Remaining Design Patterns

Dr Ian Bayley

Oxford Brookes

Advanced O-O Prog

Intents of the Design Patterns we have seen so far I

Intent of Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Intent of Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Intents of the Design Patterns we have seen so far II

Intent of Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Intent of Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Intents of the Design Patterns we have seen so far III

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 Design Patterns we have seen so far IV

Intent of Iterator

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

Intent of Observer

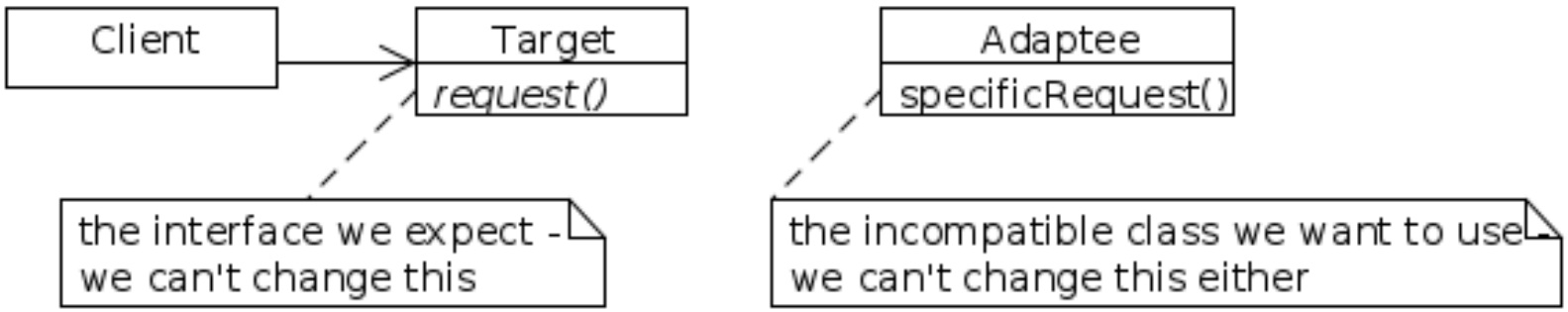
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The remaining Design Patterns

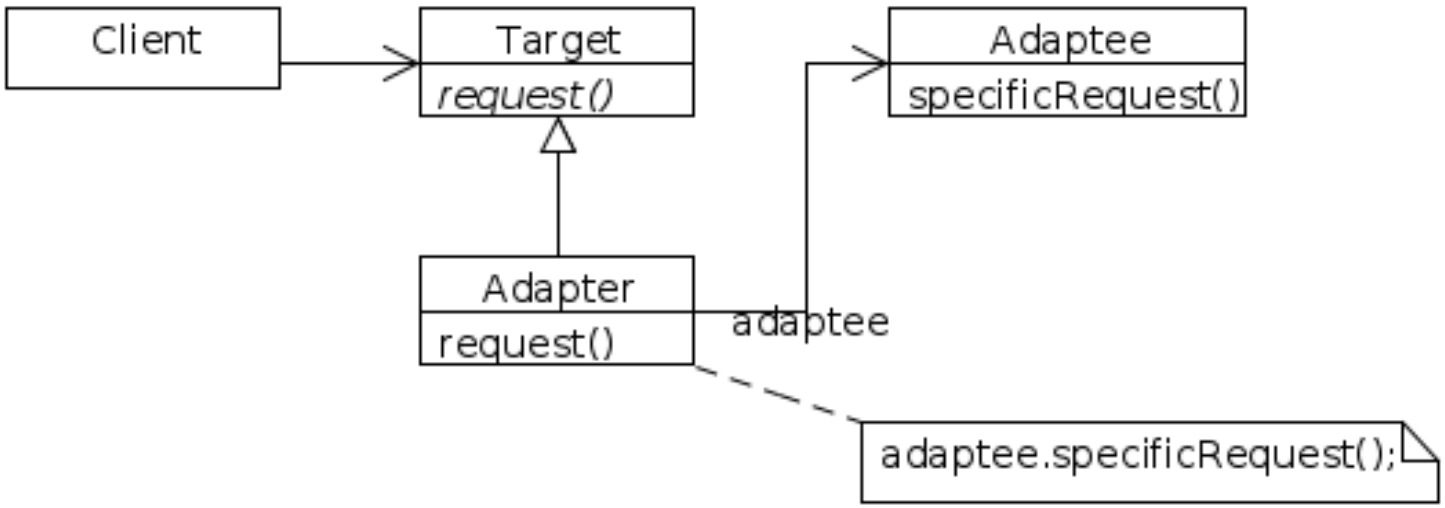
Adapter	Change the interface of an object, like a plug adapter
Facade	Provide a single interface to a set of interfaces
Singleton	Ensure that only one object can be created from a class
Command	Represent method calls as objects
State	Make the behaviour of an object dependent on its state

