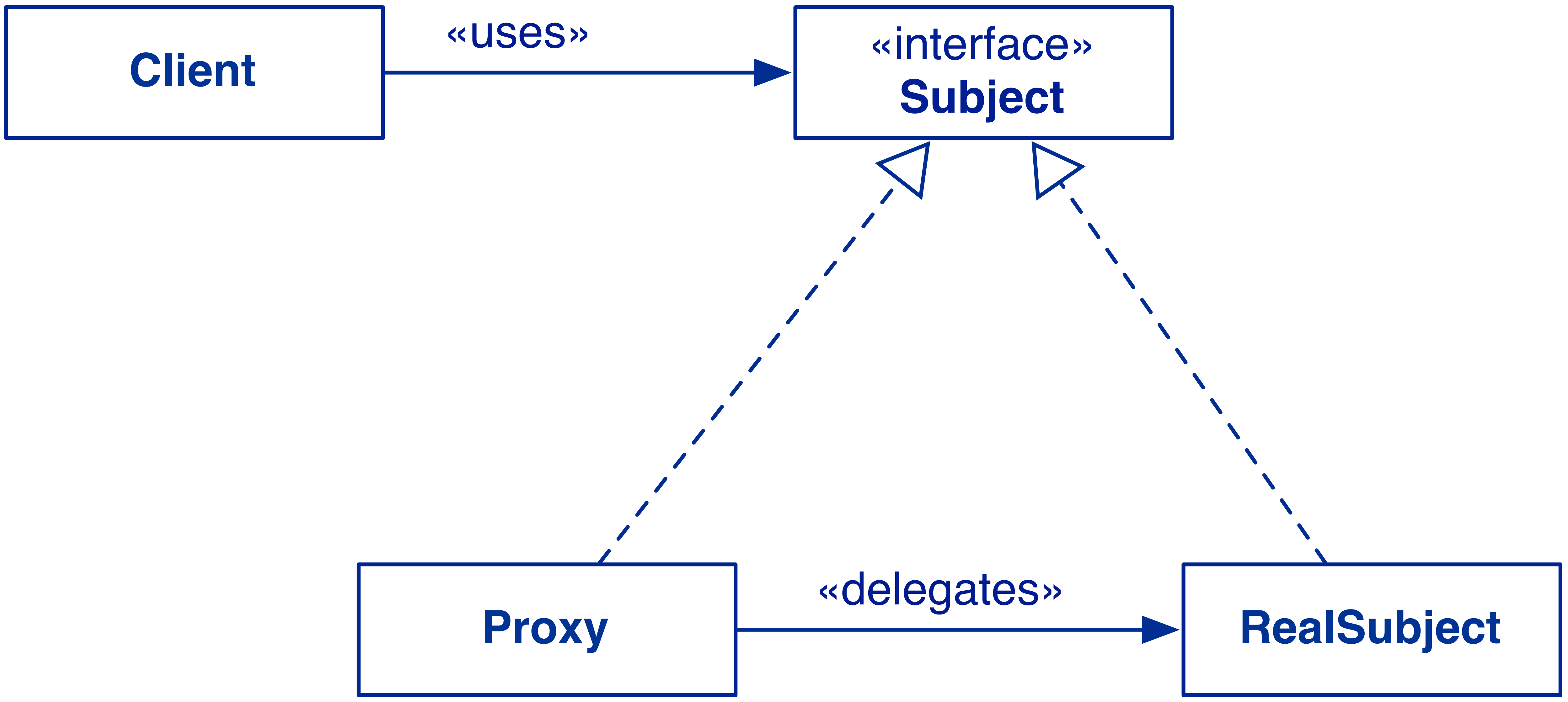
✎ How do you hide the complexity of accessing objects that require pre- or post-processing?

