7SENG003W Advanced Software Design

Bridge and Strategy patterns

Design Patterns

- Design patterns are related to heuristics, but are expressed in terms of template solutions to generally-expressed problems
- A design pattern …

"describes a problem that occurs over and over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without doing it the same way twice."

– Christopher Alexander

Recurring Problems

- The kind of problems that keep recurring are various, ranging from the general, such as:
  - how to make sure that software is sufficiently flexible that it can easily adapt to future changes in requirements
  - how to make your classes loosely-coupled so that they're not dependent on each other
- to the specific, for example:
  - how to make sure that you only create a certain number of instances of an object
  - how to be able use different versions of an algorithm without affecting the rest of the system
  - how to change the implementation of a class without affecting its interface
  - how to be able treat different parts of a whole-part hierarchy as if they were the same

Problems: Different yet the same

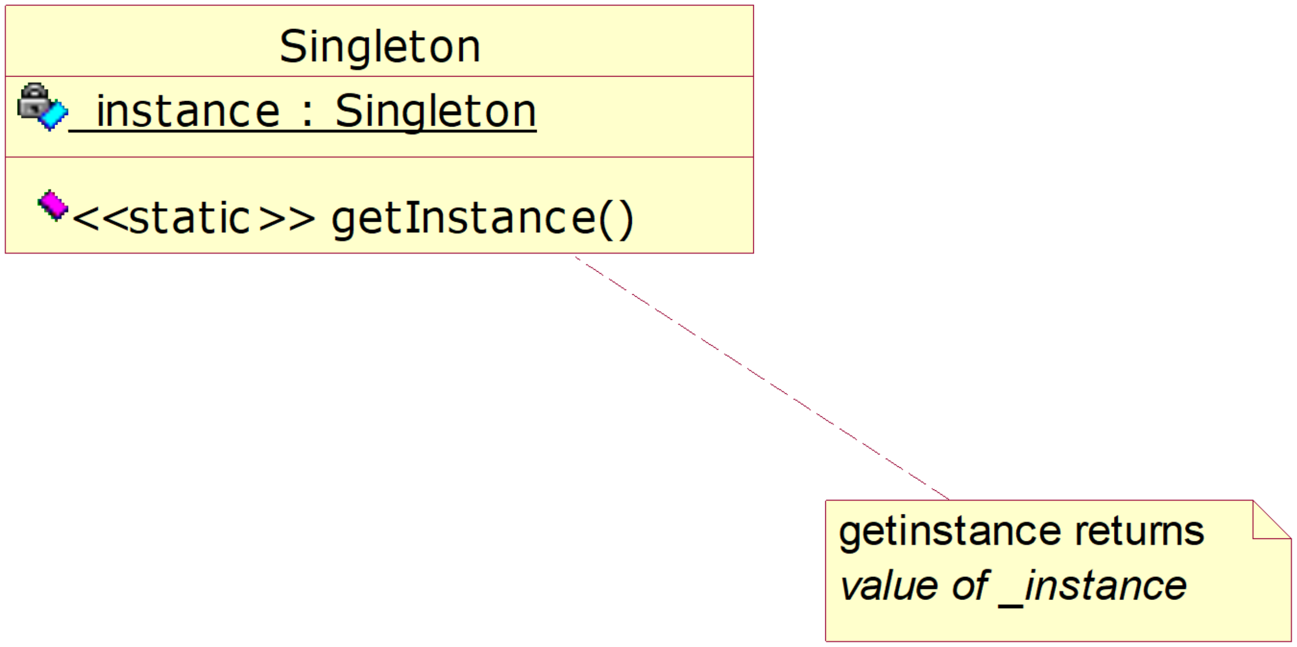
- Recurring problems arise in different systems and often in different guises, yet the basic structure of the problem is the same
- For example,
  - inside a Java-enabled browser, there should be one instance of a Java Security Manager to police the restrictions on applets – once created, the Security Manager object should not be changed
  - inside a financial transactions system, there should be one logging object which records details of a day's transactions - once an instance of the logger has been created, another instance should not be created in case it overwrites the logfile
- Both problems arise in different circumstances, yet the fundamental structure of the problem is the same.

How design patterns help

- Design patterns
  - express the recurring problem in a general way, so that instances of the problem can be easily recognized
  - they might give examples of the problem
  - they express a solution in a generic way, usually as a design fragment (using a class diagram), which can be adapted and incorporated into your particular design
  - they discuss the limitations and tradeoffs - the consequences of using the pattern
- Design patterns are not really to be used in isolation from each other, although that's how most people start using patterns. They're more a way of life for designers and developers.

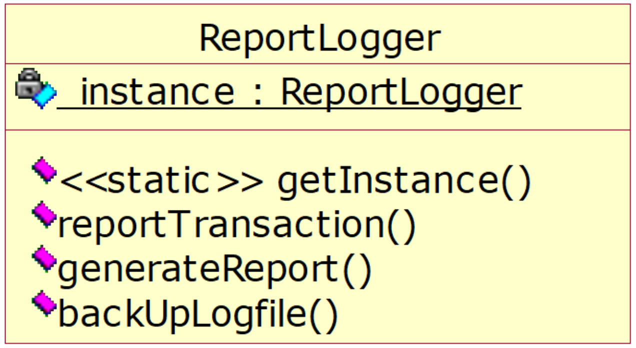
An Example Pattern

- Name: Singleton
- Problem: It may be important for some classes to only have a single instance, and for there to be a global point of access to that instance
- Solution

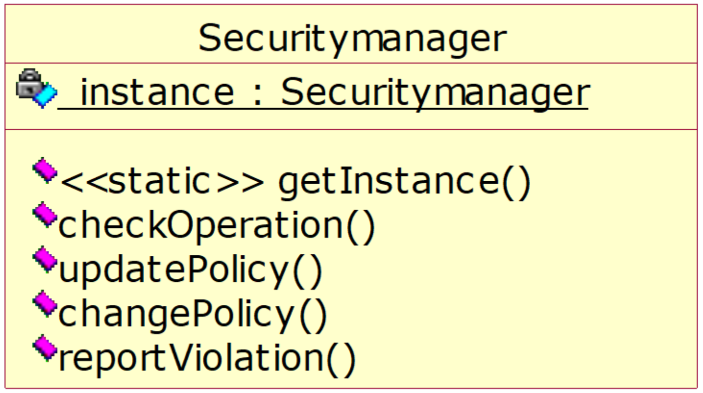


Using Singleton

- The Singleton design pattern presents a generic solution. To use it, just incorporate the pattern solution (as expressed in the class diagram etc.) into your own design.
- For example,

