

Design Patterns

Design Patterns

A programming pattern that occurs frequently

General: can be applied in many situations

Patterns describe the shape of code rather than the details

Lots of them in CSC 301 and 302

Design Pattern Categories

Creational

Singleton, Abstract Factory, Prototype

Behavioural

Iterator, Visitor

Structural

Composite, Adapter

UML

Unified Modelling Language

A way to draw information about program design

Boxes represent classes

Contents of boxes represent variables and methods

The pictures we show here come from

<http://www.dofactory.com/Patterns/Patterns.aspx>

UML not on final exam!

Singleton Pattern

Sometimes want only one object of a particular class in a program

Examples:

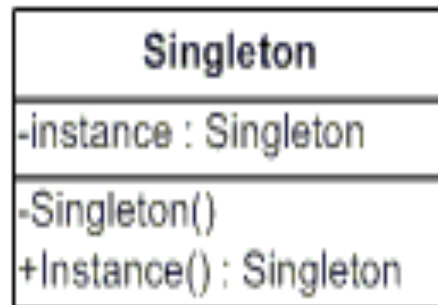
interface to a database

logging system

Singleton example code

```
public class MySingletonClass {  
    private static MySingletonClass instance  
        = new MySingletonClass();  
    public static MySingletonClass getInstance() {  
        return instance;  
    }  
    /** There can be only one. */  
    private MySingletonClass() {}  
}
```

UML: Singleton Pattern



“-” means private

“+” means public

Only one is ever created.

Abstract Factory

Depending on a value in a configuration file, we could set a variable to indicate the “situation”

But, the code would be littered with

```
if <this situation == A>  
    Class myClass = new SituationAClass()  
elseif <this situation == B>  
    Class myClass = new SituationBClass()  
...
```

Iterator Pattern

Used to iterate over a collection of objects

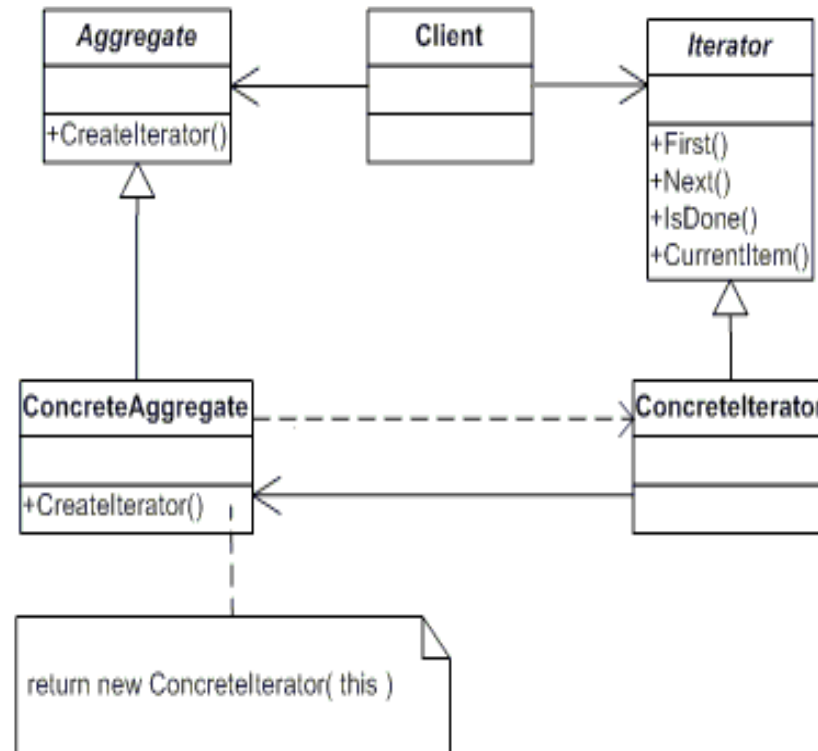
```
java.util.Iterator
```

Has:

- Collection of objects

- A way to iterate over them

UML: Iterator Pattern



Abstract Factory Pattern

Sometimes we have two or more parallel families of classes. We want all our objects to come from the same family but which family we want depends on some situation that we don't know at compile time.