✔ *Introduce a proxy to control access to the object.*

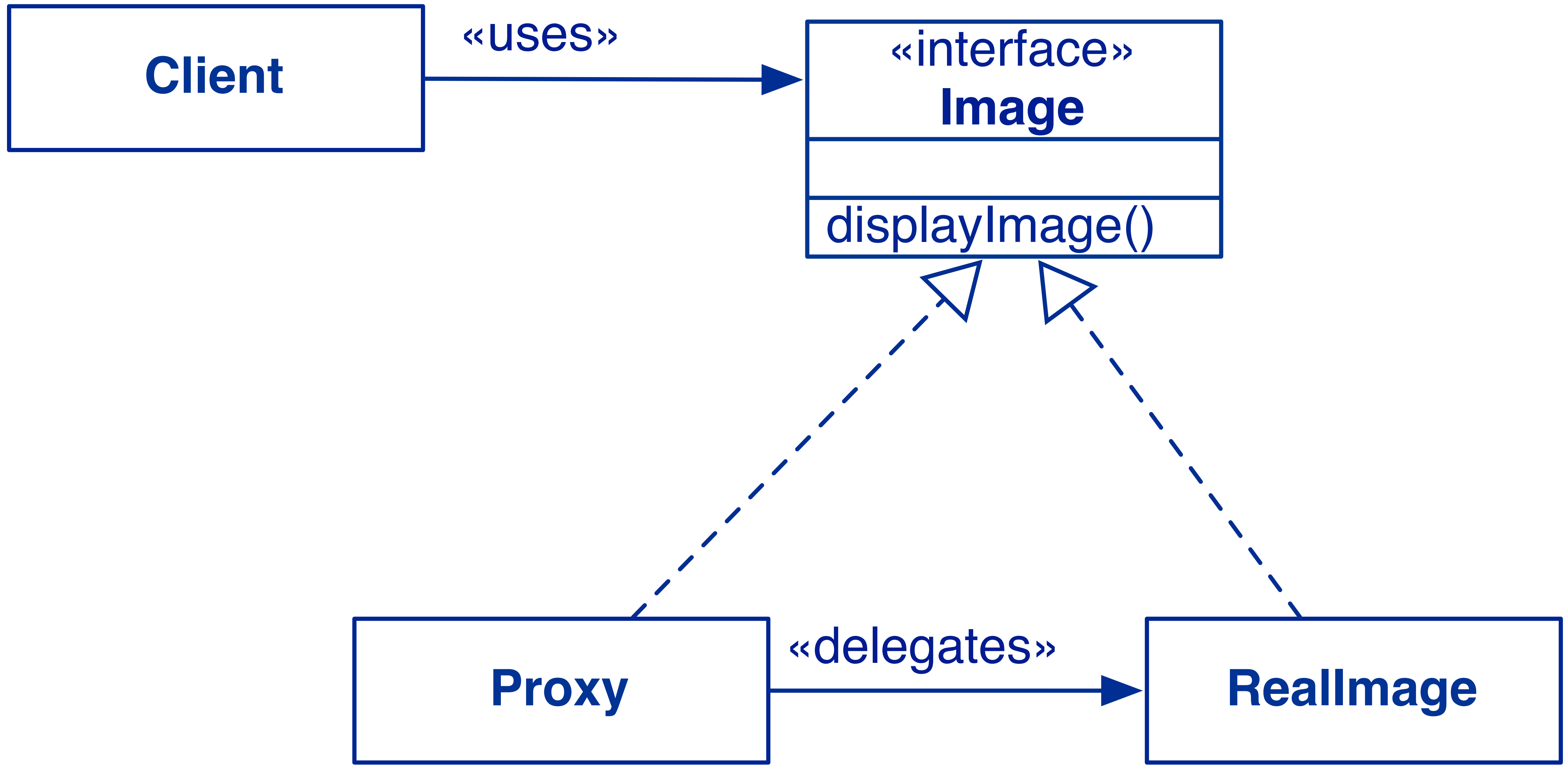
Some services require special pre or post-processing. Examples include objects that reside on a remote machine, and those with security restrictions.

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

Proxy Pattern — UML



Proxy Pattern Example



Proxy Pattern Example

delegate request to real subject

```
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();
   }
}
```

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.