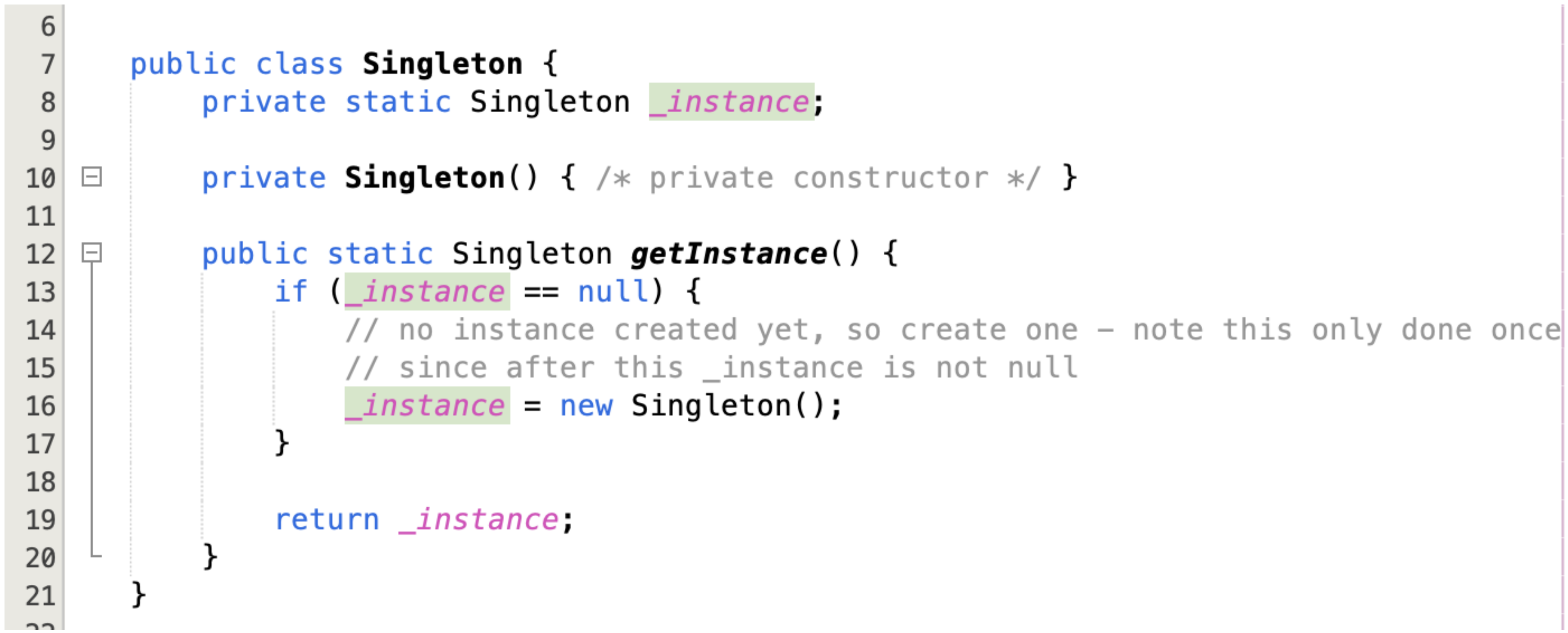


Singleton for Financial Reports
Example



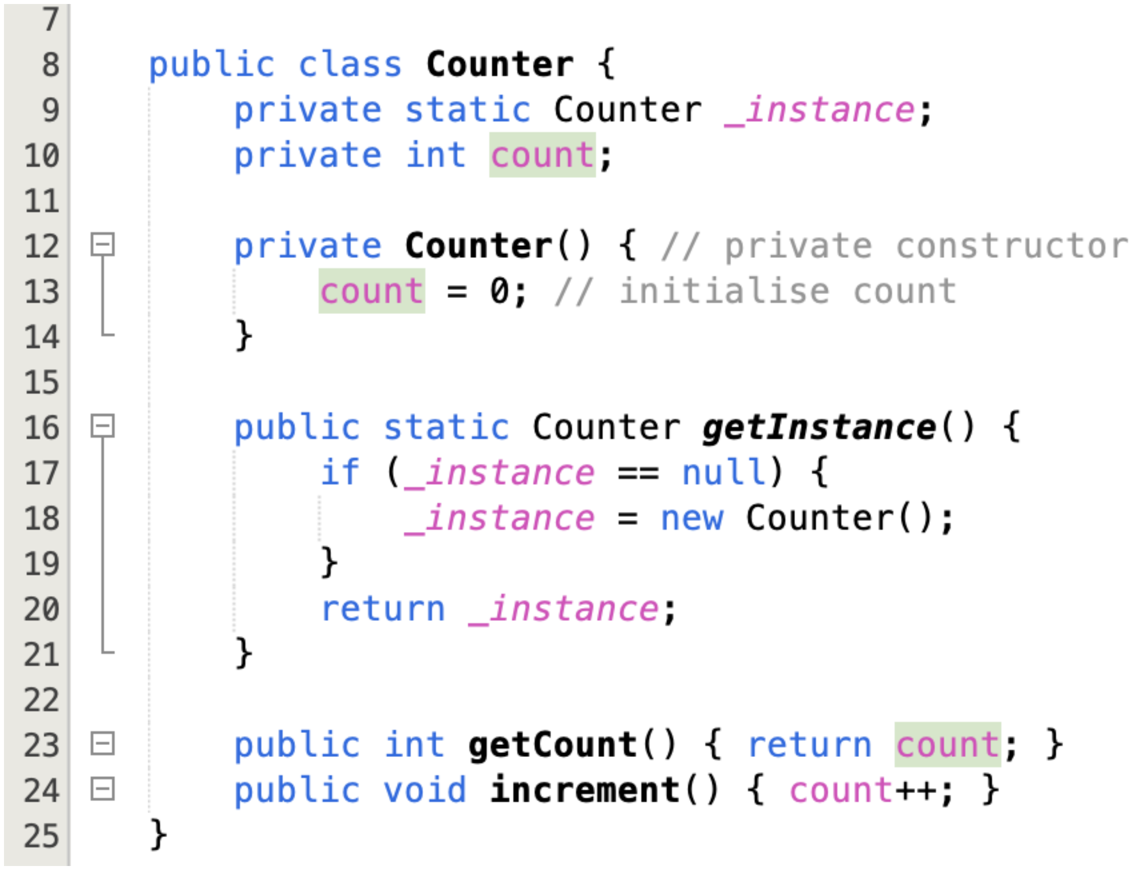
Singleton for Java Security Manager
Example

Implementing the pattern in Java



```java

public class Singleton {
    private static Singleton _instance;

    private Singleton() { /* private constructor */ }

    public static Singleton getInstance() {
        if (_instance == null) {
            // no instance created yet, so create one - note this only done once
            // since after this _instance is not null
            _instance = new Singleton();
        }

        return _instance;
    }
}
```