Example:

a GUI application that needs GUI tool classes for the particular OS and windowing system in use

Abstract Factory

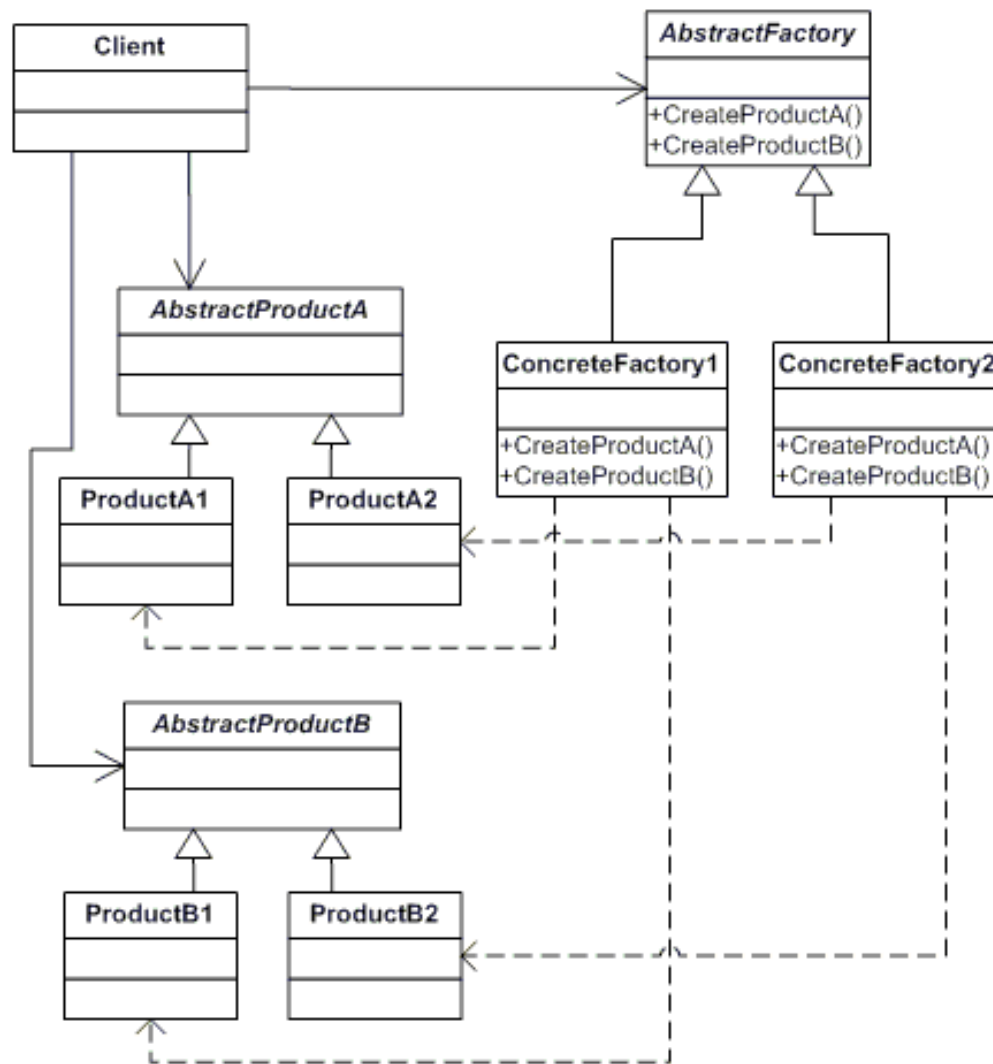
Solution: Don't instantiate classes directly.
Instead create a factory class for each situation.
SituationAFactory SituationBFactory ...

All the factory classes extend the same abstract class
Once in your code, instantiate the correct factory.

```
AbstractFactory factory = new SituationAFactory();
```

Everywhere else, call the factory methods to create appropriate objects.
factory.createClassX()

UML for Abstract Factory



Visitor Pattern

Intent:

Separate traversal of a complex structure from operations on that structure

Context:

Want an easy way to do walk around a complex structure

E.g. visit each node in a graph once

Visitor (cont'd)

Motivation:

Many different operations may need to be performed.

New operations may be added.

— implemented by adding new kinds of visitors.

Structure is complex enough that visiting elements is hard or error-prone.

What design pattern does that lead to?

The types of objects in the structure, and the ways they are connected, change infrequently.

Existing visitors will not need to be changed.

So it is worth investing time in building a support tool.