Outline

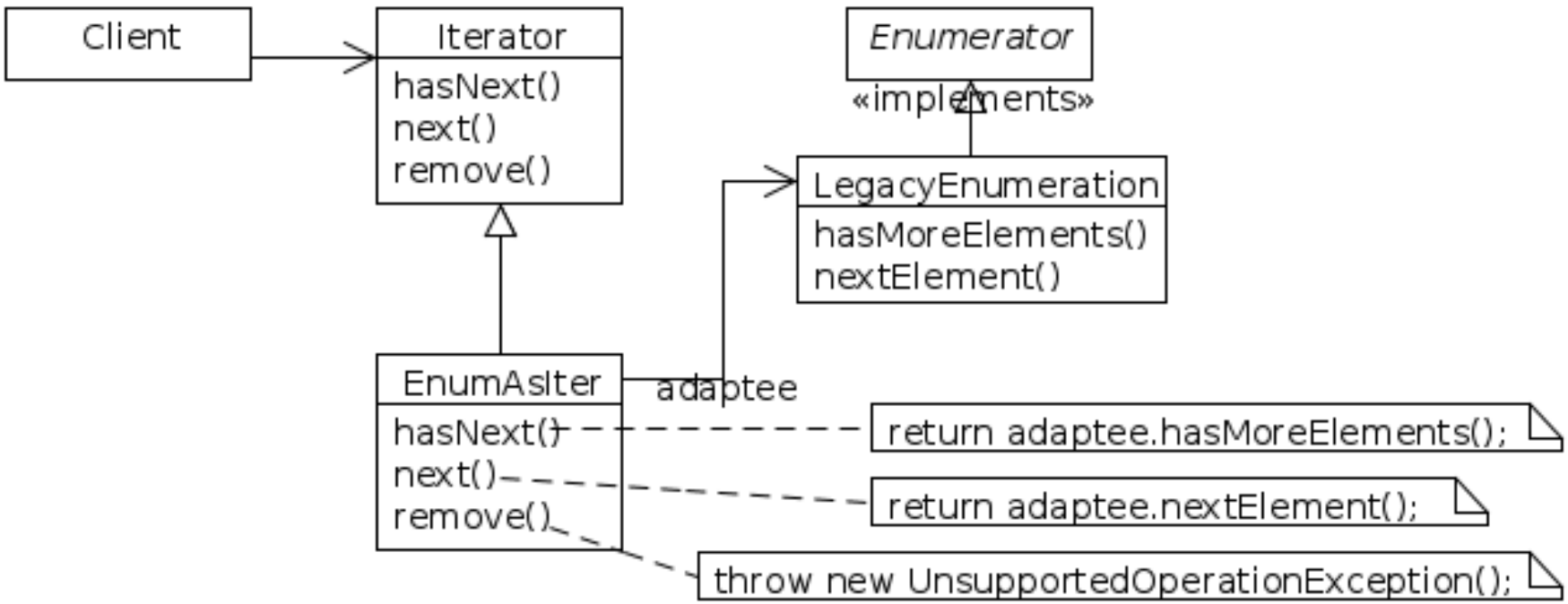
Class Diagram for when the Adapter pattern is needed



Class Diagram for the Adapter pattern



Class Diagram for Enum/Iter example of Adapter pattern

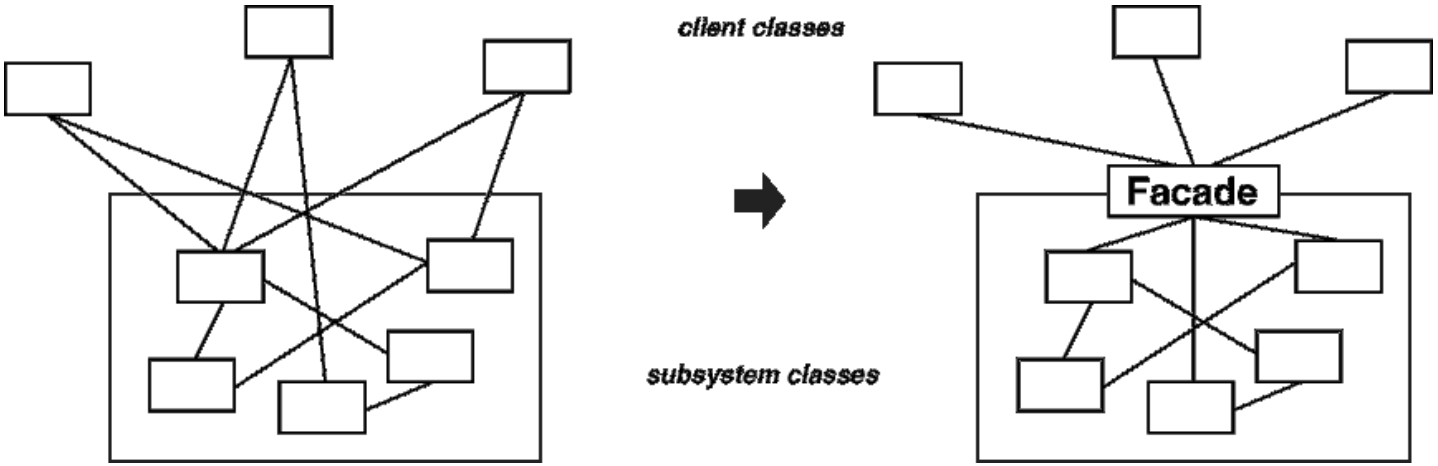


Two Kinds of Adapter Pattern (pros and cons)