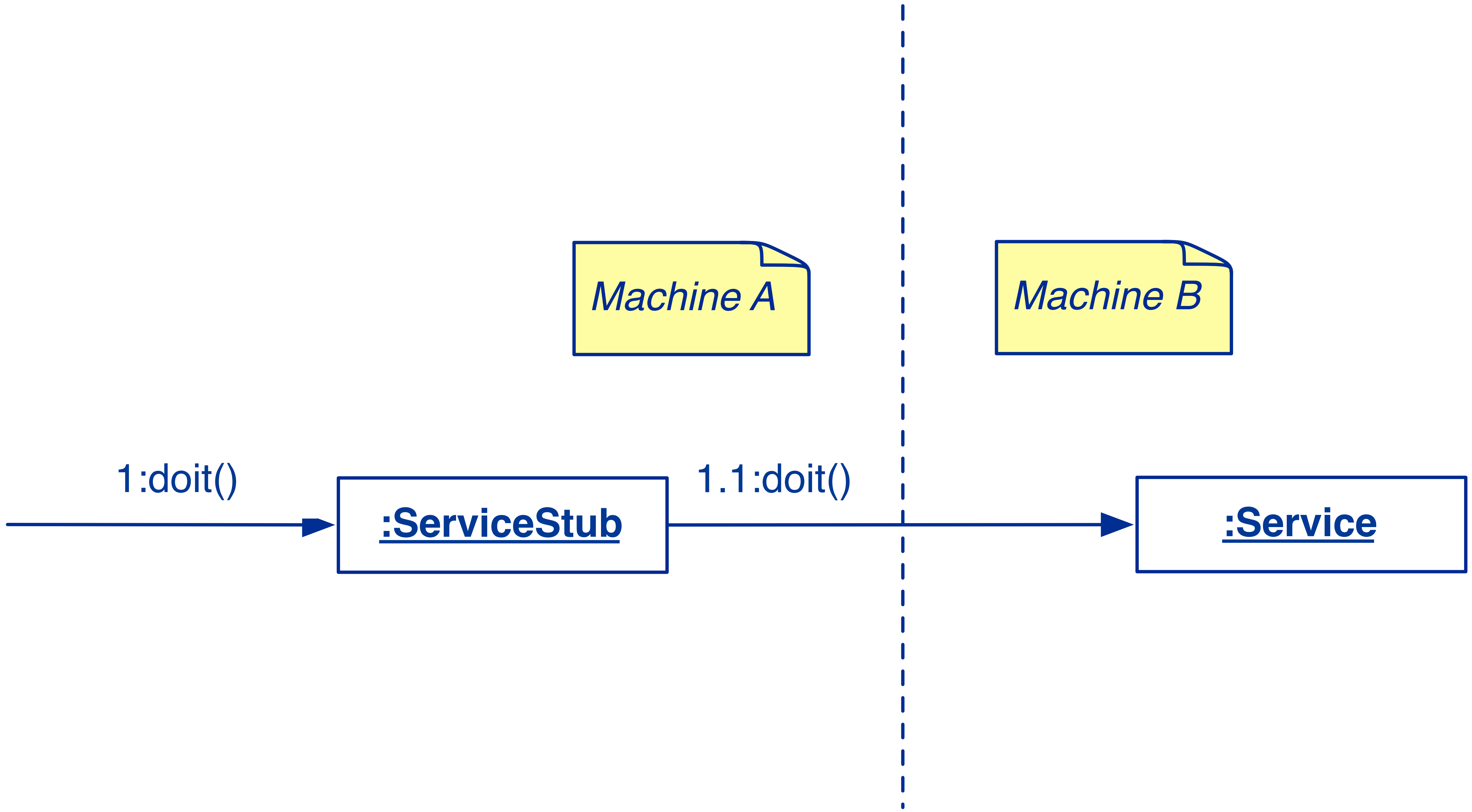
Proxy differs from Adapter in that it does not change the object’s interface.

Superficially proxies look very much like adapters. Both wrap an existing class, adding a level of indirection. Their *intent*, and consequently the design considerations and tradeoffs are very different, however.

An *adapter* exists only to adapt the interface of a class to a different interface required a by a client.

A *proxy*, on the other hand, provides the *same interface* as the object it is a proxy for. Instead of adapting the interface, it provides additional pre- or post-processing.

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*.

This is a behavioral pattern.

It defines a program skeleton of an algorithm. The algorithm itself is made abstract, and the subclasses of the method override the abstract methods to provide concrete behavior.

First a class is created that provides the basic steps of the algorithm design.

These steps are implemented using abstract methods. Later on the subclasses change the abstract methods to perform real actions. Thus, the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.

Template Method Pattern

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 classes calls the operations of a subclass and not the other way around.

