MSO
Object Creation
Singleton & Object Pool

Wouter Swierstra & Hans Philippi

October 25, 2018

This lecture

How to create objects

- The Singleton Pattern
- The Object Pool Pattern

Categories of design patterns

The Gang of Four distinguish three categories of design patterns:

- **Behavioural** patterns characterize the way in which classes or objects interact or distribute responsibility.
  - Observer, Strategy
- **Structural** patterns deal with the composition of classes or objects.
  - Adapter, Bridge, Decorator
- **Creational** patterns are concerned with the creation of new objects.
  - Abstract Factory

Why worry about creation?

- A lot of the design we have focussed on so far in this course is concerned with figuring out *how* classes should work together, and how to encapsulate variation – hiding implementation details
- But if we hide implementation details behind an abstract class, who decides which concrete instances should be made?

Factories

- Separating *creation* and *usage* leads to stronger cohesion
- Separate two distinct tasks:
  - Defining classes and how they work together
  - Writing factories that instantiate the correct objects for the right situation

Not everyone agrees...

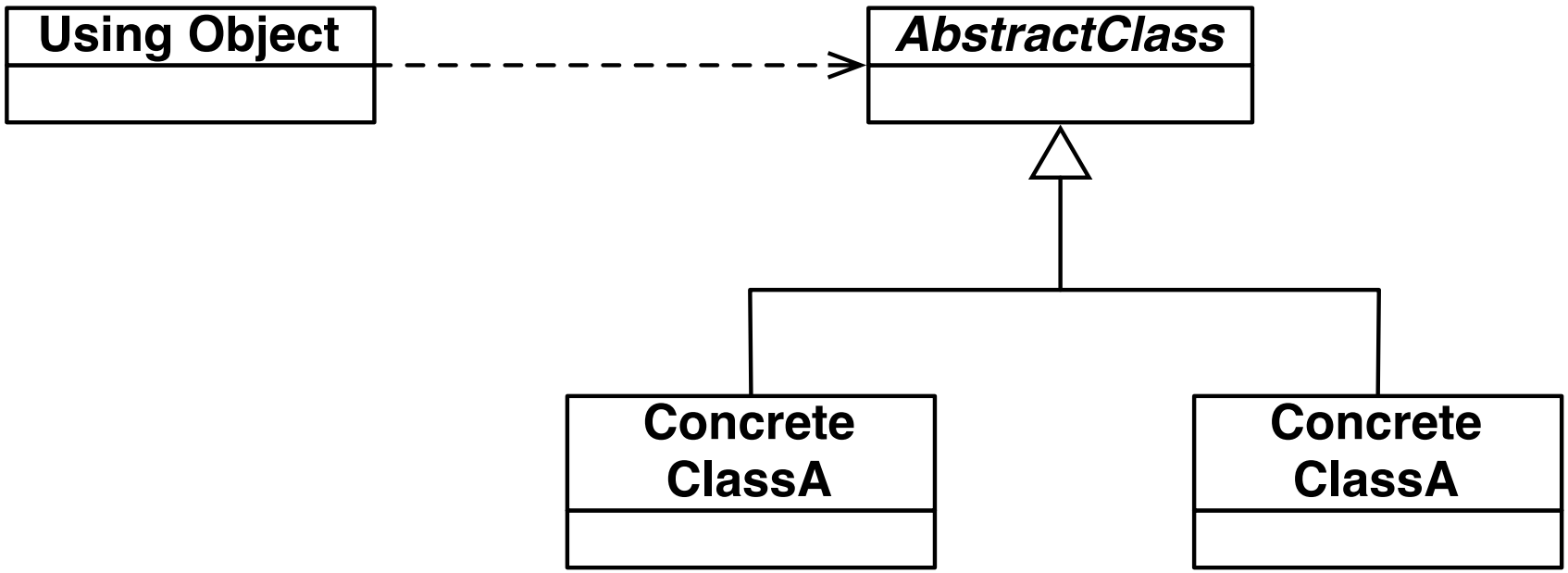
- Remember Larman's Creator GRASP principle:
- *Assign class A the responsibility to create instances of a classB if A has a B object*
- This is a good guideline for simple situations, provided you are not programming to an interface
- If you are, you may need to consider applying creational design patterns