- Object Adapter (as above)
  - Adapter is an object that associates with the Adaptee
  - ... which is an extra object
  - operation `specificRequest` cannot be overridden
  - can have many different Adaptee classes satisfying the same interface
  - can have many different Adaptee classes satisfying different interfaces
- Class Adapter
  - Adapter inherits from the Adaptee
  - so it is not an extra object
  - operation `specificRequest` can be overridden
  - commits to a particular Adaptee class so we can't adapt a class *and* all of its subclasses

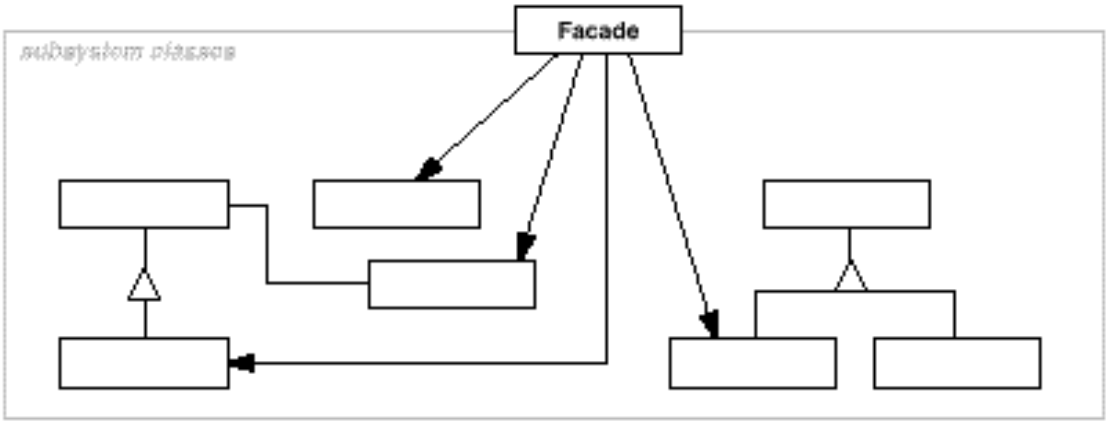
Outline

Applying the Facade Design Pattern

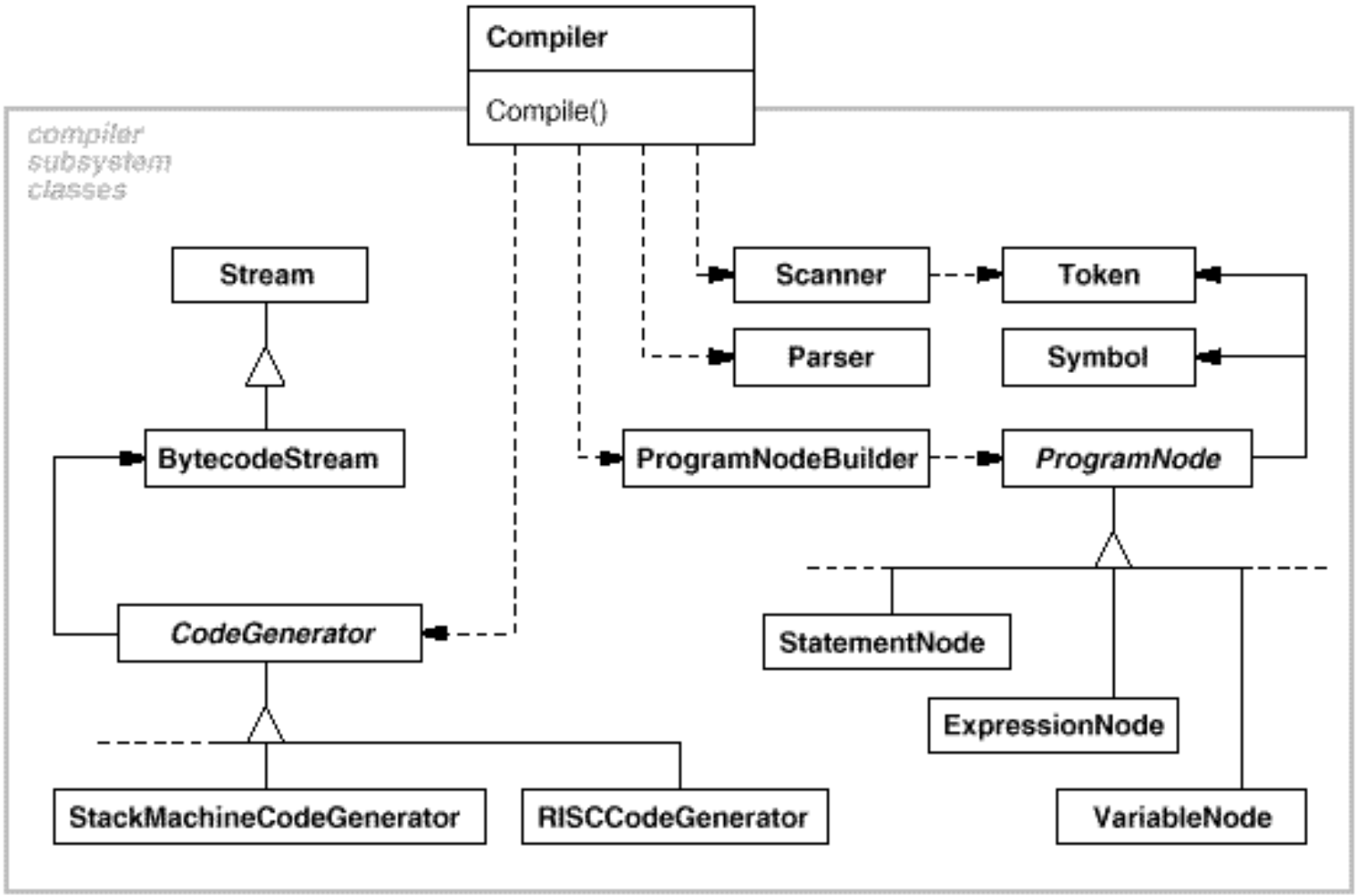


Advantages of the Facade Pattern's Single Interface

- reduced compexity because the client classes need to know about just one "middleman" object
- lower coupling between client and subsystem classes so changes to the latter are less costly
  - if there are n clients and m service classes, with $n, m \geq 1$, the number of communications (that might need to change in future) is reduced from $n * m$ to $n + m$
- note that the subsystem classes are not encapsulated so they can still be accessed



Class Diagram of an Example use of Facade



Outline

