More Design Patterns

Dr Ian Bayley

Oxford Brookes

Advanced O-O Prog

Why We Have Design Patterns

Definition

A *design pattern* is a general reusable solution to a commonly occuring problem in software design

- communicating expertise from experts to novices
  - experts with decades of industrial experience - not me!
- common vocabulary for communication between experts
- usually the solutions are examples of best practice
  - eg they minimise the amount of rewriting needed when software changes

What You Need to be Able to do With Design Patterns

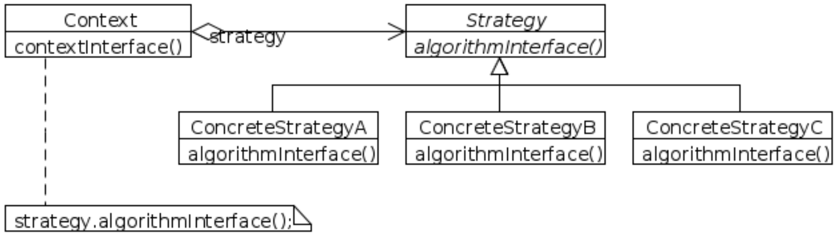
- explain what each design pattern is for
- explain how each design pattern achieves its intent
- explain the difference between two design patterns
- explain why a program conforms to a design pattern (or why not)
- recognise when a pattern might be needed in a given situation
- apply a pattern in a given situation
  - instantiating UML diagram or writing Java code or both
  - incorporating existing classes and operations
- explain in detail the consequences of alternatives to the pattern
  - duplicated code, switch statements
  - consequent cost of subsequent modification
- demonstrate how code that implements a design pattern can be extended

What You Need to do by Week 8 (for 5% of Module Mark)

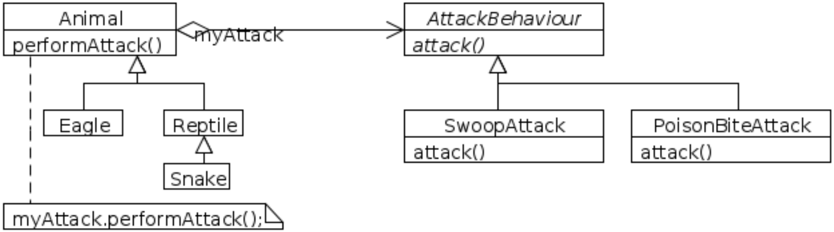
1. for each of 4 (any 4 of our 13) design patterns
    1. find an example with a class diagram
        - in a book or on the internet but not one of our own examples
        - you can make up your own example if you like
    2. prove that the design conforms to the pattern (1% each) as seen shortly
2. translate one class into corresponding Java (1%)
3. give a talk about any one of your design patterns in the lecture

- this is to make sure
    - you understand the material in time for the exam
    - you have good notes that you can understand and learn from
- to save you time (it's exam prep), code does not need to be compiled nor tested, and minor syntax mistakes will be ignored

Class diagram for the Strategy Pattern



Class Diagram for David's AttackDemo program



Checking that AttackDemo Conforms to Strategy

- there must exist [1]
  - a Context class ✓ (Context = `Animal`)
  - a Strategy interface ✓ (Strategy = `AttackBehaviour`)
  - an abstract operation contextInterface in Context ✓ (contextInterface = `performAttack`)
  - an abstract operation algorithmInterface in Strategy ✓ (algorithmInterface = `attack`)
  - a set of classes $\text{ConcreteStrategy}_1 \ldots \text{ConcreteStrategy}_n$[2] ✓ ($\text{ConcreteStrategy}_1$= `SwoopStrategy` etc)
- ... such that (remember the bindings)
  - Context associates with `Strategy` ✓
  - $\text{ConcreteStrategy}_i$ all concrete ✓, inherit from `Strategy` ✓
  - Context.contextInterface calls Strategy.algorithmInterface ✓