Creation vs Use

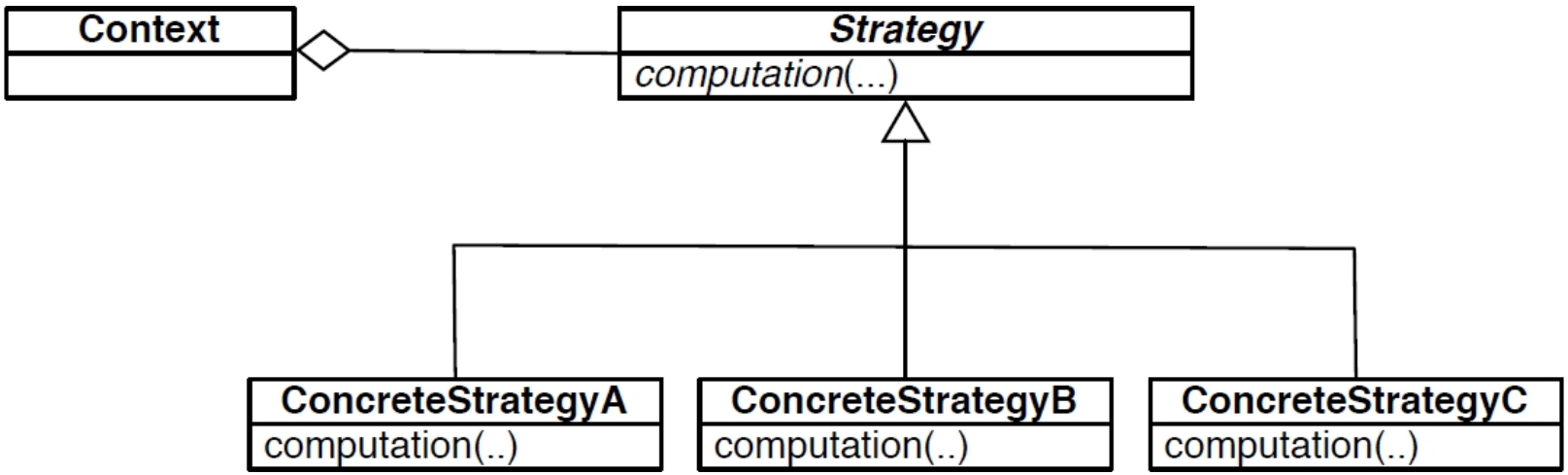
- Shalloway and Trott say that an object should either:
- . . . make and/or manage other objects
- . . . or it should use other objects
- It should never do both
- Following this rule leads to looser coupling

“Using Objects”

”Program to an interface, not to an implementation”
”Find what varies, and encapsulate it”

