Design with Software Design Patterns

Remember the **problems** with software development:

- Software systems are **complex**.
- Large software systems include **many components**.
- Software systems tend to have a long life span. Requirements **change**.
- As a consequence, maintenance cost is too high.

Therefore, we need **flexible** (modifiable, extensible), **reusable**, and **maintainable** software.

Knowledge in the field of OOP (coding) is not sufficient.

Design skills are also necessary.

- "Programming is fun, but developing quality software is hard." *Philippe Kruchten*
- "Designing object-oriented software is hard, and designing reusable object-oriented software is even harder."
  Erich Gamma

Dieter Rams' ten principles of "good design":

Dieter Rams (1932-) is a German industrial designer.

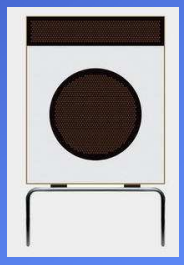
His principles are not directly related to software development.

But most of them are also applicable to the world of software.

Principles:

1. Good design is innovative.
2. Good design makes a product useful.
3. Good design is aesthetic.
4. Good design makes a product understandable.
5. Good design is unobtrusive.
6. Good design is honest.
7. Good design is long-lasting.
8. Good design is thorough down to the last detail.
9. Good design is environmentally friendly.
10. Good design is as little design as possible.

Cylindrical T 2 lighter, 1968



L 2 speaker, 1958

RT 20 tischsuper radio, 1961,

Source:
http://www.vitsoe.com/en/gb/about/dieterrams

Design Patterns

The Starting Point for Design Patterns:

Design patterns were introduced by architect Christopher Alexander[1,2] in the field of architecture.

Questions:

What makes us know when an architectural design is good?
Can we know good design?
Is there an objective basis for such a judgment?

Alexander postulates that the judgment that a building is beautiful (well designed) is not simply a matter of taste.

We can describe beauty on an objective basis that can be measured.

He studied the problem: "*What is present in a good quality design that is not in a poor quality design?*"

---

[1] Alexander, C., Ishikawa, S., Silverstein, M., *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[2] Alexander, C., *The Timeless Way of Building*, Oxford University Press, 1979.

©2012 - 2023 Feza BUZLUCA

The Starting Point for Design Patterns (contd)

Alexander observed buildings, towns, streets, and virtually every other aspect of living spaces that humans have built for themselves.

He discovered that, for a particular architectural creation, suitable constructs had things in common with each other.

Structures that solve similar problems (schools, hospital buildings, streets, gardens, etc.), even though they look different, they have similarities if their designs are high quality.

He called these similarities ***patterns***.

He defined a pattern as "a solution to a problem in a context."

Each pattern describes a problem, which occurs 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 ever doing it the same way twice.

The Works of Alexander have influenced the World of software development and led to the creation of software design patterns.

©2012 - 2023 Feza BUZLUCA

Software Design Patterns

What:

- Software design patterns are our guidelines for making decisions at the design level.
- Software designers face common (similar) problems in different projects.
- Experienced designers reuse solutions that have worked in the past.
- Patterns describe solutions discovered by experienced software developers for common problems in software design.
- A **software design pattern** is a named and well-known problem/solution pair that can be applied in new contexts.

Why:

- Using patterns allows designers to create **flexible** and **reusable** designs.
- Names of design patterns are also crucial as they constitute a new vocabulary (**common language**) for designers.
  A single word or noun phrase (controller, pure fabrication, adapter, strategy, observer, etc.) can express many information pages.

Which:

- There are many software pattern sets.
- In this course, we will cover famous GoF design patterns.

GoF Design Patterns

GoF (*Gang of Four*) patterns are introduced by the book written by four authors.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading MA, Addison-Wesley, 1995.

The book includes 23 patterns; 15 of them are used more frequently.

GoF patterns are grouped into three3 categories:

Creational Patterns:
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns:
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns:
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

The Adapter Pattern (Structural)

The Adapter Pattern converts the interface of a class into another interface the client expects.

Motivation 1:

We already have a class that can perform the required job.

This class may have been designed in a previous project, it can be provided by the development environment in its library, or we may have bought such a class.

However, we cannot reuse this class because its interface is incompatible with the interface of the classes we have written in the new project so far.

We cannot change the interface of the existing class because we may not have its source code.

Even if we did have the source code, we probably would not prefer to re-program the class because it might be a challenging job, and testing would be necessary.

In the future, this class can be replaced by another class.

Motivation 2:

We have multiple classes that can provide the same or similar service.

For example storing data in different environments such as file system, local database system, or cloud database.

These classes have different (incompatible) interfaces.

Later, other classes with different interfaces may be added to the system.

How can these classes that have incompatible interfaces work together?

How should we connect our classes to these classes?

Definition of the Adapter Pattern:

Problem:
You want to use an existing class, and its interface does not match the one you need.

Solution:
Create an intermediate adapter object to convert the original interface of a class into another interface.

Example: Shapes Library [1]

Requirements:

- Wee need a library of shapes (points, lines, and squares) with common behavior such as display, fill, and undisplay.
- The client objects (library users) should not have to know the type of shape they are actually using.