Visitor (cont'd)

Solution #1: create an “iterator” class with do-nothing methods that are called at specific times

This class takes care of things like remembering which nodes have already been seen

Users derive their own iterators from this, filling in methods to do what they want

Solution #2: create two classes:

a visitor class that has a method for each different kind of object in the structure

is activated by callback from structure object

an iterator class that hands the callback argument to each object in structure

iterator doesn't know what the visitor does, but knows how to carry the visitor to the visitees

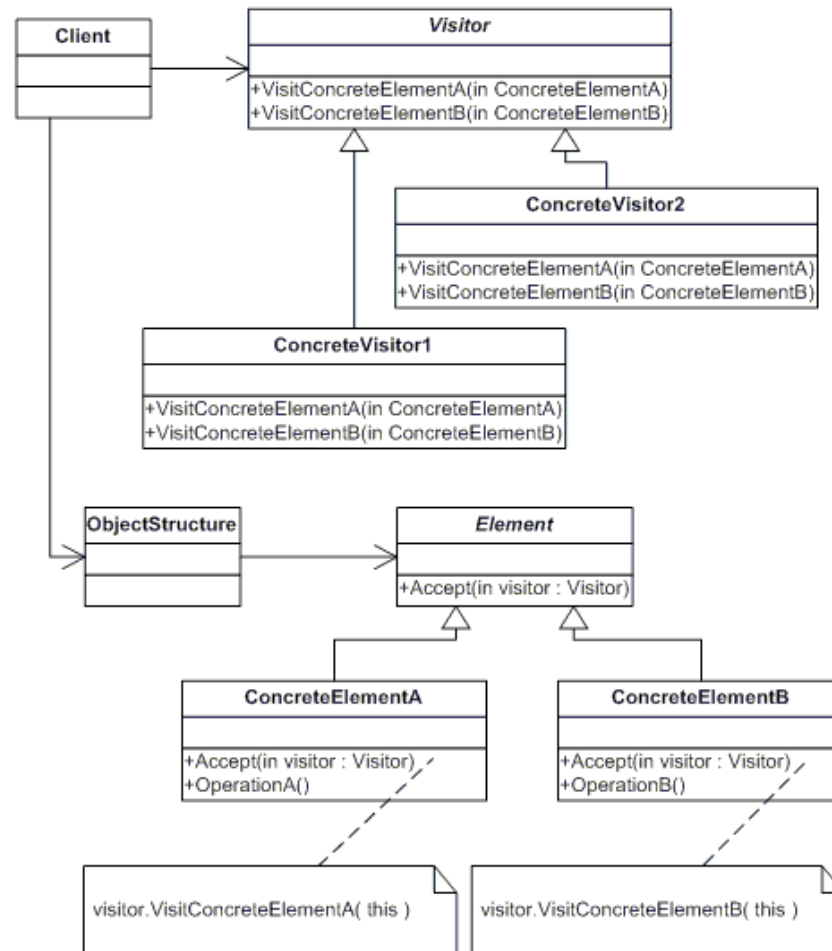
Visitor (cont'd)

Solution #1 is simpler, but:

Some languages (e.g. C) don't support derivation, so you have to use solution #2

Solution #2 allows you to re-use callbacks with different kinds of collections

UML:Visitor Pattern



Composite Pattern

Intent:

build complex hierarchies by making sub-hierarchies look like individual objects

Context:

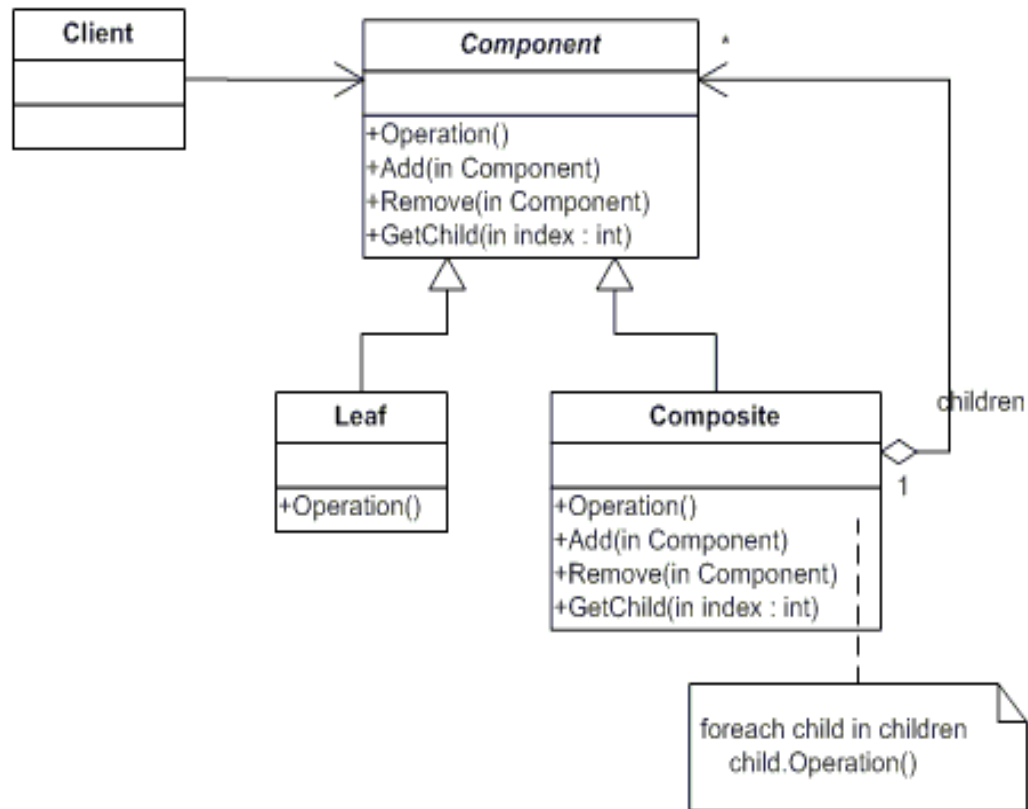
want to be able to treat nested collections uniformly

Motivation:

Want other classes to be able to treat collections and individual objects uniformly

Want to minimize the number of special cases that collections have to be aware of

UML: Composite Pattern



Composite (cont'd)

Solution:

Create an interface (or abstract class) defining the properties of both individuals and collections

Have individuals and collections implement this interface

Have collections treat everything they contain as instances of this interface

Composite (cont'd)

Example (old):

In JUnit, `TestCase` and `TestSuite` both implement `Test`

`TestSuite` contains zero or more `Tests`

So it can contain both test cases and test suites

Example:

Classes that implement `JComponent`

Components may contain other components. They implement the same interface so all components implement the same method.

JUnit – Test

```
public interface Test {  
    public int countTestCases();  
    public void run(TestResult result);  
}
```

The `Test` interface is implemented by `TestCase` and `TestSuite`

`TestSuite` is a composite of `TestCases`

We can call `Test` methods on both without knowing that one is individual and one is a composite

Composites and trees

Analogy: implementing a tree such that internal nodes and leaf nodes implement the same interface

`getName()`

`getValue()`

`getChildren()`

`add()` — A leaf node may need to throw an exception if add is not allowed on a leaf node.

`remove()`

Example interfaces

```
public interface Visitable {
    public void accept(Visitor visitor);
}
public interface Visitor {
    public void visit(Object o);
}
public class TreeNode implements Visitable {
    public void accept(Visitor visitor) {
        visitor.visitTreeNode(this);
        visitor.visitTreeNode(leftsubtree);
        visitor.visitTreeNode(rightsubtree);
    }
}
```

Using reflection

```
public void visit(Object object) throws Exception {
    // getMethod looks for a "visit" method
    // in the class, superclass and interfaces
    Method method =
        getMethod(getClass(), object.getClass());
    method.invoke(this, object);
    if (object instanceof Visitable){
        callAccept((Visitable) object);
    }
}
public void callAccept(Visitable visitable) {
    visitable.accept(this);
}
```

Adapter Pattern

Intent:

implement an interface known to one set of classes so that they can communicate with other objects that don't know about the interface

Context:

want to use a class in a way that its original author didn't anticipate

E.g. write data to a string instead of to a file

Or apply regular expressions to streams instead of to strings

Adapter (cont'd)

Motivation:

You want to use a class as though it implemented an interface that it doesn't actually implement

You do not want to modify or extend that class

You can translate the operations you want to perform to the ones the class actually implements

Solution: create an adapter that implements the interface you want, and calls the methods the class has