How Singleton Pattern works

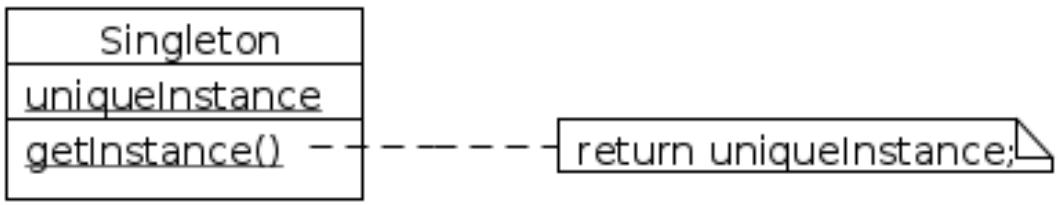
- used whenever you want to ensure that only one instance of an object is created
  - eg print spooler, system registry, window manager etc

1. make the constructor private to stop the use of `new`
2. hold the instance in a static field `uniqueInstance`
   - so the class is responsible for keeping track of its only instance
3. provide a static method `getInstance` to access it

- remember that a non-static field has a copy for each object but a static field has exactly one copy, associated with the class, no matter how many objects have been created

Code and Class Diagram for the Singleton Pattern

```
class Singleton {
  private static Singleton uniqueInstance;
  private Singleton (){}
  public static synchronised Singleton getInstance(){
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
}
```



More on Singleton Pattern

- `synchronized` means that two invocations of the method (on the same object) cannot interleave
- initialisation could be eager, for simpler code, at the cost of creating an object always even if it is not needed

The eager alternative to lazy initialisation

```
private static Singleton uniqueInstance = new Singleton();
```