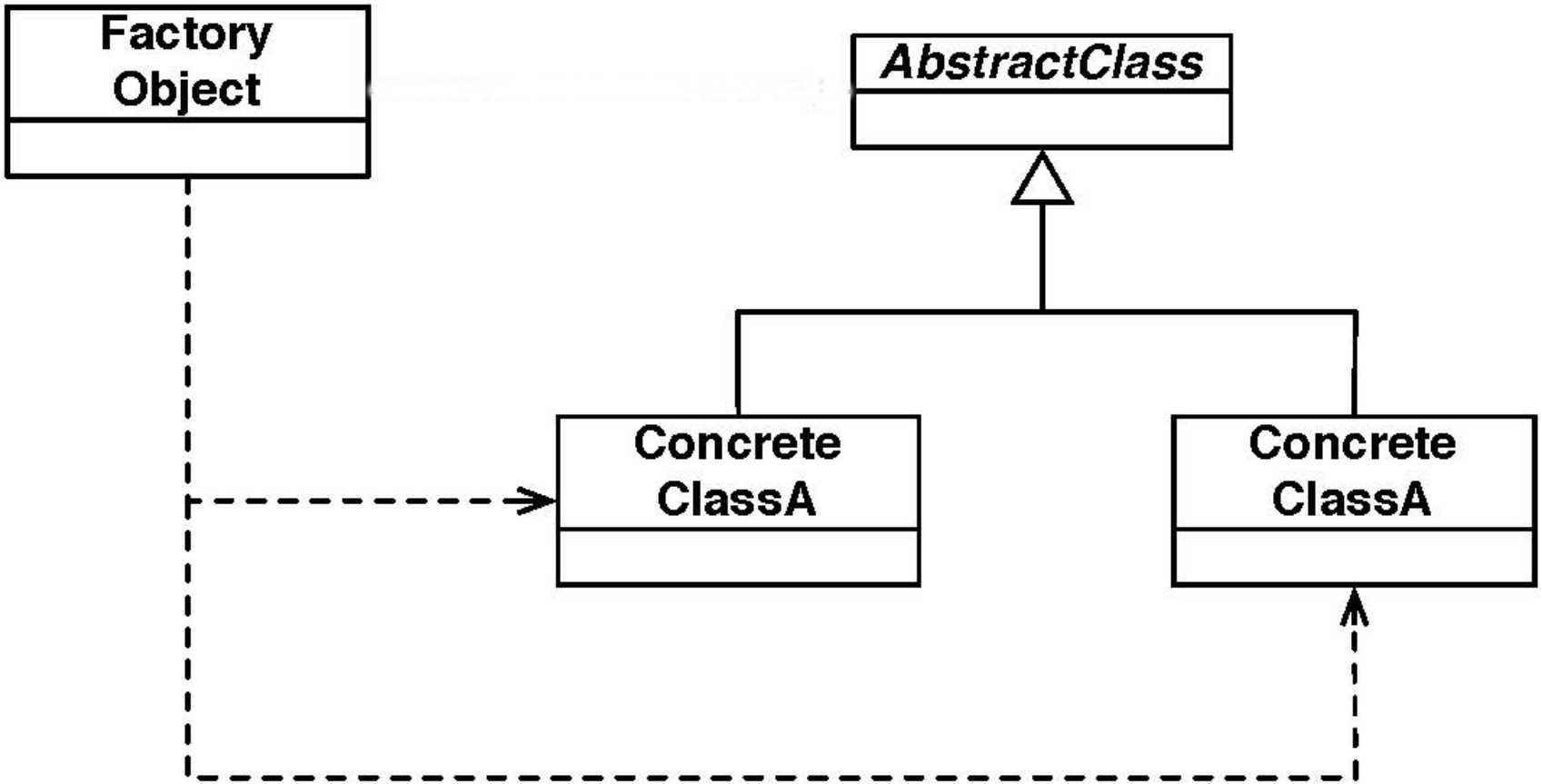


Example: strategy



- If the *Context* creates concrete strategies, it is breaking an abstraction barrier
- Instead, we want to assign the *responsibility* for *creation* to another object

Factory objects



Compare figure 20-3 on page 352 of S&T

Using vs creating objects

- The Factory can create values of type *AbstractClass* on request; it stores all the creational logic and creates a concrete object
- Using objects means knowing only about the interface, not the implementation
- The Factory objects should only know how to create concrete instances; they should not rely on the details of the abstract class/interface
- We are free to add new concrete classes as necessary – we know that the client objects should never inspect which concrete class they are using – the Open-Closed principle!

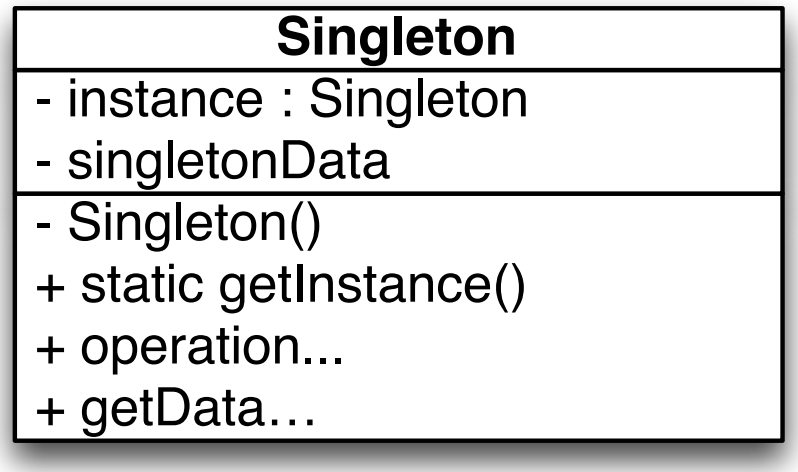
Summary

- Patterns add indirection (hide information behind an abstract class, move functionality to new classes, . . .)
- This makes code more *maintainable*, but adds *complexity*, in case the client code must know about these constructions
- Factories and other creational patterns are designed to hide the complexity that patterns sometimes introduce

The Singleton Pattern

- Factories can be used for more than just controlling which concrete instances to create
- For instance, factories may implement constraints
- The *Singleton Pattern* for example, can be used to guarantee that exactly one object exists of a particular type

The Singleton Pattern



The Singleton Pattern

```
public static class Singleton
{
   private static Singleton instance;

   private Singleton() { ... }

   public static Singleton getInstance()
   {
         if (instance == null)
         {
            instance = new Singleton();
         }
         return instance;
   }
}
```

Singleton: example

- One example of a Singleton could be the CalcTax strategy for a specific country
- Another example of a Singleton could be a logging service:
- different objects want to be able to log a message
- . . . where the log should be shared by all

Caution!