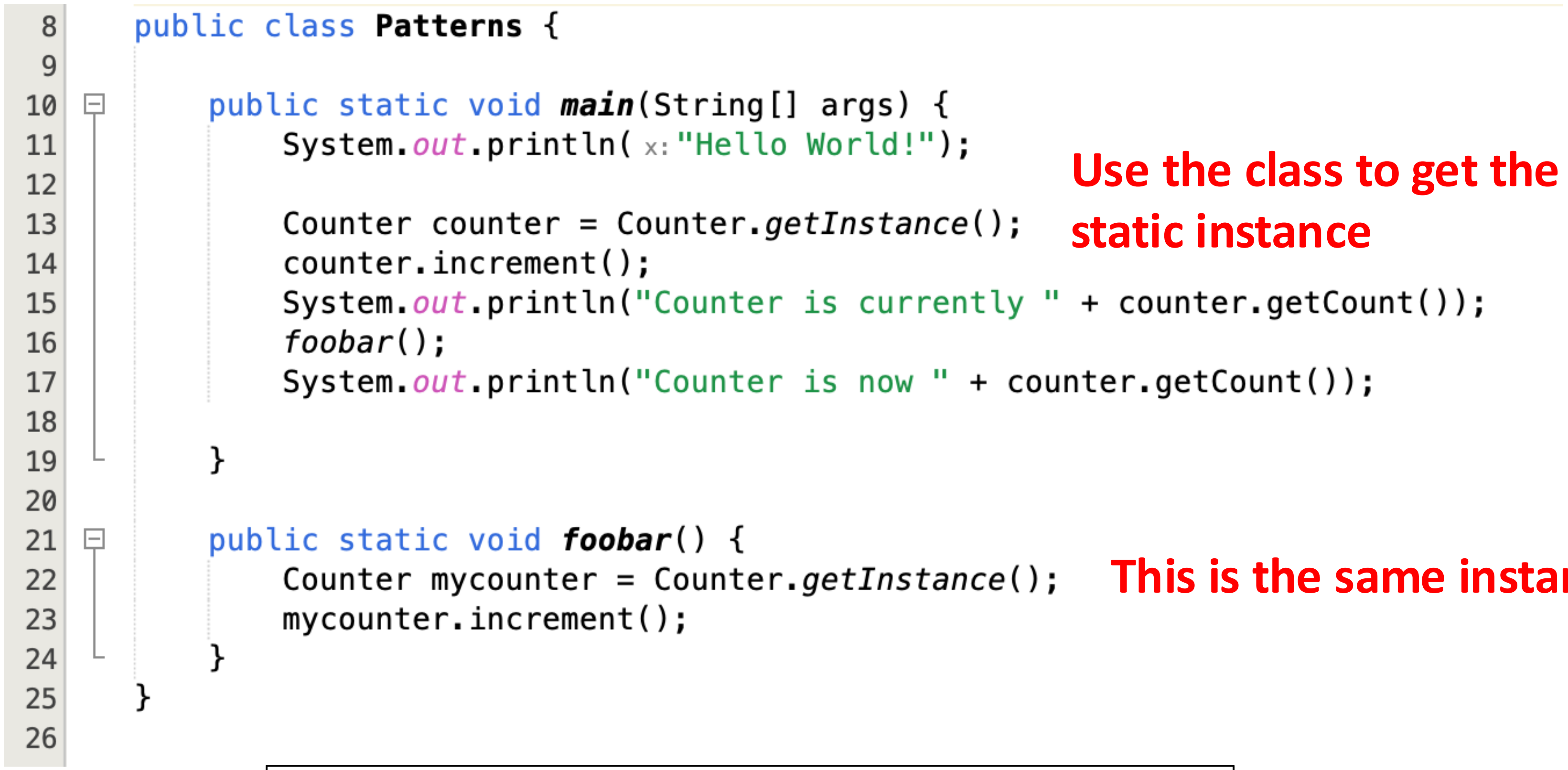
Singleton example



```

public class Counter {
    private static Counter _instance;
    private int count;

    private Counter() { // private constructor
        count = 0; // initialise count
    }

    public static Counter getInstance() {
        if (_instance == null) {
            _instance = new Counter();
        }
        return _instance;
    }

    public int getCount() { return count; }
    public void increment() { count++; }
}
```

Class declaration of Counter class, which is based on the Singleton pattern – note the resemblance between this and the Singleton example

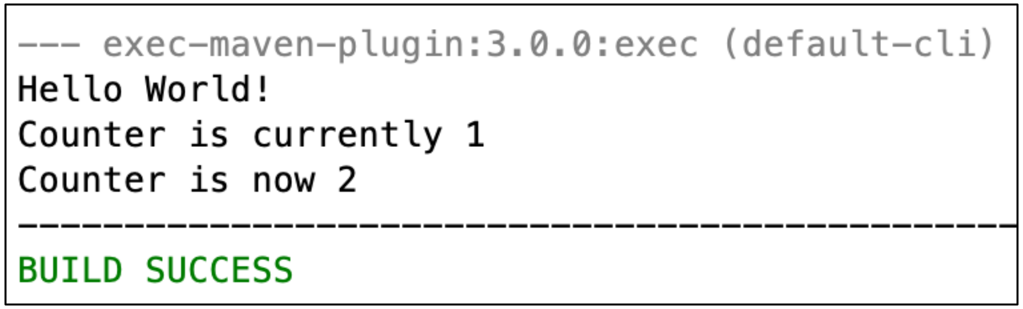
Using the Counter class



```java
public class Patterns {

    public static void main(String[] args) {
        System.out.println( x: "Hello World!");

        Counter counter = Counter.getInstance();
        counter.increment();
        System.out.println("Counter is currently " + counter.getCount());
        foobar();
        System.out.println("Counter is now " + counter.getCount());

    }

    public static void foobar() {
        Counter mycounter = Counter.getInstance();
        mycounter.increment();
    }
}
```

Use the class to get the static instance

This is the same instance



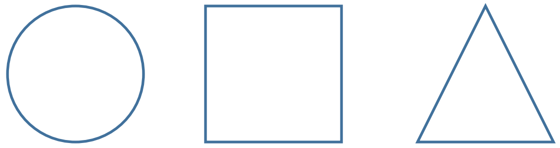
```
--- exec-maven-plugin:3.0.0:exec (default-cli)
Hello World!
Counter is currently 1
Counter is now 2
------------------------------------------------
BUILD SUCCESS
```