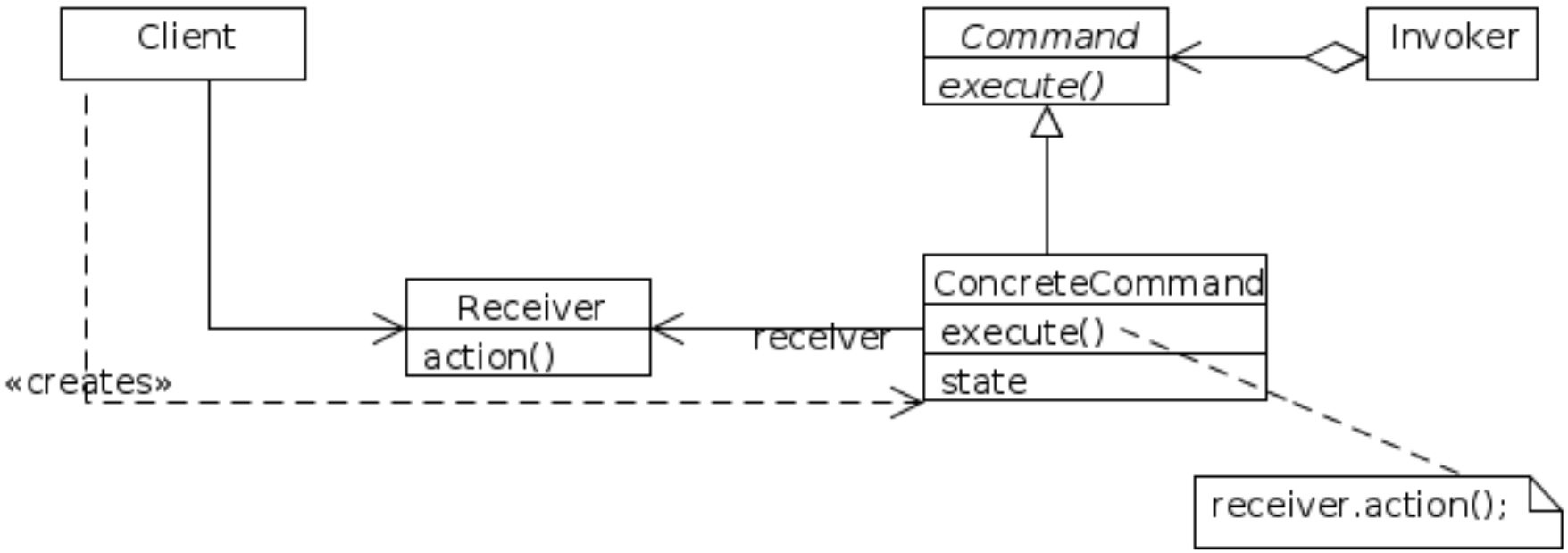
- the instance is globally accessible without being a global variable
- you could adapt this to have exactly two instances or three instances etc

Outline

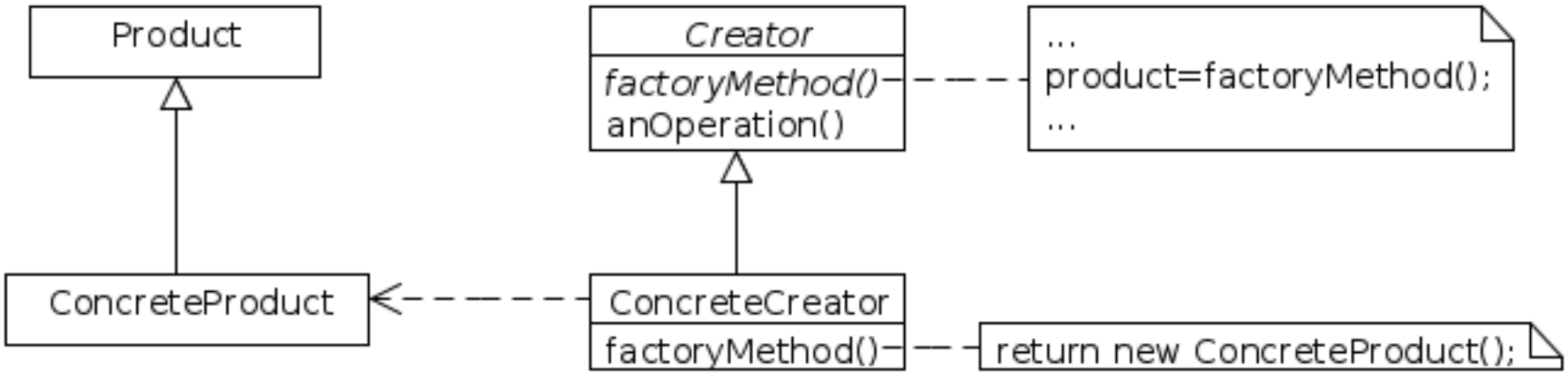
Motivation for Command Pattern

- issue request to objects without knowing the operation or the receiver
  - as used in user interface toolkits (by MenuItem, Button etc)
- so we represent each request with a command object
  - and then the invoker only needs to know how to run the command
- using polymorphism makes it easier to add new commands (as usual)
- note similarity with Factory Method pattern

Class Diagram for Command Pattern



Class Diagram of the Factory Method pattern (comparison)



More on Command Pattern

Example instances of the Command Pattern

Invoker= Button, Command = PasteCommand, Receiver = Document
Invoker= MenuItem, Command = OpenCommand, Receiver = Application