- Singletons are not thread safe
- If you have multiple threads trying to create a singleton at the same time, you may create two (or more) Singleton objects

Solution: The Double-Checked Locking Pattern addresses this, but we will not cover it in this course

More caution!

- Singletons are a controversial pattern
- The Singleton is regarded to be an 'over applied' pattern - it is often not even needed
- To decide whether a class is truly a singleton, you must ask yourself some questions (Rainsberger 2001):
  - Will every application use this class *exactly* the same way?
  - Will *every* application ever need only *one* instance of this class?

So much for singletons

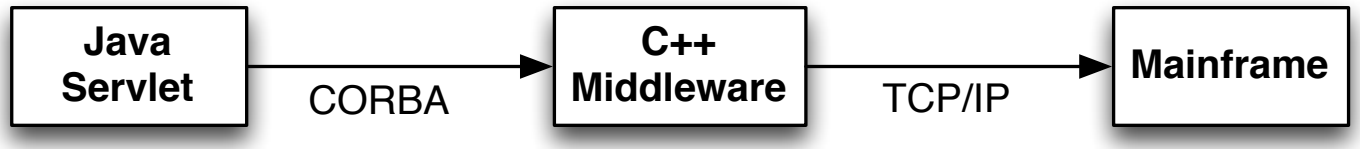
Case study: electronic banking

Today let's suppose we need to write software for online investment services, like alex.nl, that allow users:

- to browse their investment portfolio
- to give orders for purchasing or selling stocks
- ...

We will see that Factory tasks may be a bit more involving than just creating objects!

Architecture



Requirements

- The only way to communicate with the mainframe is through TCP/IP connections using a custom messaging protocol
- The C++ (or C#, or ...) middleware is responsible for verifying user orders, before they are executed by the mainframe

Example interaction

- User logs in through their webbrowser
- Middleware gets user ID and password; it reformats message in a format the mainframe can understand
- Mainframe accepts the request and responds accordingly

Performance concerns

- The middleware needs to handle requests from many, many people at the same time
- We should aim at handling as many transactions as possible - maximizing throughput - and not worry about the time a single transaction needs

Design question

How many TCP/IP connections should I use?

- 1 is too little – all traffic goes over a single connection
- 100 is too much – the bandwidth available could not handle more than 20

This is a *high-risk* issue; the performance of the entire system can rely on it

Issues involved

- Establish a single, robust TCP/IP connection
- Determine the number of TCP/IP connections that are necessary
- Establish a method to load balance the connections – we don't want to route all traffic through a single connection, while the others are idle

What to do first?

Design choice

- Start by establishing a single connection
- In other words, start with a single responsibility, but don't forget that you will have to address these other issues soon
- Open-closed: insulate yourself from change, but anticipate new requirements
- We need to design the system so that we can easily *change* the number of connections used
- . . . but that is another responsibility

TCP/IP management

Distinguish separate responsibilities:

- A **Port** class is responsible for communicating with the mainframe
- A **PortManager** class is responsible for keeping track of the **Port**s

How many?

- There should be multiple **Port** objects
- The **PortManager** should be able to change the exact number easily
- There should only be a *single* **PortManager** object – otherwise we don't have control over how many connections are established exactly
- So here we use the *Singleton* pattern!

Functionality

The **PortManager** supports has the following methods and attributes:

- a private array of **Port** objects
- GetInstanceOfPort() - which looks through the array to find an inactive port; if no port is inactive, it sleeps and retries
- ReturnInstanceOfPort(Port release) - sets the status of the argument port to inactive

Using Ports

Client code is now straightforward:

1. Ask the **PortManager** for a **Port** object
2. Complete the transaction
3. Release the **Port** to the **PortManager**

The client code does not need to know anything about how many ports are active, setting up connections, etc

Strong cohesion, loose coupling

- This solution has strong cohesion and loose coupling – the **Port**, **Client**, and **PortManager** classes all have a clear set of responsibilities
- What happens when I need to add error handling?

Adding Error Handling

What happens when a connection goes down?

1. The **Client** code needs to request another **Port**
2. The **Port** that went down should be closed
3. The **PortHandler** may want to establish a new connection to replace the **Port** that went down

It is clear who is responsible for handling this exceptional scenario

Are we done?