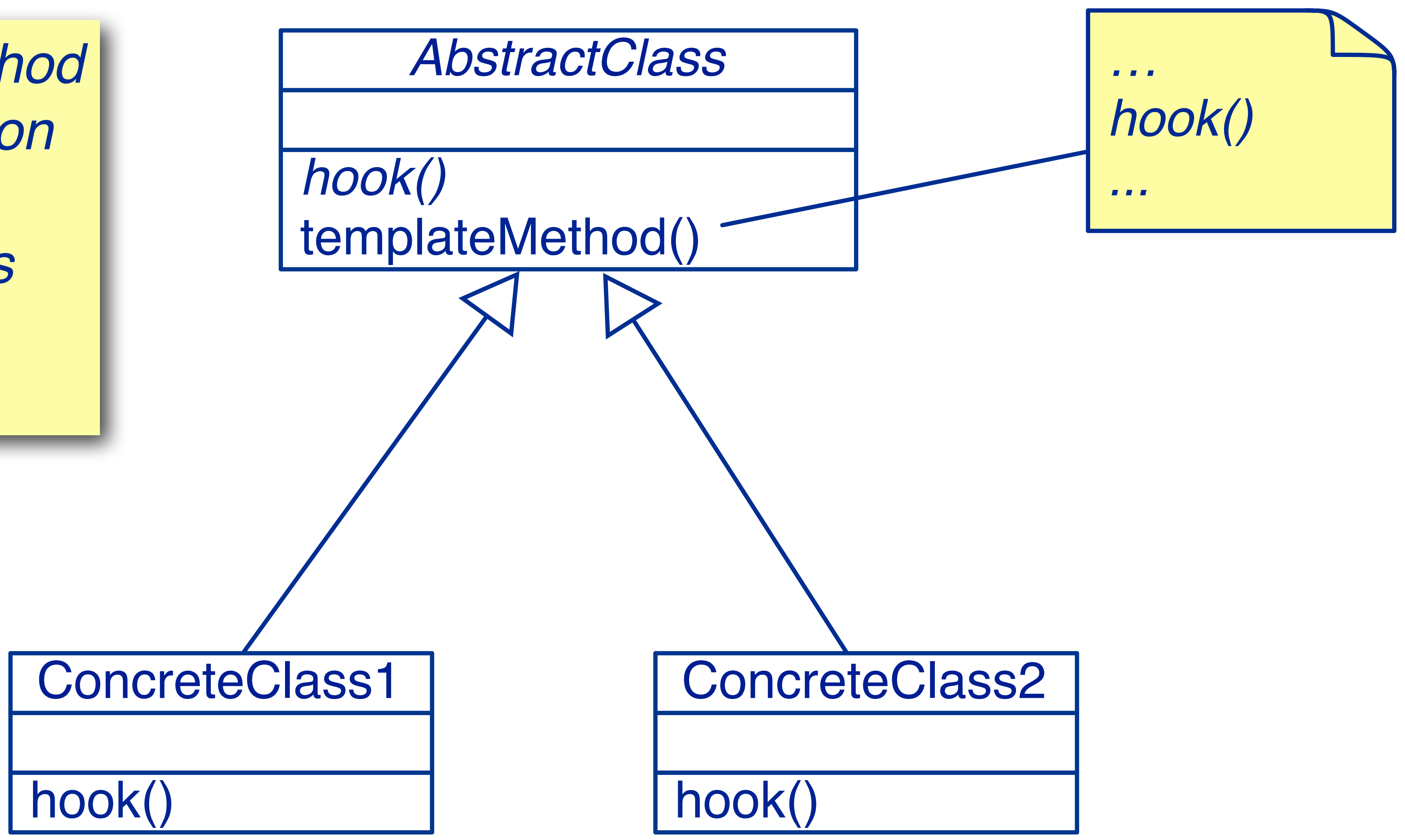
Template Method is used in most frameworks to allow application programmers to easily extend the functionality of framework classes.

The Template Method pattern is fundamental to object-oriented frameworks, such as the original JUnit. Such a framework defines the *architecture* of a generic application in terms of interacting objects, but leaves open the details of the work to be done by a specific application.

The generic behaviour of the framework is defined in template methods, such as the methods that actually run tests and gather the test results. The details are deferred to hook methods, containing the actual tests. These are implemented by the specific application classes that inherit from framework classes, like `TestCase`.

Template Method Pattern — UML

The template method defines the skeleton of an algorithm. Concrete methods override the hook methods.



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.*

When working with tree-structured data, a programmer has to discriminate between a leaf node and a branch. This makes code more complex, thus error-prone.