[1] and they must all be different

[2] for $n \geq 1$; this generalisation from n=3 is implicit

Strategy Conditions Restated in Predicate Logic for Clarity

$$\begin{aligned}
&\exists Context, Strategy, contextInterface, algorithmInterface, ConcreteStrategies \cdot \\
&\quad Context \in classes \wedge \\
&\quad Strategy \in classes \wedge \\
&\quad ConcreteStrategies \subseteq classes \wedge \\
&\quad contextInterface \in Context.opers \wedge \\
&\quad algorithmInterface \in Strategy.opers \wedge \\
&\quad isInterface(Strategy) \wedge \\
&\quad assoc(Context, Strategy) \wedge \\
&\quad \forall CS \in ConcreteStrategies \cdot \neg isAbstract(CS) \wedge inherits(CS, Strategy) \wedge \\
&\quad calls(Context.contextInterface, Strategy.algorithInterface)
\end{aligned}$$

- to see if the design satisfies the pattern
  - replace the existentially-quantified variables with actual classes/operations from the diagram eg Context=Animal etc
  - evaluate the condition with the classes/operations
  - if the condition is true then the design satisifies the pattern; if not, no

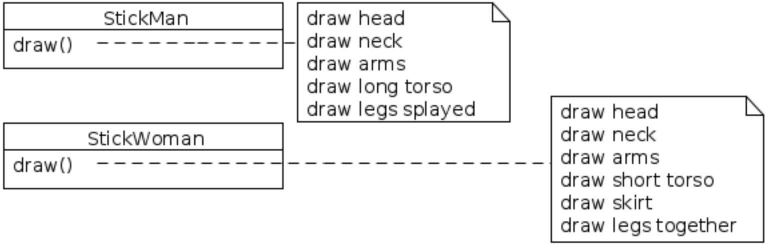
Which Design Patterns are in This Lecture

Template Method	looks like the Strategy pattern with Context = Strategy but used for sharing code between several slightly different algorithms
Iterator	like Factory Method but for making an iterator, a pointer into a data structure, as used behind-the-scenes by the enhanced for loop
Observer	the heart of Model/View/Controller, as used in all well-designed GUIs
Composite	the right[3] way to form tree structures

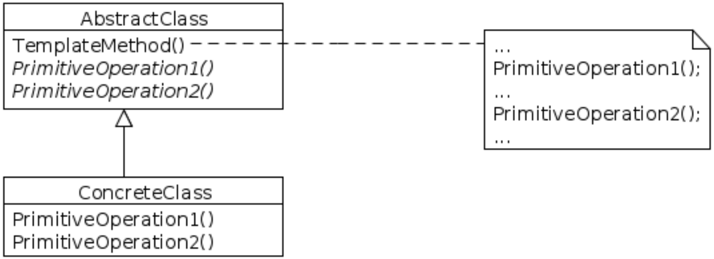
[3] according to Mr O-O

Outline

An instance of the Duplicated Code Smell



Class Diagram for the Template Method Pattern



Instantiating Class Diagrams for Patterns

1. apply bindings to class variables and sets of class variables
    - a single subclass usually represents a set of subclasses
2. apply bindings to operation variables and sets of operations variables
    - a single operation often represents a set of operations
3. add extra classes and operations, as appropriate