If you're responsible for developing a system that handles millions of dollars worth of transactions, will you sleep easy at night?

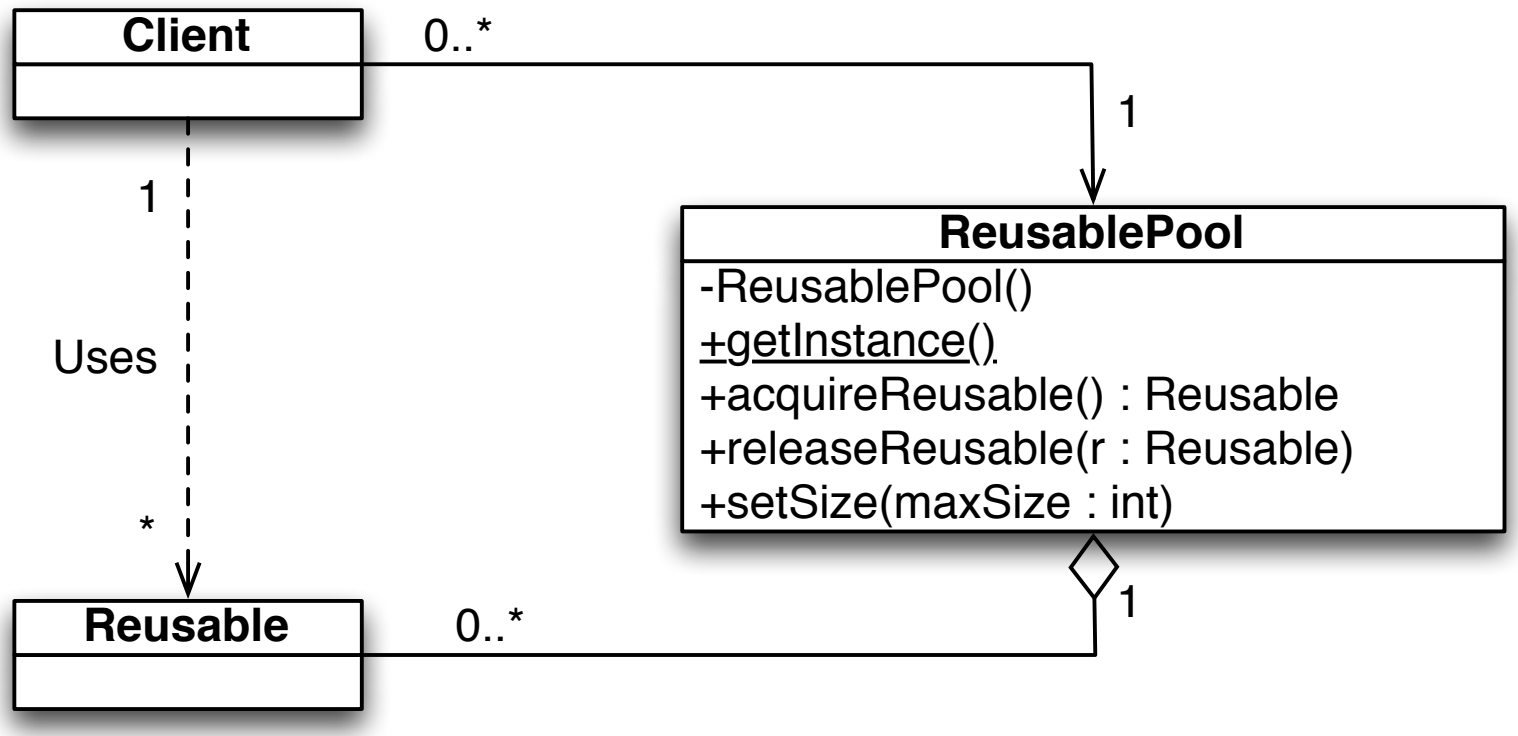
Add checking

- Maybe we would sleep easier if the **PortManager** also checked the integrity of the connection
- Every 15 (or so) minutes it should:
  1. Check how many unanswered requests there are for a **Port**
  2. Query all the **Ports** to see if they are active or not
  3. Check how many **Ports** have run into an error
- If any of these checks produces unexpected results, warning bells should go off

Even better...

- Don't stop there!
- It's not enough to know that a system is not failing
- Perhaps you cannot detect connection problems, but people still are not able to use the website for other reasons
- Log successful transactions

The Object Pool Pattern



Note that `getInstance()` refers to the creation of a singleton ReusabePool

The Object Pool Pattern: Lessons

Factories are not just about instantiation!