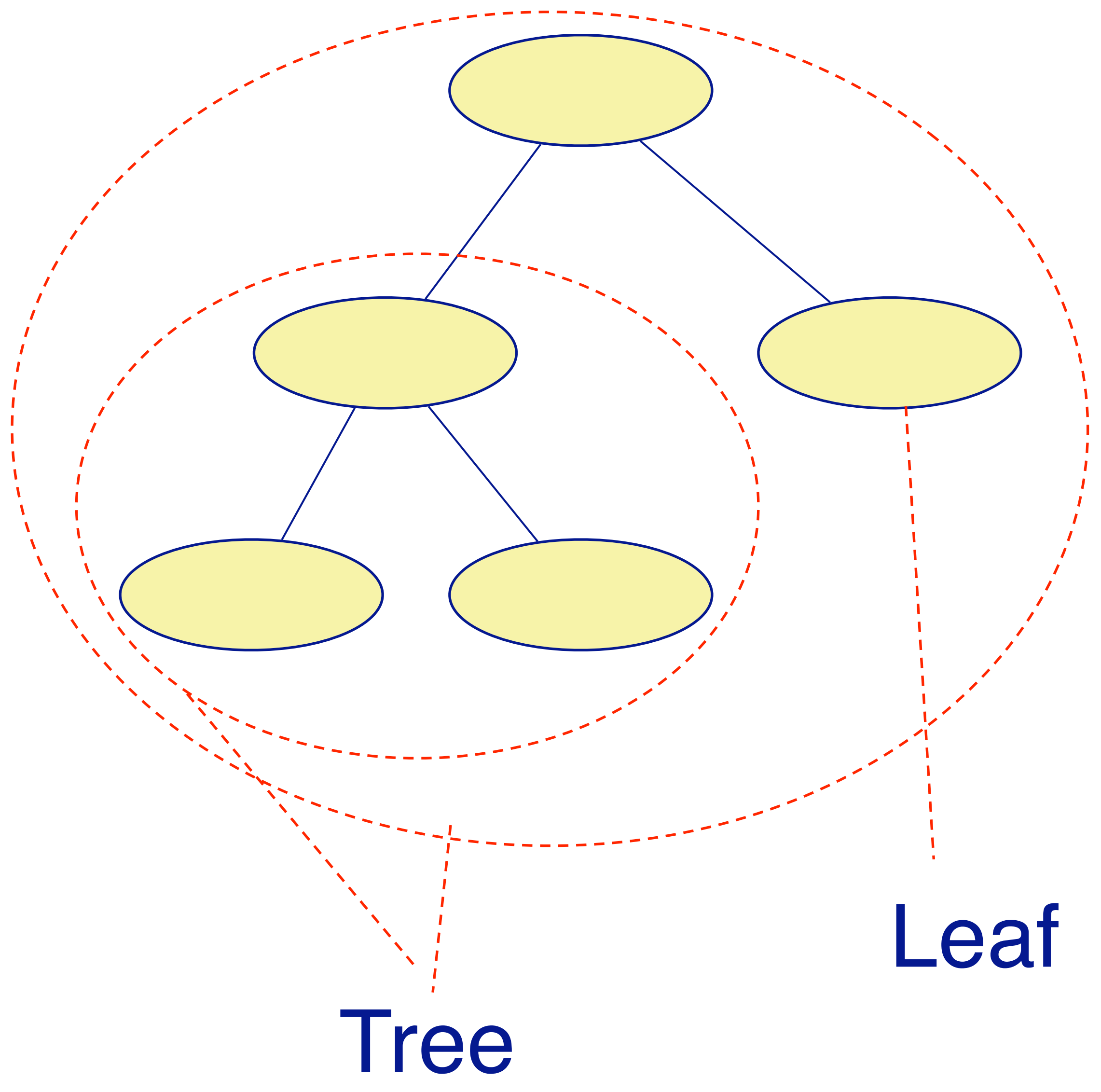
A Composite is an object (e.g. a shape) designed as a composition of one or more similar objects (other kinds of shapes) exhibiting similar functionality.