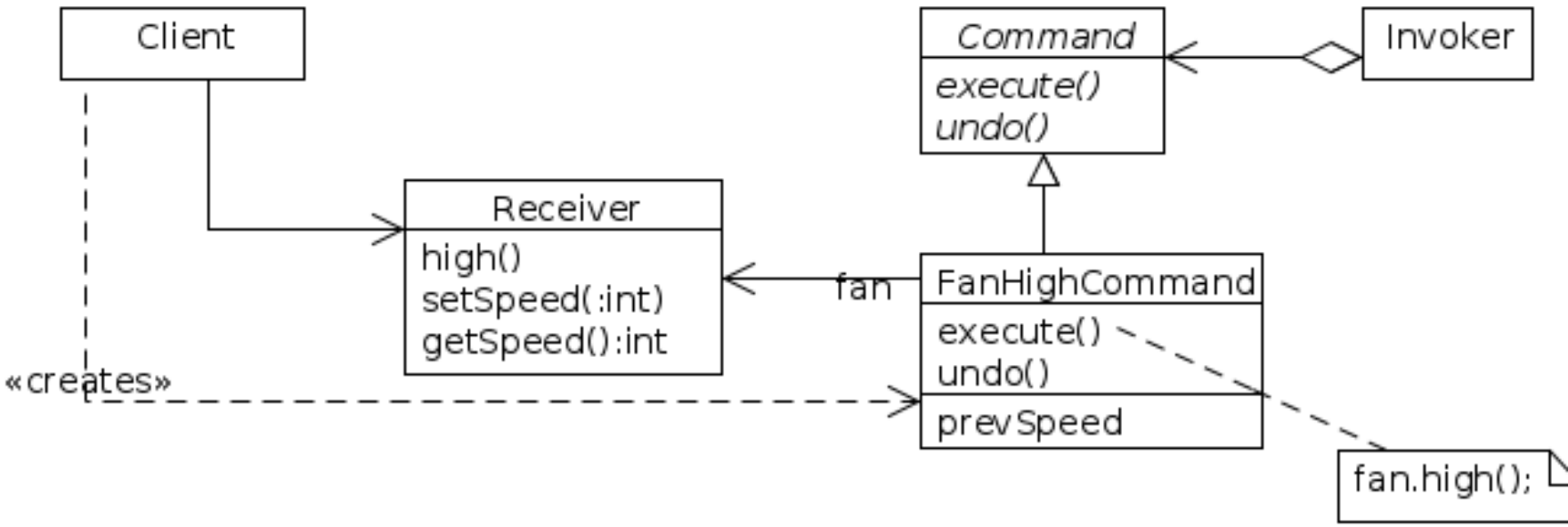
- commands can be undone or stored in a log (eg ArrayList) for redoing
- composing with the Composite pattern allows commands to be macros:

Composing Composite and Command patterns (unassessed)

$$(Command * Composite)[Component = Command \\ \wedge Composite = ConcreteCommand \wedge operation = execute]$$

- where $*$ denotes superposition and [] denotes restriction

Class Diagram for Fan example of Command Pattern



Code for Fan example of Command Pattern

```
public class FanHighCommand implements Command {
  private Fan fan;
  private int prevSpeed;
  public FanHighCommand(Fan fan){
    this.fan = fan;
  }
  public void execute() {
    prevSpeed = fan.getSpeed();
    fan.high();
  }
  public void undo(){
    fan.setSpeed(prevSpeed);
  }
}
```