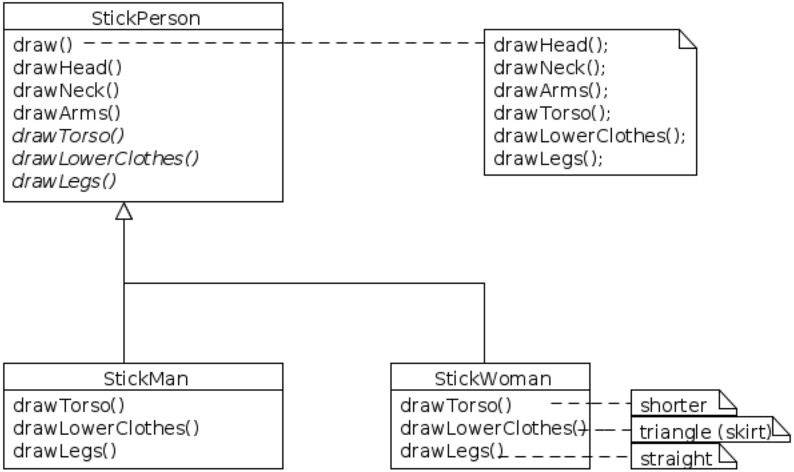
1. Drawing Stick People

AbstractClass=`Person`
ConcreteClasses={`Man`, `Woman`}
TempleMethod=`draw`
PrimitiveOperations={`drawTorso`, `drawLowerClothes`, `drawLegs`}

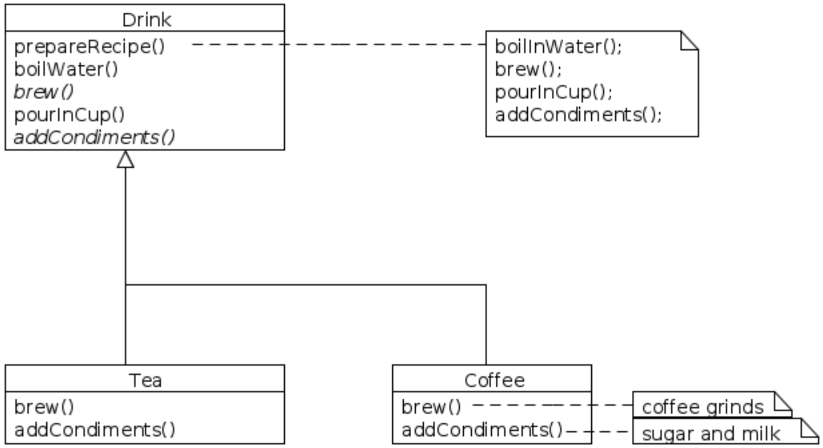
2. Preparing a Drink

AbstractClass=`Drink`
ConcreteClasses={`Tea`, `Coffee`}
TemplateMethod=`prepareRecipe`
PrimitiveOperations= {`brew`, `addCondiments`}

Class Diagram for Drawing Stick People



Class Diagram for Preparing a Drink



Variations on the Template Method Pattern

- variable operations may either be
  - hooks (=empty-bodied operations that may be overridden) or
  - abstract operations (=no bodies, must be overridden, as in diagram)
- you should make `templateMethod` final and `primitiveOperations` protected
- try to minimise the number of primitive operations for clarity

The Hollywood Principle exemplified by Template Method

The Hollywood Principle

- "don't call us, we'll call you"
- ie children don't call parents, parents call children

- often used by software frameworks, which supply `AbstractClass`, eg `Activity` in U08928, while the user writes `ConcreteClass`

Software framework

a set of classes that unlike a library forms a complete program that already performs some useful function. The user tailors it by supplying extra code. Unlike a library, a framework dictates the flow of control.

- the more normal "non-Hollywood" practice is for children to call parents using `super` but this can't be enforced by the framework
- Java compiler enforces this when you are writing a constructor

Outline

The Enhanced For Loop, Which Uses the Iterator Pattern

Example use of enhanced for loop

```
Collection<String> strings; // then some code to fill it
for (String s:  strings)
  System.out.println(s);
```

Behind-the-scenes implementation using the Iterator pattern

```
Collection<String> strings; // then some code to fill it
Iterator<String> iterator = strings.iterator();
while (iterator.hasNext()){
  String s = iterator.next();
  System.out.println(s);
}
```