This is known as a *has-a* relationship between objects.

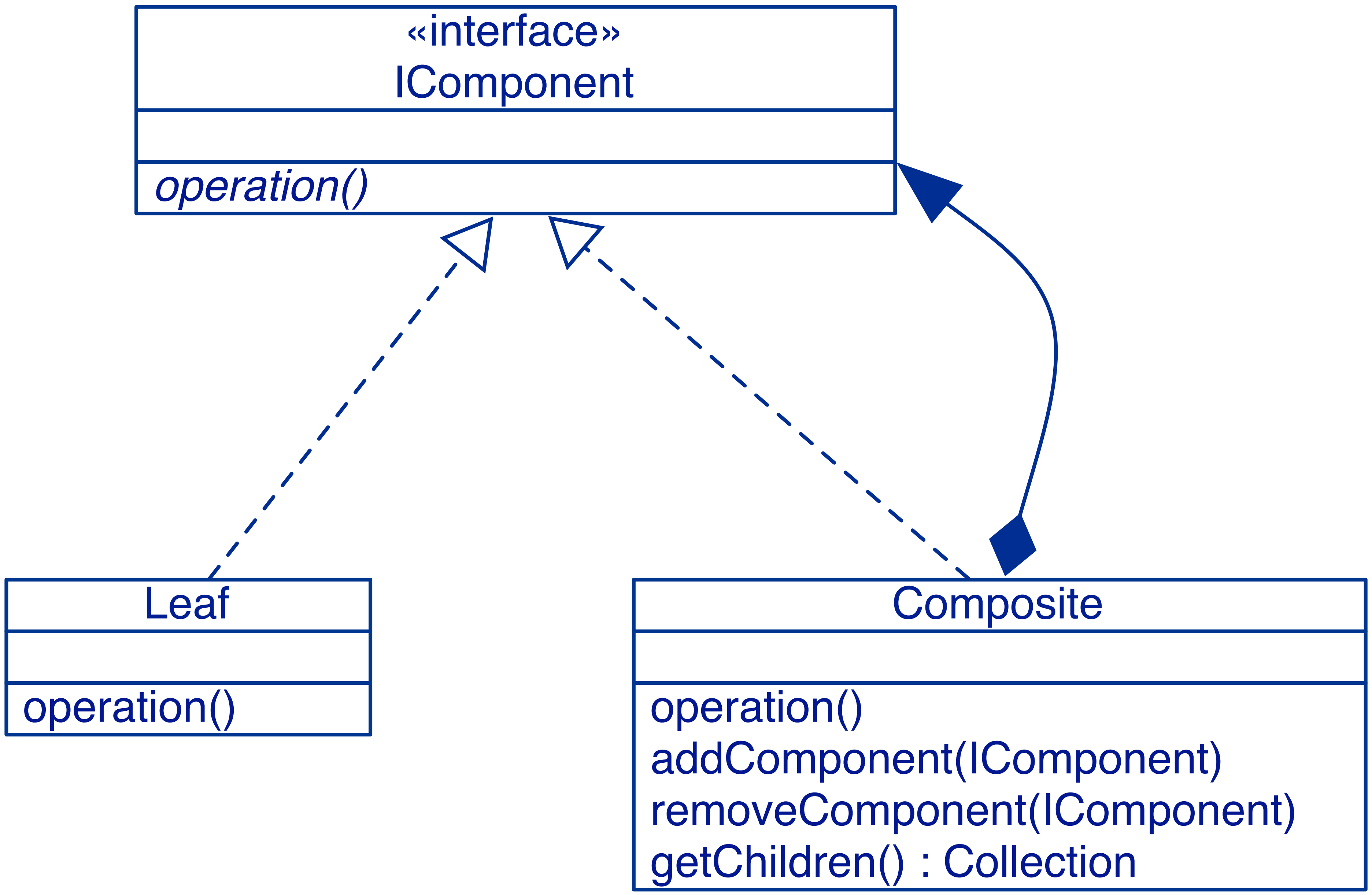
The key concept is that you can manipulate a single object just as you would a group of them.

