

COURSE NAME

OBJECT ORIENTED
ANALYSIS AND DESIGN

CSC 2210

(UNDERGRADUATE)



CHAPTER 10 DESIGN PATTERN

VICTOR STANY ROZARIO

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE, AIUB

Web: <https://cs.aiub.edu/profile/stany>



WHAT IS DESIGN PATTERN?

- ❑ A design pattern describes a problem that occurs over and over in software engineering
- ❑ And then describes the solution in a sufficiently generic manner as to be applicable in a wide variety of contexts.

TYPICAL PATTERN FORMAT

- ❑ **The pattern name**
- ❑ **The problem**
 - Specification of the problem
 - Explanation why it is important
 - Applications
 - Examples of known uses

SOFTWARE MEASUREMENT

❑ The solution

- A description of classes possibly with a structure diagram
- A language independent implementation, with language-specific issues as appropriate
- Sample code

❑ Consequences

- Results
- Trade-offs of using the pattern
- A discussion of related patterns

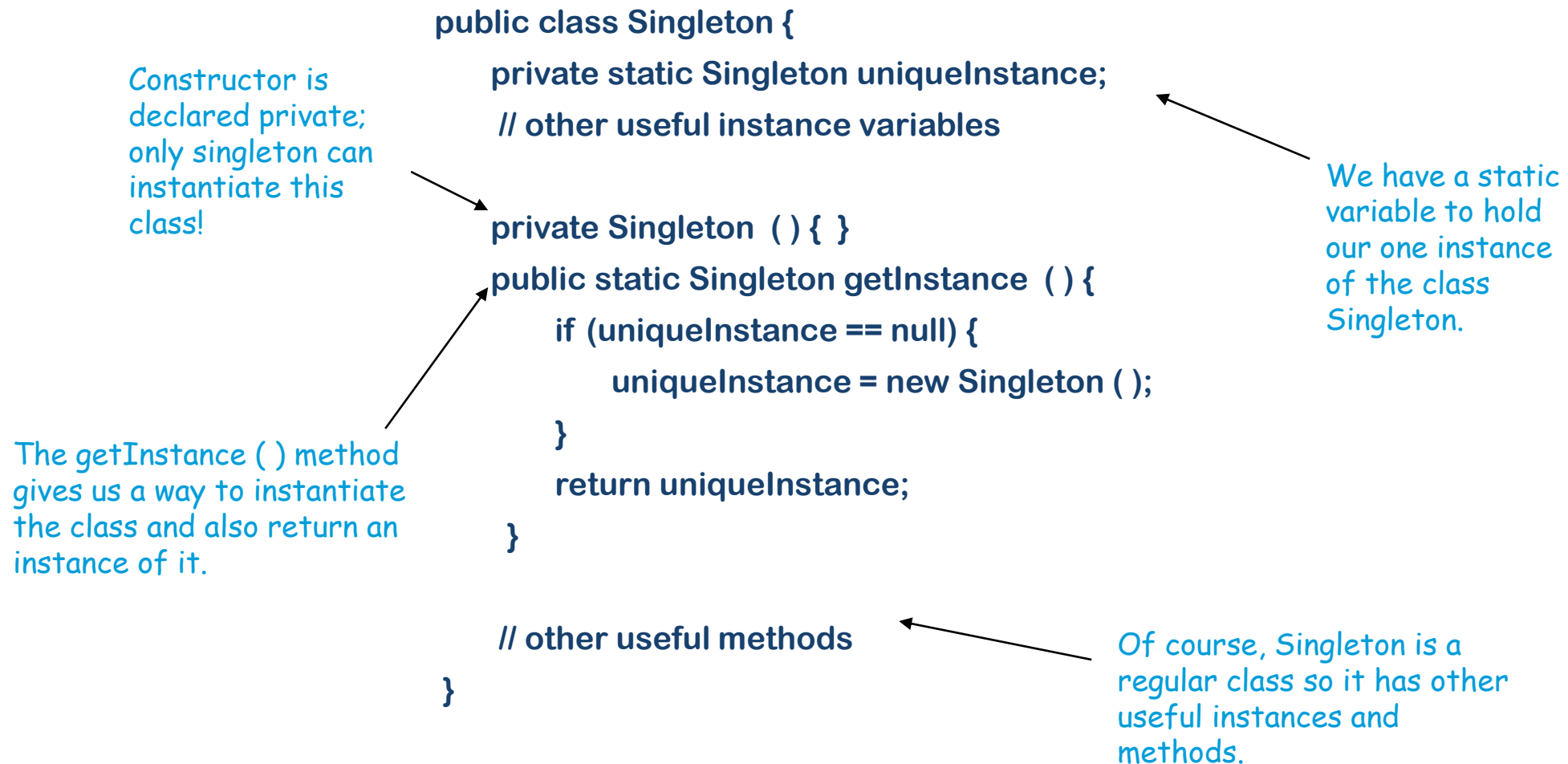
SINGLETON

- ❑ The Singleton pattern ensures you have at most one instance of a class in your application (One of a Kind Objects)
- ❑ For example, in a system there should be only one window manager (or only a file system or only a print spooler)
- ❑ If more than one instantiated: Incorrect program behavior, overuse of resources, inconsistent results

THE LITTLE SINGLETON

How would you create a single object?	<code>new MyObject ();</code>
And what if another object wanted to create a MyObject ? Could it call <code>new</code> on MyObject again?	Yes.
Can we always instantiate a class one or more times?	Yes. Caveat: Only if it is public class
And if not?	Only classes in the same package can instantiate it - but they can instantiate it more than once.
Is this possible? public class MyClass { private MyClass () { } }	Yes. It is a legal definition
What does it mean?	A class that can't be instantiated because it has a private constructor

THE CLASSIC SINGLETON PATTERN



ADAPTER

- ❑ An adapter is used when you need to use an existing class, but its interface is not the one you need
- ❑ An adapter changes an interface into one that a client expects.