The Iterator Pattern in the Collections API

The interface Iterator<E>

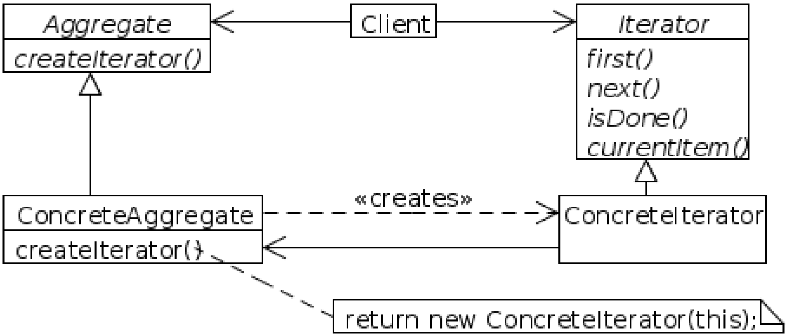
`boolean hasNext();` Returns true if the iteration has more elements.
`E next();` Returns the next element in the iteration.
`void remove();` Removes from the underlying collection the last element returned by this iterator (optional operation).

The interface Collection<E> (only a tiny part)

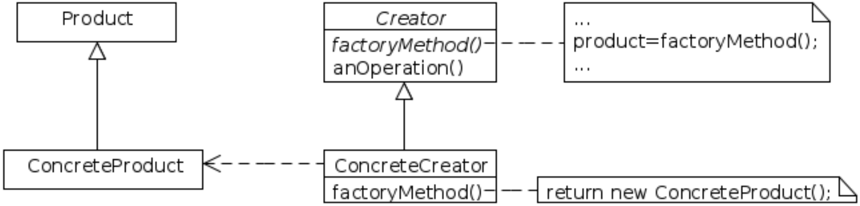
`Iterator<E> iterator();` Returns an iterator over the elements in this collection.

- this almost conforms to Iterator pattern overleaf with
  - $Aggregate = Collection < E >$ and
  - $ConcreteAggregate = ArrayList < E >$, for example
- but you'd need to change the operations to make it match
- likewise, Iterator almost conforms to Factory Method

Class Diagram for the Iterator Pattern



Class Diagram for the (Similar) Factory Method Pattern



Variations on the Iterator pattern

- you could let the Client advance the iterator himself
    - instead of advancing it automatically with each call to `next`, as above
    - more control for the Client but more work calling the advance operation
- could add more operations eg `previous`, `skipTo`
- remove is in general very difficult to implement so most people use:

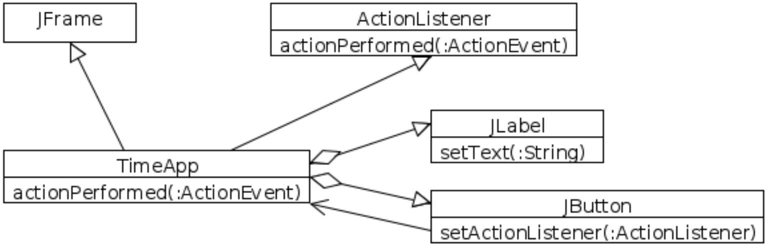
  ```
  throw new UnsupportedOperationException();
  ```

Implementation and Consequences of the Iterator Pattern

- `ConcreteIterator` has field to keep track of current position
    - eg `int index;` for position in an array or `ArrayList`
- two tricky problems (see references for solutions - unassessed):
    1. thread safety issue - what if somebody changes the collection while you are traversing it?
    2. making an iterator for Composites is hard
        - head node, current child node, iterator for that child node
- the Iterator pattern is extremely versatile:
    - you can traverse many different `ConcreteAggregates` at once
    - and for each one, you can run several difference traversal strategies at once, each one a `ConcreteIterator` object
        - eg backward/forward, pre-/in-/post-order