Composite Pattern

- *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.

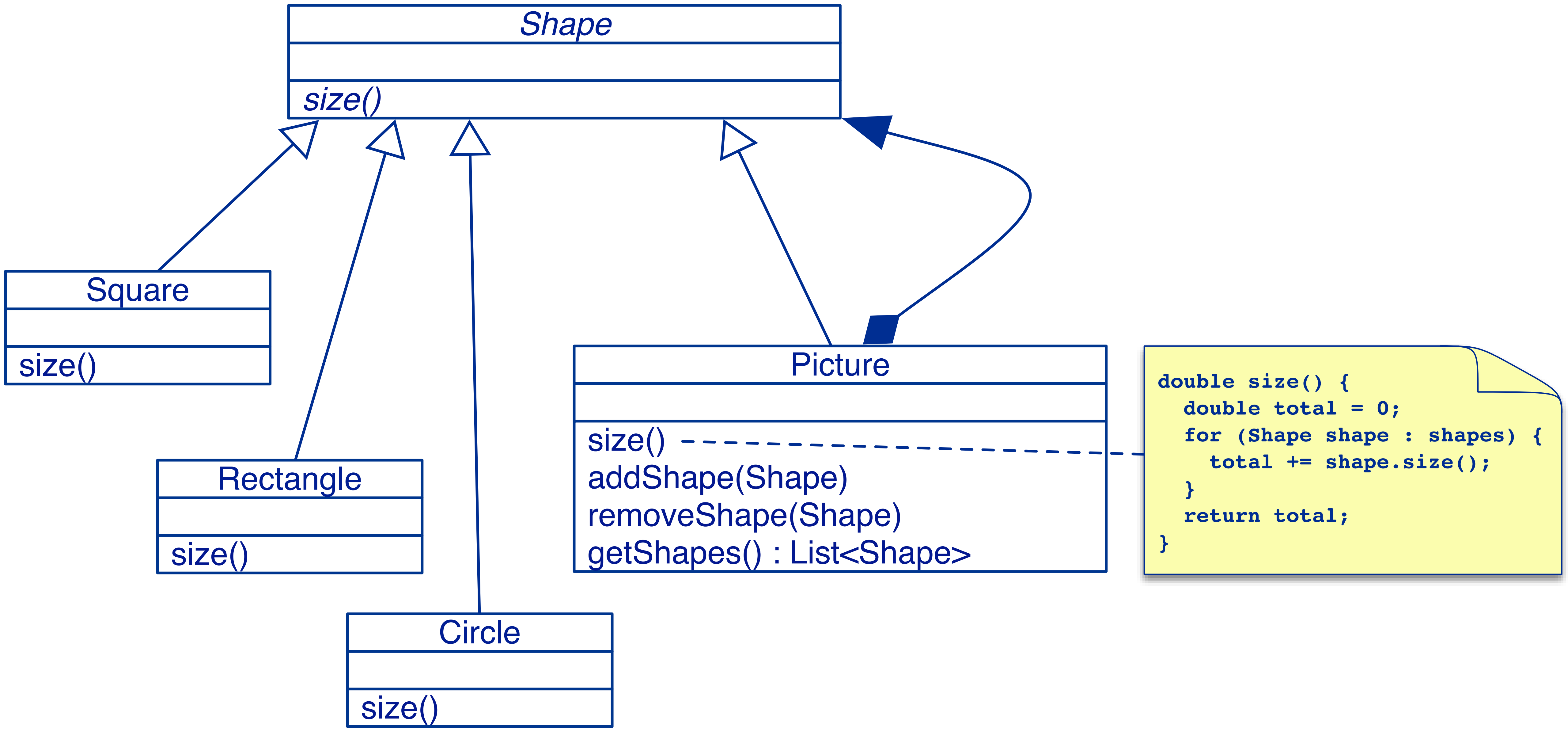


Composite Pattern — UML



Here we see the essence of the Composite pattern: both Leaf entities and Composites implement the same interface (possibly inheriting from a common superclass). The Composite is furthermore composed of entities that implement that same interface, i.e., the components may themselves be either leaves or composites.

Composite Pattern — Example



```
double size() {
  double total = 0;
  for (Shape shape : shapes) {
    total += shape.size();
  }
  return total;
}
```

Recall the Shapes example from the first lecture. We could easily redesign it to make a Picture a Composite.

Here we see one of the key implementation considerations of the Composite pattern: operations defined in the common interface are often implemented in the composite by invoking those same operations in the individual components (i.e., possibly recursively), and combining their results. In this example, the sizes of the components are added together. In the JUnit example, tests are executed recursively and their test results are combined into an overall test report.

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.

We saw the Observer pattern extensively in the GUI lecture.

Observers can, however, be used in any setting where parts of an application need to be informed about events in another part.

Consider an online shopping platform where users want to be informed if an items becomes available below a certain price, or sellers want to be informed if bids are issues for similar items that they are also selling. Such cases can be effectively handled using Observers, even though the domain has nothing to do with user interfaces.

Observer Pattern

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?

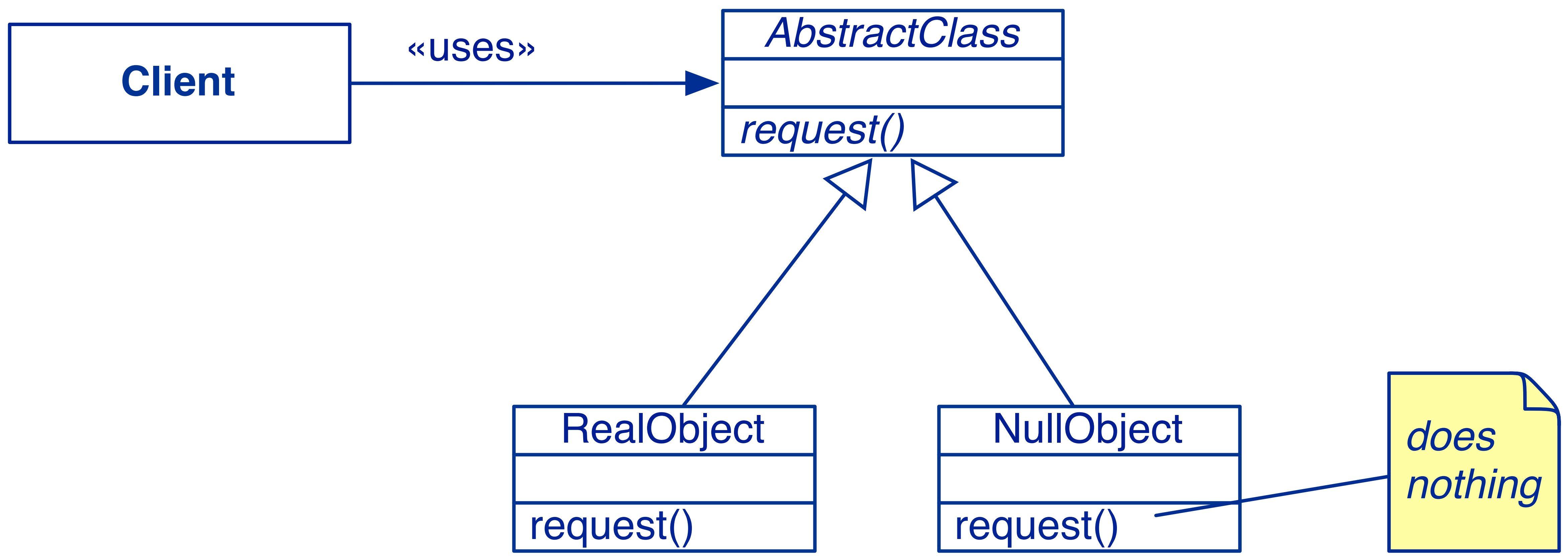
✔ *Introduce a Null Object that implements the interface you expect, but does nothing.*

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

The Null Object provides intelligent “do nothing” behavior.