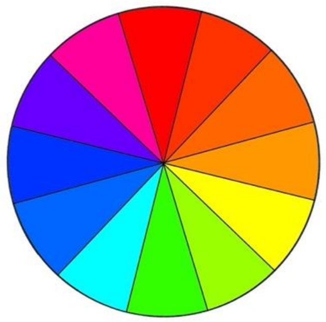
Another Example pattern - Bridge

- Name: Bridge (also known as Handle/Body)
- Purpose: A class may need different implementations, the actual implementation to be decided at run-time. One way to solve this is to have a class hierarchy, with each sub-class providing a different implementation. However, this is inflexible since it binds the class permanently to an implementation.
- Solution: Create a class hierarchy of implementation classes and establish a containment association between the base implementation class and the class defining requiring the implementation.
- Consequences
  - decouple abstraction and implementation
  - improved extensibility

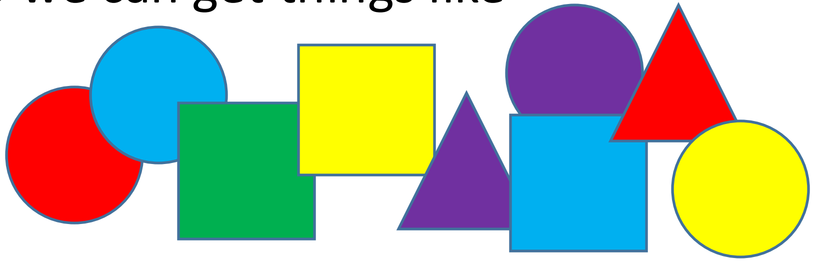
An example problem

- This example is taken from https://refactoring.guru/design-patterns/bridge
- We want to draw different shapes with different colours…