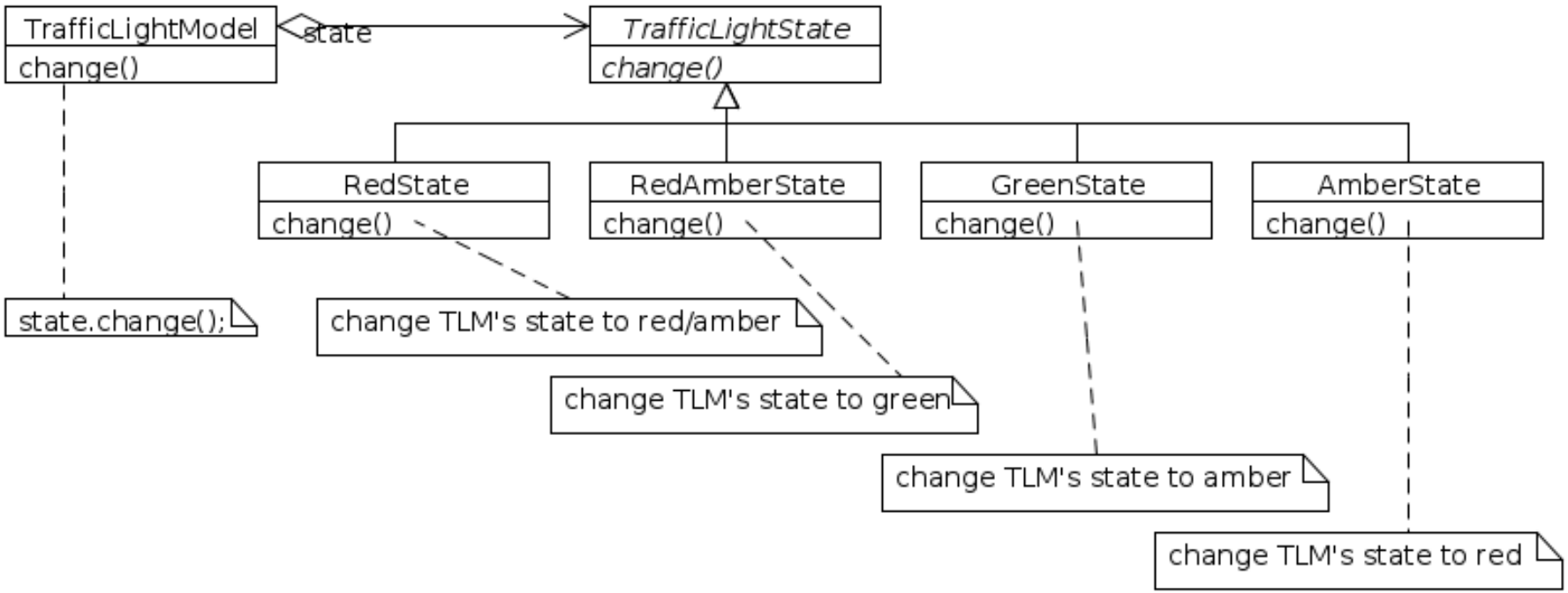
Outline

Motivation for State Pattern

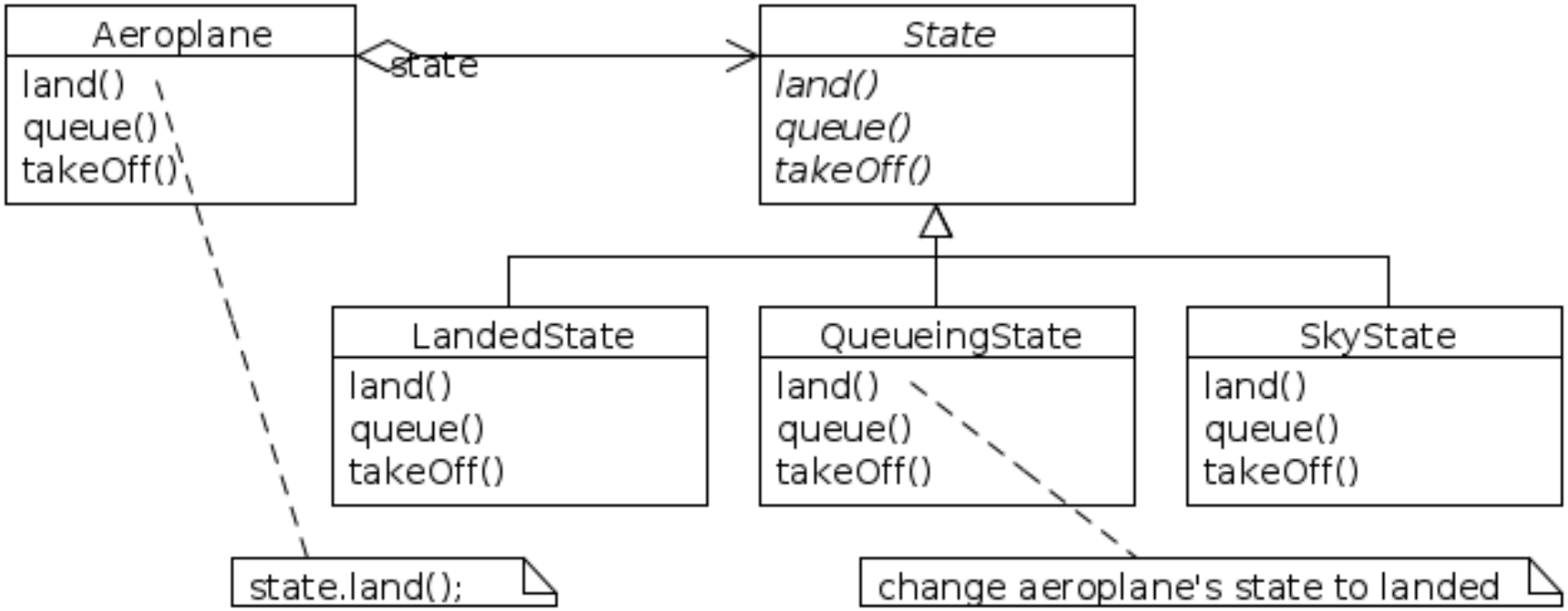
- we want the behaviour of the object to depend on its state
- the state of an object is the values of all of its attributes
- suppose the number of states is small
  - eg traffic lights has {red, redamber, green, amber}
- procedural approach will lead to excessive switch statements
  - the usual argument for polymorphism
- so instead we represent each state by a subclass

- how can we change the state from one subclass to another?