It shifts the responsibility for “doing nothing” from the client to the supplier. Instead of having to actively check whether an object is a valid target for some action, we simply ensure that all objects support that action, even if it is a no-op. This considerably simplifies our code and avoids accidental “null object reference” errors.

Null Object Pattern — UML



The `Client` requires a collaborator.

`AbstractObject` declares the interface for `Client`’s collaborator. It implements default behavior for the interface common to all classes, as appropriate.

`RealObject` defines a concrete subclass of `AbstractObject` whose instances provide useful behavior that `Client` expects.

`NullObject` provides an interface identical to `AbstractObject` so that a null object can be substituted for a real object. It implements its interface to do nothing. (NB: what exactly it means to “do nothing” depends on the behavior `Client` is expecting.)

When there is more than one way to do nothing, more than one `NullObject` class may be required.

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

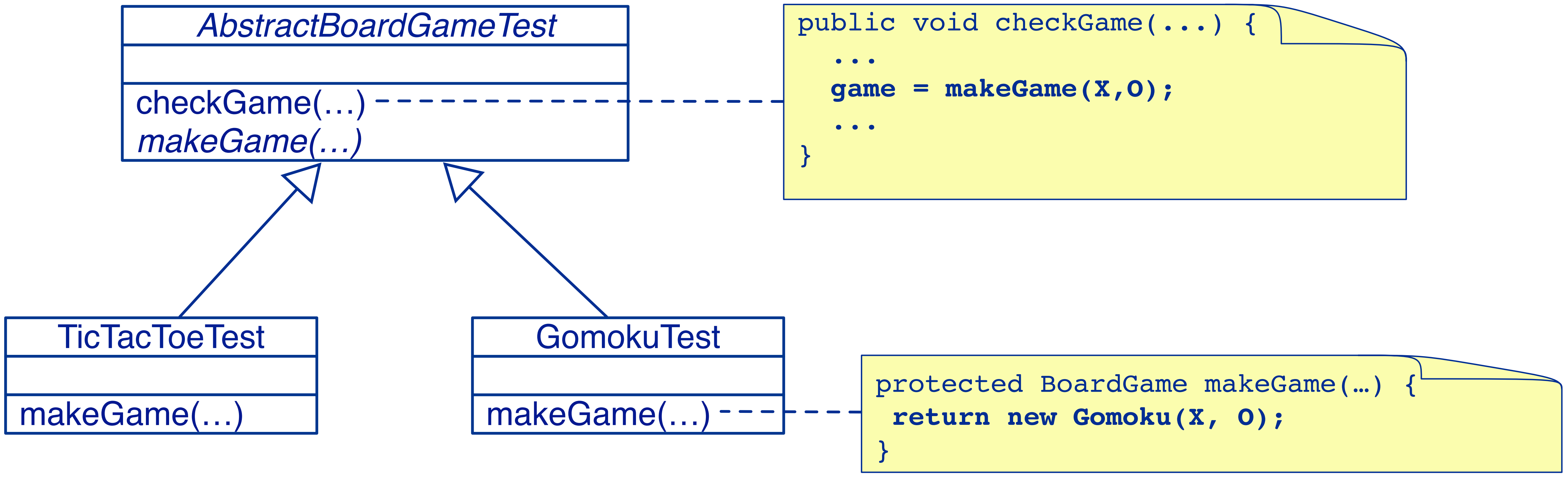
Factory Method

✎ How can you write code that abstracts which classes to instantiate?

✔ *Define an interface for creating an object, but let subclasses decide which class to instantiate.*

A Factory method lets a class defer instantiation to subclasses.

Factory Method — UML



```
public void checkGame(...) {
   ...
   game = makeGame(X,O);
   ...
}
```

```
protected BoardGame makeGame(…) {
 return new Gomoku(X, O);
}
```

Factory Method

Examples

> `AbstractBoardGameTest` defines an abstract factory method `makeGame()` that is implemented in its subclasses

Consequences

> Abstracts away from which game to test

Factory methods are often combined with *AbstractFactories* to encapsulate creation of related objects.

Factory methods also reduce coupling by avoiding the need to explicitly name the classes to be instantiated in the host code. This can make your code much more easily configurable.

Abstract Factories take this idea further and offer a general interface for creating objects. Different factories can then be easily plugged into an application to configure it for one platform or another. For example, dedicated factories may create UI widgets for one operating system or another using a common interface.

Some other Design Patterns…

State	The state pattern is a behavioral design pattern, also known as the objects for states pattern. This pattern is used in to represent the state of an object. This is a clean way for an object to partially change its type at runtime.
Decorator	that allows new/additional behaviour to be added to an existing method of an object dynamically.
Visitor	a way of separating an algorithm from an object structure. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.

and many more…

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.)?*