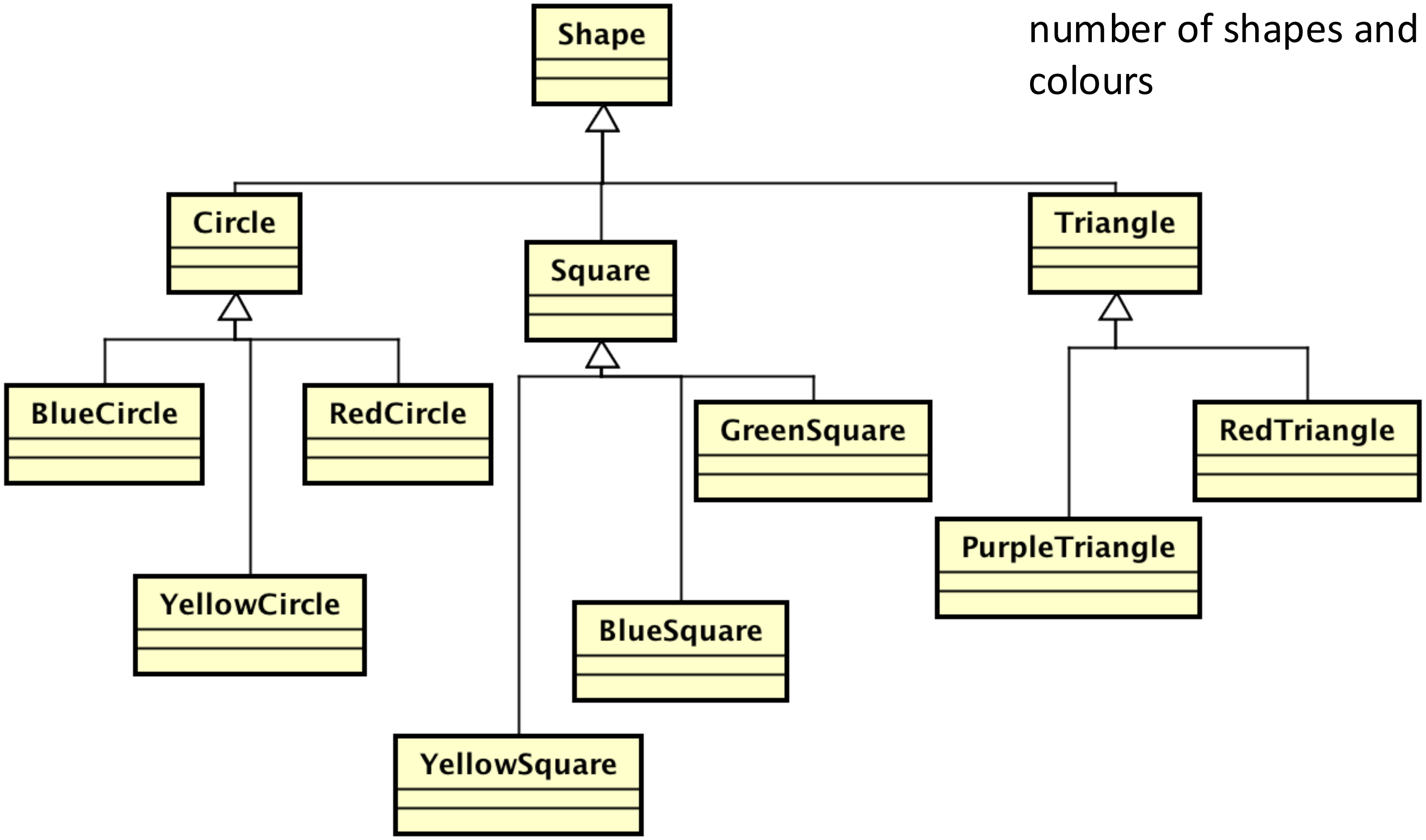


- … so we can get things like

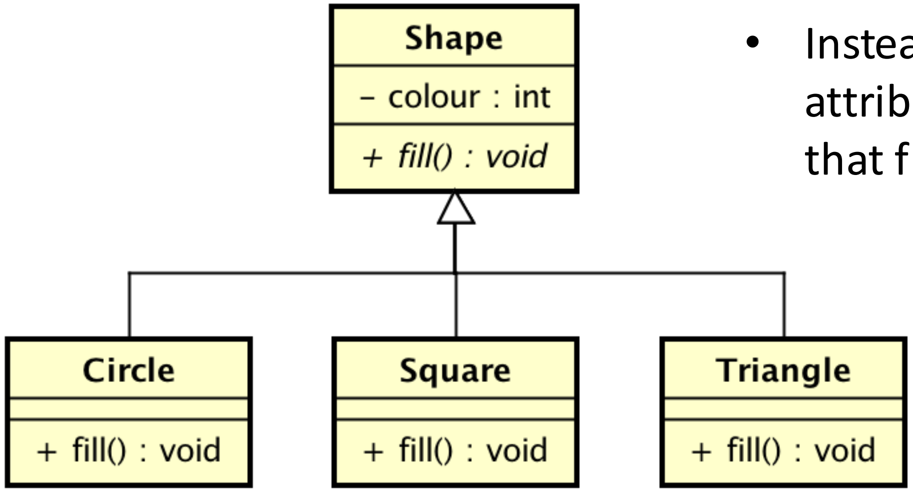


The wrong way to do it…

Too many classes!!!
Even for small
number of shapes and
colours



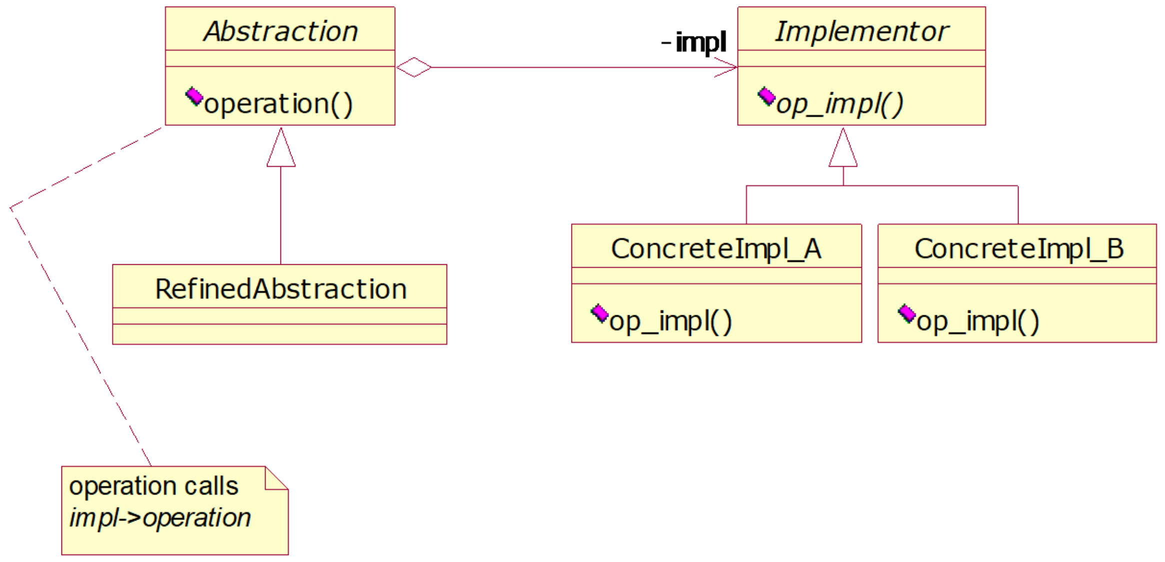
One possible solution



- Instead of using classes, we just use attributes and overload the method that fills the shapes with colour

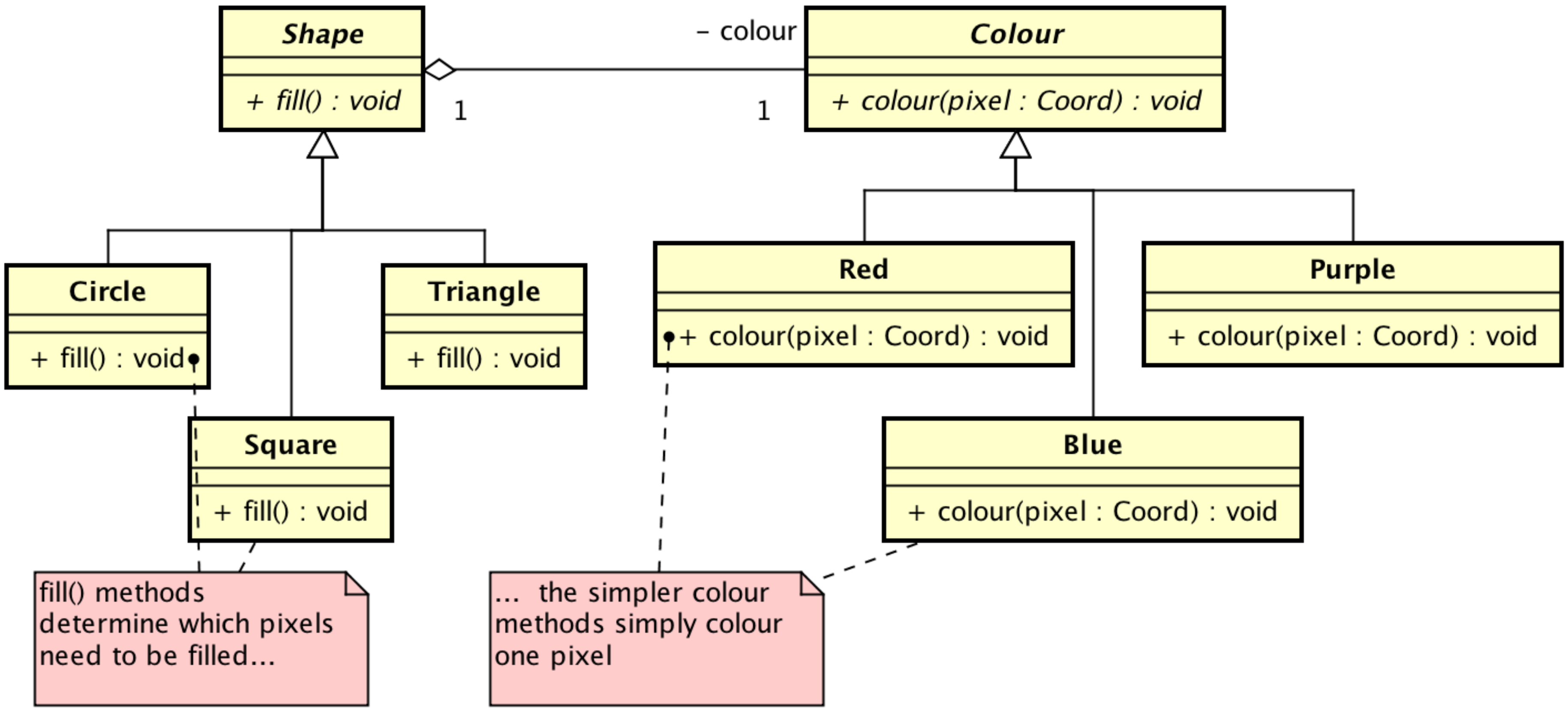
- Disadvantages are:
  - We complicate the implementation of the shape – it must now not just know how to draw the shape but also how to colour it
  - What happens if we want to add other features, such as texture?