Class Diagram for Traffic Light Example Of State Pattern



Class Diagram for Aeroplane Example Of State Pattern



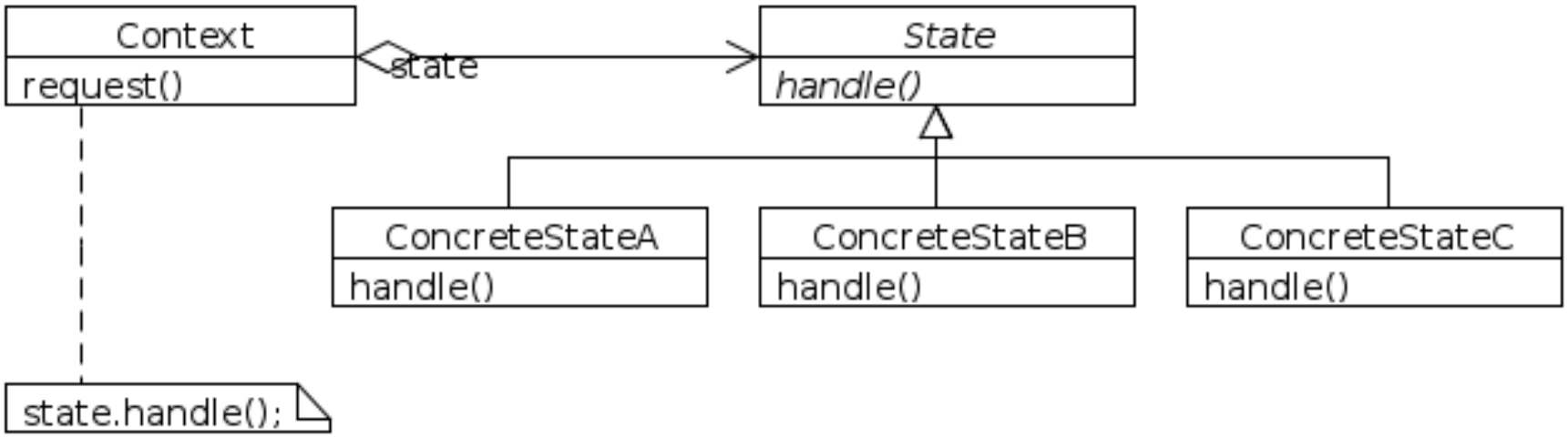
Code for Aeroplane Example Of State Pattern I

```
public class Aeroplane {
  private State landedState;
  private State queueingState;//and others
  private State state = skyState;
  public Aeroplane(){
    landedState = new LandedState(this);
    queueingState = new QueueingState(this);//and others
  }
  public void land(){
    state.land();
  }
  public void setState(State state){
    this.state = state;
  }
```

Code for Aeroplane Example Of State Pattern II

```
public class QueueingState implements State {
  private Aeroplane aeroplane;
  public QueueingState (Aeroplane aeroplane){
    this.aeroplane = aeroplane;
  }
  public void queue(){
    System.out.println("It is already in the queue!");
  }
  public void land(){// and others
    aeroplane.setState(aeroplane.getLandedState());
  }
}
```

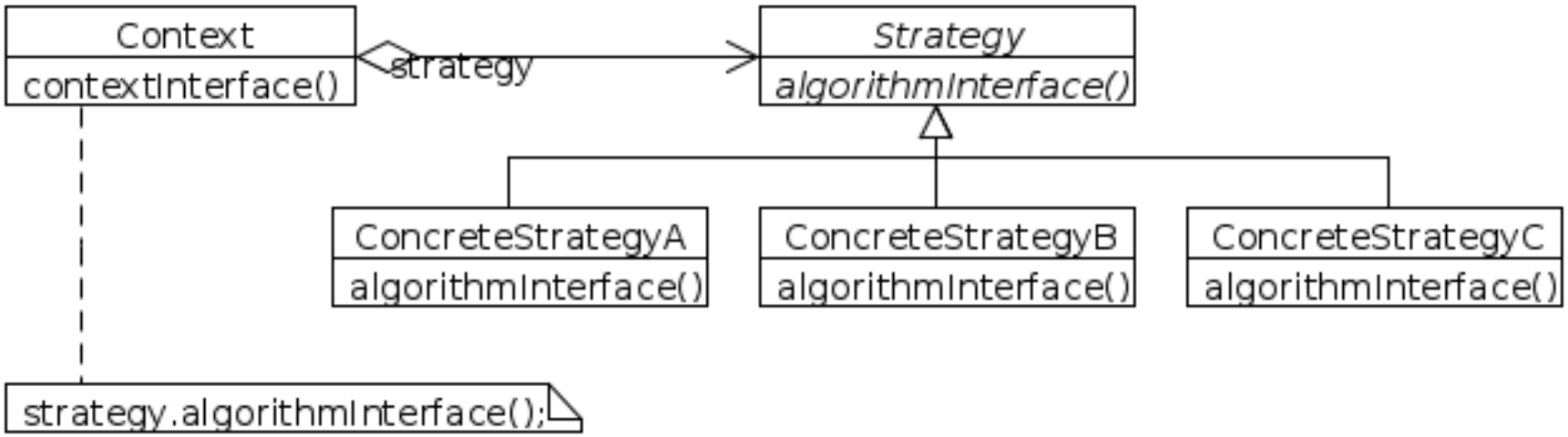
Class Diagram for the State Pattern



More on State Pattern

- further advantages are that state-specific behaviour is localised and state transitions are made explicit
- it would be slightly simpler to make each transition operation return an object representing the new state instead of setting it
- simpler too to create each state object when needed (but wasteful of memory)
- class diagram is the same of that of Strategy but the intent and code is different

Class Diagram for the Strategy Pattern (for comparison)



Conclusion I

Intent of Adapter

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

Intent of Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Intent of Singleton

Ensure a class only has one instance, and provide a global point of access to it.

Conclusion II

Intent of Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Intent of State

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.