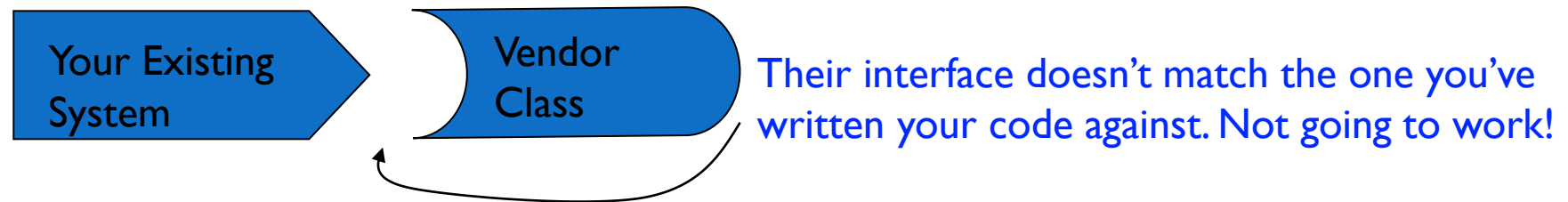
“Putting a Square Plug in a Round Socket”



- ❑ Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- ❑ An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities.

ADAPTER

- **Scenario:** you have an existing software system that you need to work a new vendor library into it, but the new vendor designed their interfaces differently than the last vendor
- What to do?

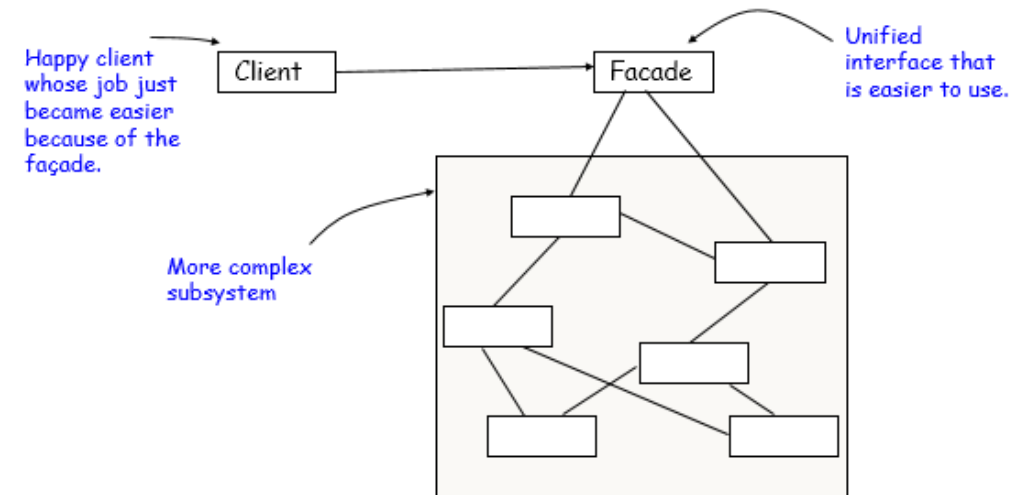


- Write a class that adapts the new vendor interface into the one you're expecting



FAÇADE

- ❑ A façade is used when you need to simplify and unify a large interface or a complex set of interfaces
- ❑ The **Façade Pattern** provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use
- ❑ A façade decouples the client from a complex subsystem
- ❑ Implementing a façade requires that we compose the façade with its subsystem and use delegation to perform the work of the façade
- ❑ A façade “wraps” a set of objects to simplify
- ❑ This pattern hides all the complexity behind



OBSERVER

- ❑ **Observer pattern** is used when there is **one-to-many relationship** between objects such as if one object is modified, its dependent objects are to be notified automatically
- ❑ **Example:** when you subscribe to your local newspaper agent, every time there is a new edition, it gets delivered to you
- ❑ It is mainly used to implement distributed event handling systems

STRATEGY

- ❑ **Strategy pattern** allows selection of one of several algorithms dynamically
- ❑ In Strategy pattern, a class behavior or its algorithm can be changed at run time.
- ❑ Strategies don't hide everything -- client code is typically aware that there are a number of strategies and has some criteria to choose among them -- shifts the algorithm decision to the client.
- ❑ In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

FACTORY

- ❑ When a method returns one of several possible classes that share a common super class
- ❑ Class is chosen at run time
- ❑ Factory pattern allows to create objects without specifying the exact class of object that will be created

REFERENCES

- R.S. Pressman & Associates, Inc (2010). *Software Engineering: A Practitioner's Approach*.