A Better way… The design pattern



- Note:
  - The interface and the implementation are in separate classes
  - abstraction classes are those that other clients use
  - abstraction class methods defined in terms of interface on implementation base class
  - Allows us to use a particular interface class and vary the implementation

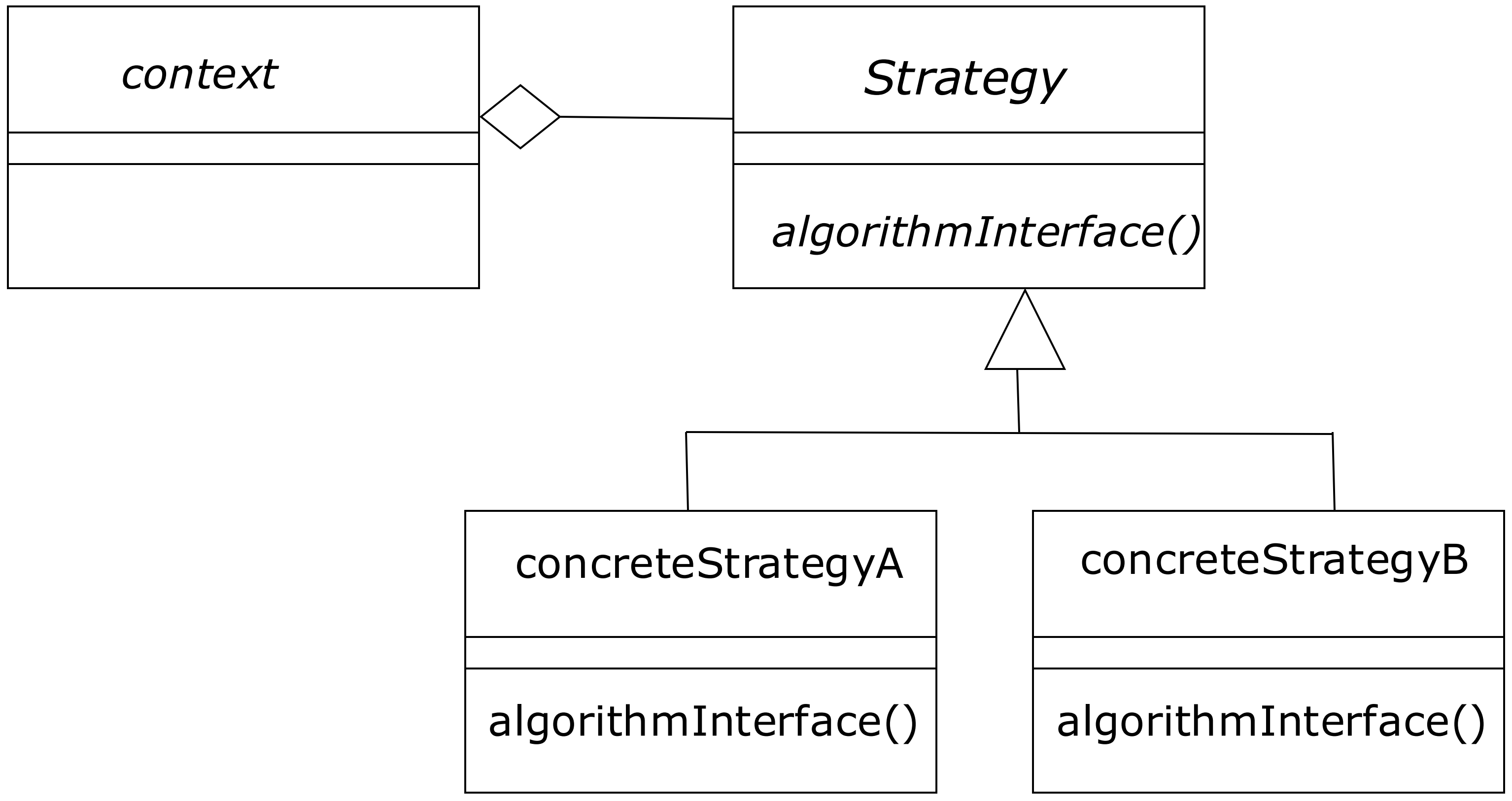
Another possible solution using Bridge


Shape
+ fill() : void
- colour
1
1
Colour
+ colour(pixel : Coord) : void
Circle
+ fill() : void
Triangle
+ fill() : void
Square
+ fill() : void
Red
+ colour(pixel : Coord) : void
Purple
+ colour(pixel : Coord) : void
Blue
+ colour(pixel : Coord) : void
fill() methods determine which pixels need to be filled...
... the simpler colour methods simply colour one pixel