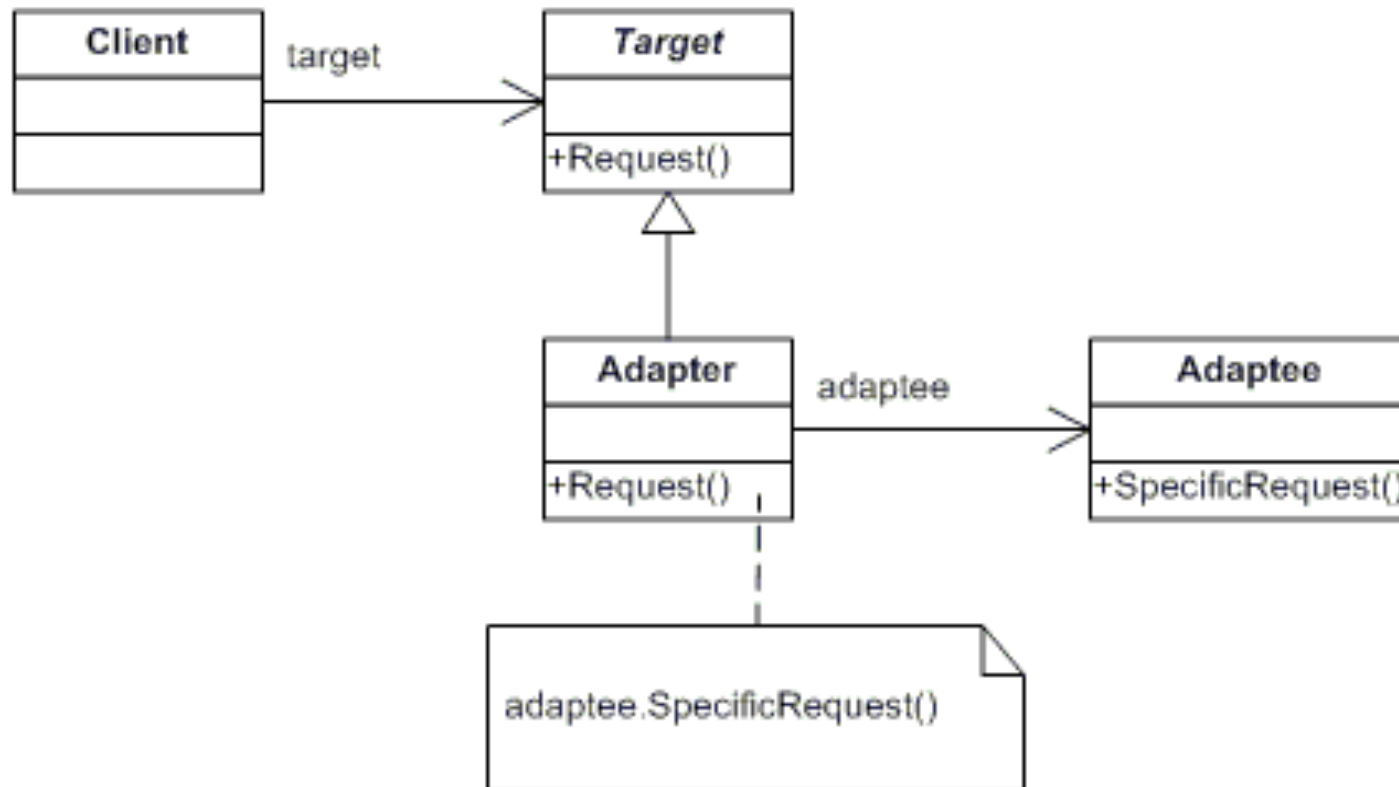
Adapter examples

The `StringIO` class:

allows a program to treat a string as though it were a file

A class that has methods that expects polar coordinates and you want it to work with a class that uses cartesian coordinates

UML: Adapter Pattern



Design Patterns: discussion

Not just about object-oriented design

User interface patterns

Business patterns

Anti-patterns (things to avoid)

Be careful, not every coding problem is a design pattern

Serve two purposes

Communication: a concise way for designers to communicate with each other

And argue out exactly what they mean

Often without worrying about specific implementation details

Education: gives them a way to communicate what they know to newcomers

Don't expect to connect them all to your own experience the first time

But keep them in mind as you work on other courses

"Hey, I know how to do this!"