Outline

GUI Programming Without the Observer Pattern

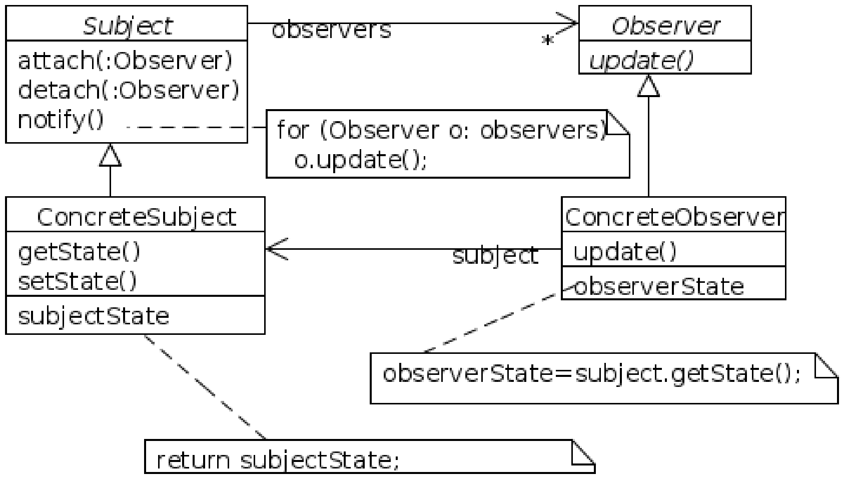

JFrame
ActionListener
actionPerformed(:ActionEvent)
JLabel
setText(:String)
TimeApp
actionPerformed(:ActionEvent)
JButton
setActionListener(:ActionListener)

The Limitations of That (Most Obvious) Solution

- when the button is pressed, class TimeApp has just one class to update to change the display so the `actionPerformed` method is simple
- but what if there were several display classes, each with different methods to call
- then `actionPerformed` would be complex and it would have to change whenever a new class was added or an old class would change

- it would be simple though if actionPerformed only needed to a list of display classes, all of which shared the same interface, having only one method to ask the display class to change
- these are Observer classes, all sharing the same interface Observer
- there is also a Subject class, containing data to be displayed
- we say that the coupling between the Observer and Subject is loose because Subject has to know very little about Observer (which can change without Subject having to change)

Class Diagram of Observer pattern



Java API Support for the Observer Pattern

The interface Observer

`void update(Observable o, Object arg);` This method is called whenever the observed object is changed

- here the new state is sent (push model) to the Observer so it does not have to query the Observable (the pull model shown in diagram)

The class Observer

```
void addObserver(Observer o){}
void deleteObserver(Observer o){}
void notifyObservers(){}
protected void setChanged(){}
```

- major drawback of Observer pattern is unnecessary updates; this can be mitigated by having different sorts of update

Code for an Observable

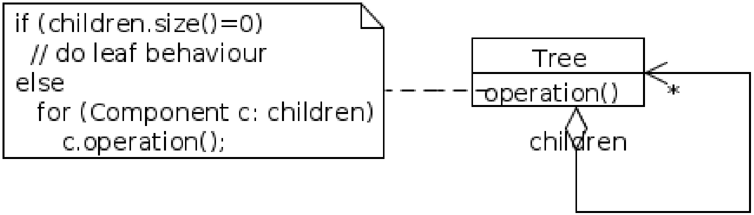
```
public class WeatherData extends Observable {
  private float temperature, humidity, pressure;
  public WeatherData() {}
  public void measurementsChanged() {
    setChanged();
    notifyObservers();
  }
  public void setMeasurements(float t, float h, float p){
    this.temperature = t;
    this.humidity = h;
    this.pressure = p;
    measurementsChanged();
  }
}
```

Code for an Observer

```
public class CondsDisplay implements Observer, Display {
  private Observable observable;
  private float temperature, humidity;
  public CondsDisplay(Observable observable){
    this.observable = observable;
    observable.addObserver(this);
  }
  public void update(Observable obs, Object arg){
    if (obs instanceof WeatherData) {
      WeatherData weatherData = (WeatherData) obs;
      this.temperature = weatherData.getTemperature();
      this.humidity = weatherData.getHumidity();
      display();
    }
  }
}
```

Outline

Class Diagram for Trees (Procedural Approach)