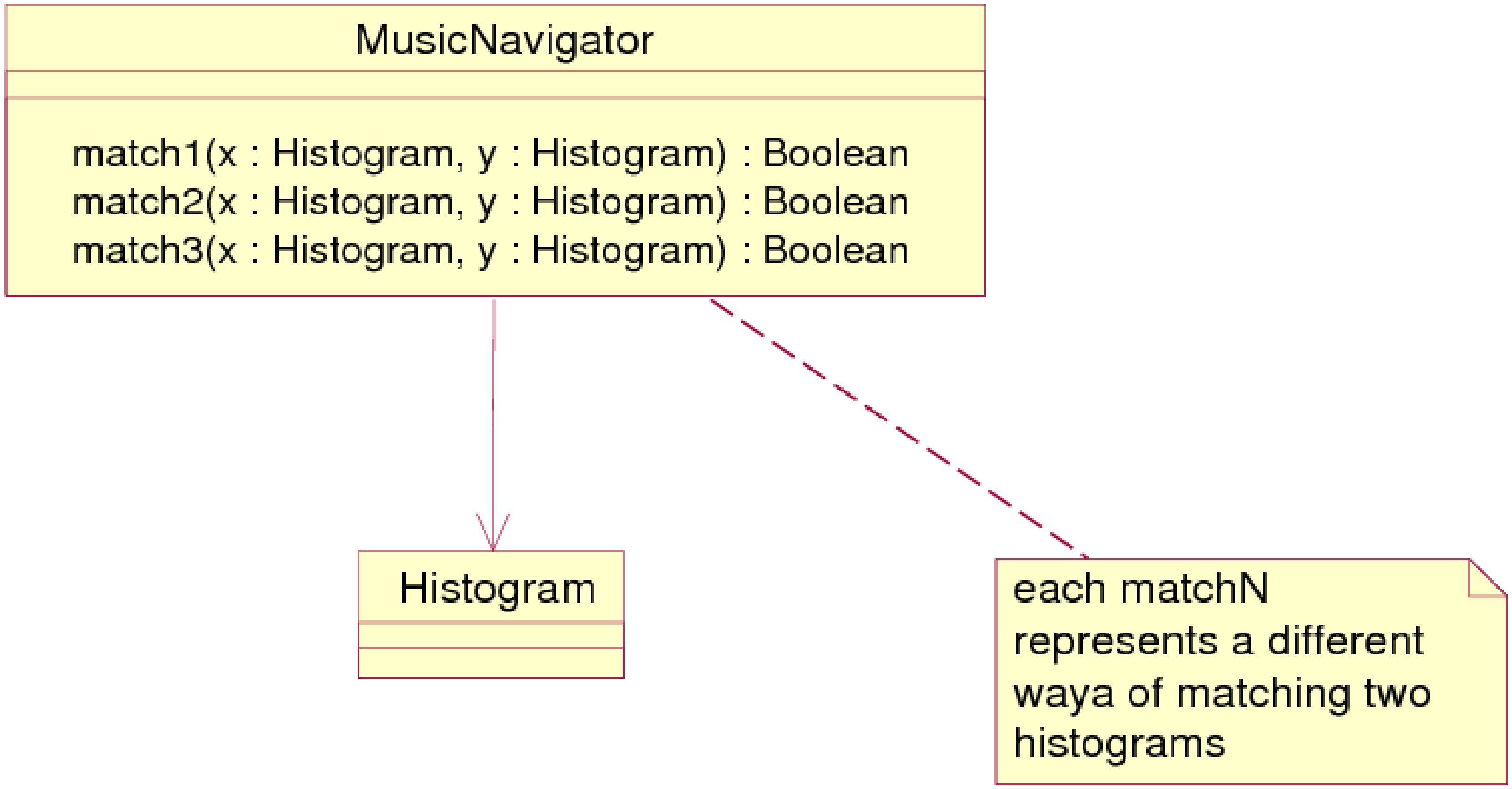
Strategy

- Hard-coding algorithms into a program makes it very inflexible
- For every task, there is usually a choice of algorithms, and in many cases, we want to test a set of algorithms to find out which one is most suited to our purposes
- Strategy is a design pattern that aims to make a set of related algorithms interchangeable, so we can swap them around with the minimum of fuss possible
- We do this by coding the algorithms as separate classes and relate them via generalization to a base class that enforces a common interface to each of the algorithm subclasses
  - Can treat them interchangeably through polymorphism

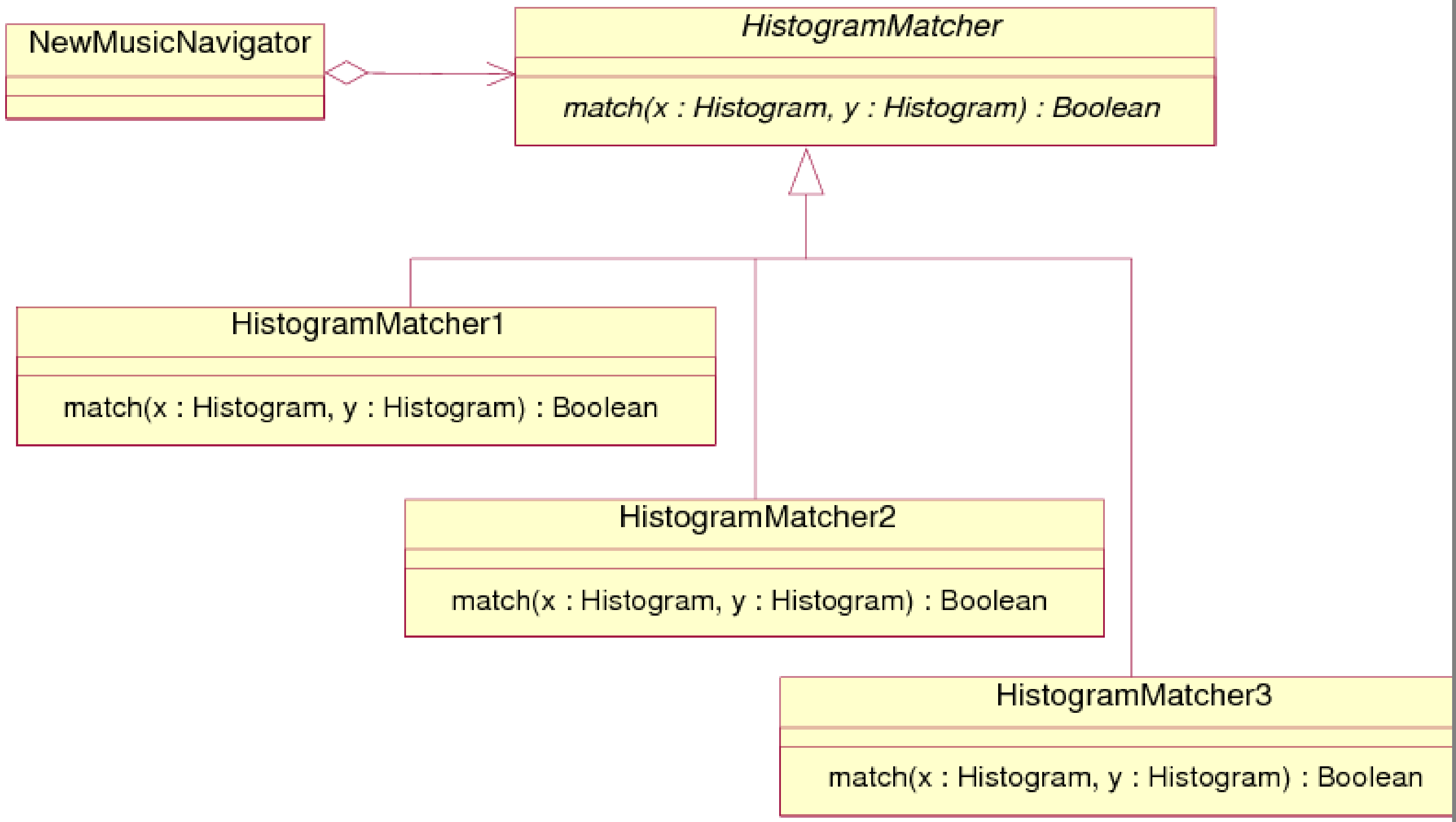
Strategy solution



Example of related algorithms without Strategy


MusicNavigator
match1(x : Histogram, y : Histogram) : Boolean
match2(x : Histogram, y : Histogram) : Boolean
match3(x : Histogram, y : Histogram) : Boolean
Histogram
each matchN represents a different waya of matching two histograms

Example of related algorithms USING Strategy!


NewMusicNavigator
HistogramMatcher
match(x : Histogram, y : Histogram) : Boolean
HistogramMatcher1
match(x : Histogram, y : Histogram) : Boolean
HistogramMatcher2
match(x : Histogram, y : Histogram) : Boolean
HistogramMatcher3
match(x : Histogram, y : Histogram) : Boolean

Elements of a design pattern

- The structure of a design pattern has 4 components:
- Name - a brief phrase that sums up the purpose of the pattern
- Problem - a description of the problem to be solved, plus some context in the way of examples of the problem
- Solution - a template solution: the elements of the solution (classes, objects, relationships) in general terms, usually accompanied by a class diagram
- Consequences - a discussion of the costs and benefits of using the pattern

Benefits

- Design patterns are a representation of best practice and accumulated wisdom. Design patterns ...
  - mean we don't have to reinvent everything from first principles - instead we can apply the solutions of the best designers and developers
  - create a common vocabulary which designers and developers can use in communicating features of designs
  - emphasize composition of objects, rather than object hierarchies
    - Why is this important?
    - Surely generalization and inheritance are the most important parts of an OO design???

Composition vs. Inheritance

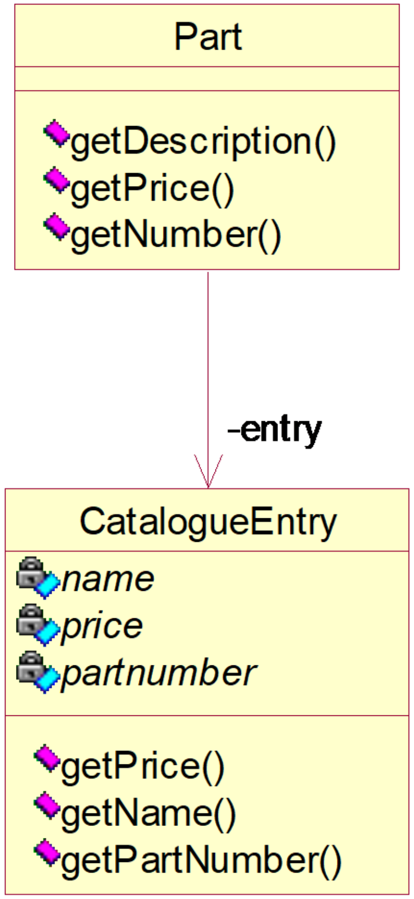
- The question of composition vs. inheritance is an issue concerning software reuse. Software is often easier to reuse when it is structured using containment associations (aggregation or composition), rather than just generalization
- Inheritance hierarchies are static structures, fixed at compile-time, and are therefore easy to understand. But they mean that subclasses are permanently fixed to the implementation of their parent classes. They *may* also allow subclasses to have access to the implementation details of parent classes (through protected access), breaking encapsulation.
- Containment associations
  - force objects to *respect each other's interfaces*
  - allow objects to be swapped for each other at run-time, provided they're of the same type (generalization may play a role here to create polymorphism!)

Delegation

- Delegation allows object composition to be as powerful as class inheritance in building the functionality of a system.
- In delegation, two objects are involved in handling a request: an object that receives a request (i.e., a method is called on it) *delegates* the request to its delegate.
- This is equivalent to a subclass deferring request to parent classes.

Example of Delegation

- The Stock Control example shows an example of delegation
- The functions in the Part class delegate the request to the corresponding function in the CatalogueEntry class
- Usually the class that is delegated to has a simpler interface than the one which is doing the delegation
  - The methods of the delegating class might use one or more of the methods of the class delegated to in order to implement their functionality
- The benefit is that we can swap the class being delegated to (with some minor recoding of the delegating class) without needing to change the interface of the class doing the delegation

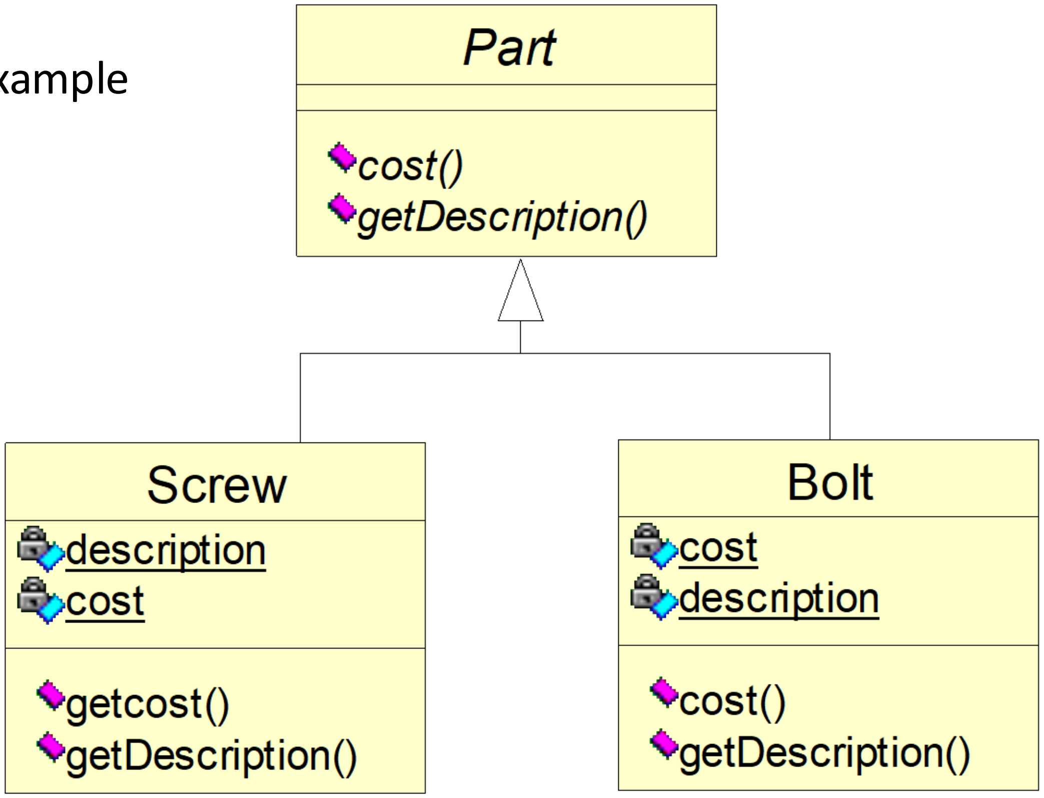


Drawbacks

- The disadvantage of using composition and delegation is that the compile-time structure of a program doesn't accurately correspond to the run-time structure of the program
  - hence, just by reading the code (which is static), you cannot easily understand the run-time behaviour of a system
- Delegation also introduces performance inefficiencies, through following chains of references/pointers and method calls.

Not Using Delegation

- What if we did not use delegation
- This class diagram shows how we might do that
- It doesn't need CatalogueEntry – instead it uses class inheritance and static attributes to solve the problem
- But is it as "good" as the example that uses composition and delegation?
- If not – why not?



Design patterns

- Design patterns can be classified according to their use or purpose
- For example:
  - Creational patterns: those patterns that help solve problems to do with creating objects
    - E.g., Singleton, Abstract Factory
  - Structural : "Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. " (GoF book)
    - E.g. Composite
  - Behavioural : focus on object interaction
    - E.g., Chain of Responsibility, Strategy, Bridge

Creational problems

- Problem: specifying object implementations
- An object has a class and a type associated with it. The two are related but different. An objects implementation is specified by its class
  - the class defines the internal structure of the object, and the methods it supports
- The type of an object, however, only specifies its interface. An object can have many types, and many objects of different classes can have the same type.
  - In C++: abstract base classes
  - In Java: abstract base classes or interfaces
- The benefit of programming to types, rather than to classes is that we gain maximum flexibility, by reducing implementation dependencies between classes/packages/modules/subsystems etc.