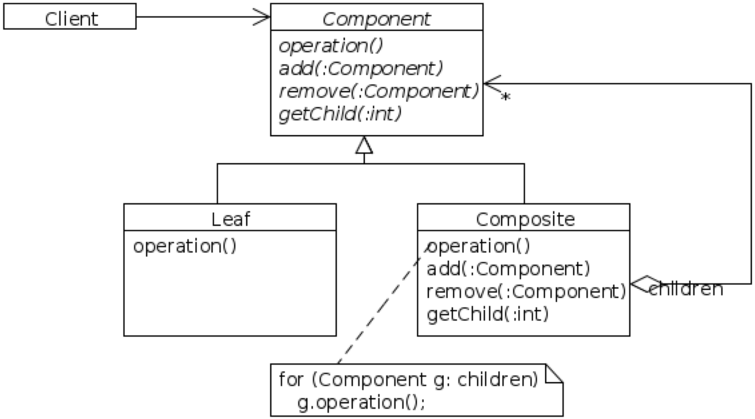


```
if (children.size()=0)
  // do leaf behaviour
else
   for (Component c: children)
      c.operation();
```

Mr O-O's Criticisms of That (Most Obvious) Approach

- what if there were several operations? Then you'd have to change every one every time you wanted to add a new type of leaf or internal node! And then retest them! Argh!
- don't you remember polymorphism from a few weeks ago?
  - each type of node should have a different class
  - each inheritting from a more general class
  - call the version defined on the more general class, which makes the Client class simpler
- this gives the Composite design pattern, as used by Swing, for example

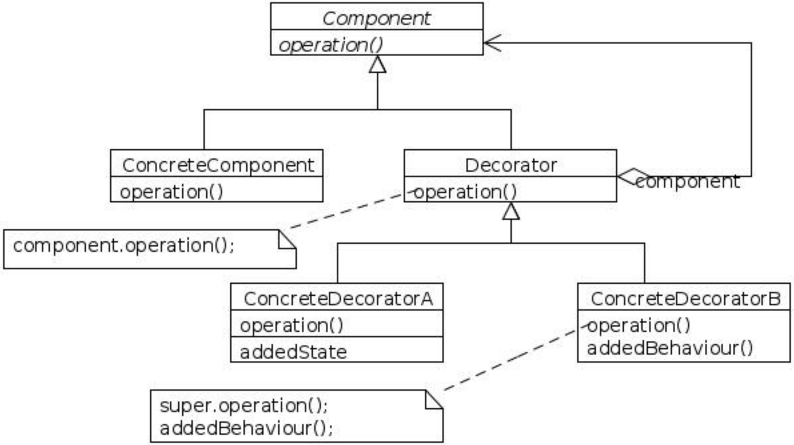
Class Diagram for the Composite Pattern



Variations on the Composite Pattern

- you can add many different types of leaf and many different types of composite
- you may want certain composites to be used only with certain components but if so you have to add the checks yourself; the type system can't be used
- you may want child components to reference their parents
  - invariant: every child of a composite has the composite as its parent
- operations `add`, `remove`, `getChild` cannot meaningfully be defined for the `Leaf` class so either include them in the Composite interface and give them "meaningless" definitions or narrow the interface and don't include them
- arguably, Decorator is a degenerate (linked list vs tree) version of Composite but it has a different purpose, adding responsibility

Class Diagram for the Decorator Pattern (for comparison)



Outline

Intents of the four design patterns I

Intent of Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Intent of Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Intents of the four design patterns II

Intent of Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Intent of Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

What We Have Done With Design Patterns

- instantiated patterns to particular situations
  - Template Method to Stick People
  - Template Method to Drink Preparation
- confirmed that a design is an instance of a pattern
  - David's AttackDemo is an instance of Strategy
- considered variations on patterns
  - Template Method, Iterator, Observer, Composite
- compared two similar patterns
  - Template Method vs Strategy
  - Iterator vs Factory Method
  - Composite vs Decorator
- considered the consequences of not using particular patterns
  - Template Method, Iterator, Observer, Composite