Solution (*Design to interface principle*):

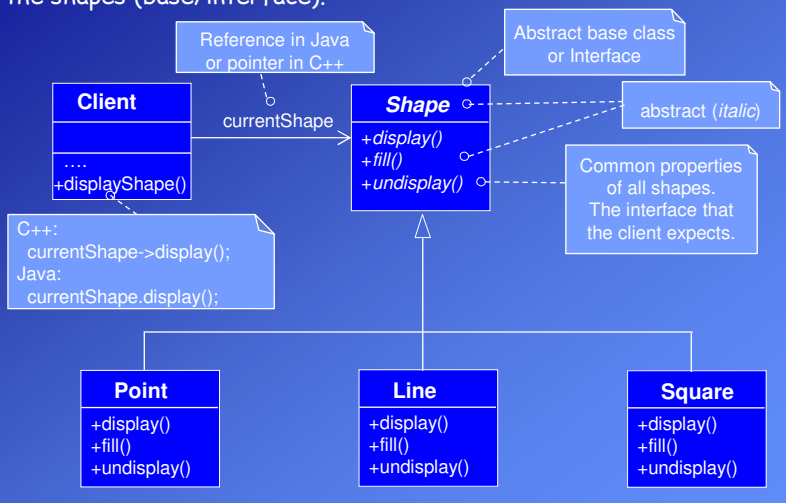
- We create an abstract base class (or interface in Java) Shape.
- From this base (interface), we will derive the concrete classes representing points, lines, and squares.
- With the help of polymorphism, we will have different objects (shapes) in the system, but client objects will interact with them in a common way.
- This allows the client objects to deal with all these objects similarly.
- It enables us to add different kinds of shapes in the future without having to change the clients.
- We have not used the adapter pattern yet!

[1] Alan Shalloway, James R. Trott , *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

Solution: Shapes Library

Design Principle: Program (design) to an interface, not an implementation.

We will program the client (user) objects according to common properties of the shapes (base/interface).



New Problem with the Shapes Library:

Requirements change!

We need to implement **a circle**, a new kind of Shape.

We can create a new class Circle that implements the shape "circle" and derive it from the Shape class.

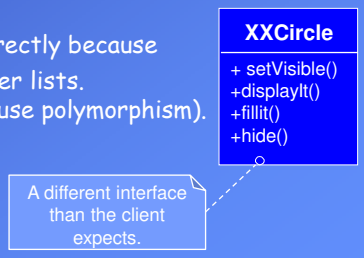
Thanks to polymorphism (*program to interface principle*), this new class will not affect the client.

However, we must write Circle's display, fill, and undisplay methods. That could be a complex and time-consuming task.

Fortunately, we already have (or can buy) an existing class XXCircle that deals with circles.

However, we cannot use this class XXCircle directly because

- It has different method names and parameter lists.
- We cannot derive it from Shape (we cannot use polymorphism).

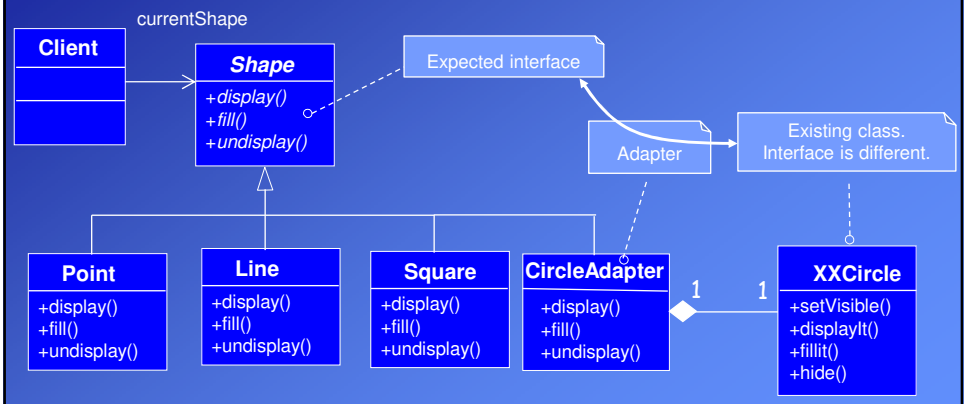


Solution with the **Adapter Pattern**:

We can create an adapter class Circle that is derived from *Shape*.

The adapter will include (wrap) the XXCircle object.

The client will call the methods of the adapter Circle that converts these calls to the interface of the XXCircle.



Coding in Java:

```
class CircleAdapter extends Shape  {    // Adapter
    ...
  private XXCircle realCircle;  // reference to the objects of the existing
                                // class

  public Circle(....)  {              // constructor
     realCircle = new XXCircle(....);  // object of the existing class is
  }                                    // created


    void public display()  {       // Adapting  (conversion) method
      realCircle.setVisible();     // Method calls are converted
      realCircle.displayIt();
   }
}
```


Coding in C++:

```
class CircleAdapter : public Shape  {       // Adapter
private:
  XXCircle  *realCircle;  // pointer  to the objects of the existing class
   ...                                      // other members
};

CircleAdapter::CircleAdapter(....){         // constructor
  realCircle = new XXCircle{....};  //object of the existing class is created
}

void CircleAdapter::display(){            // Adapting  (conversion) method
  realCircle->setVisible();               // Method calls are converted
  realCircle->displayIt();
}

CircleAdapter::~CircleAdapter()  {      // destructor
  delete realCircle;                    // contained object is destroyed
}
```

The Adapter Pattern (cont'd)

Creating a Common Stable Interface

The adapter pattern is also used to solve a more complicated problem.

Sometimes, to get the same service, a client object must work with multiple similar classes but with different interfaces.

Problem:

How to provide a **stable interface** to similar components with different interfaces?

Example: The NextGen POS system with external tax calculators

Suppose the NextGen POS system must support different third-party external tax calculator systems with different interfaces.

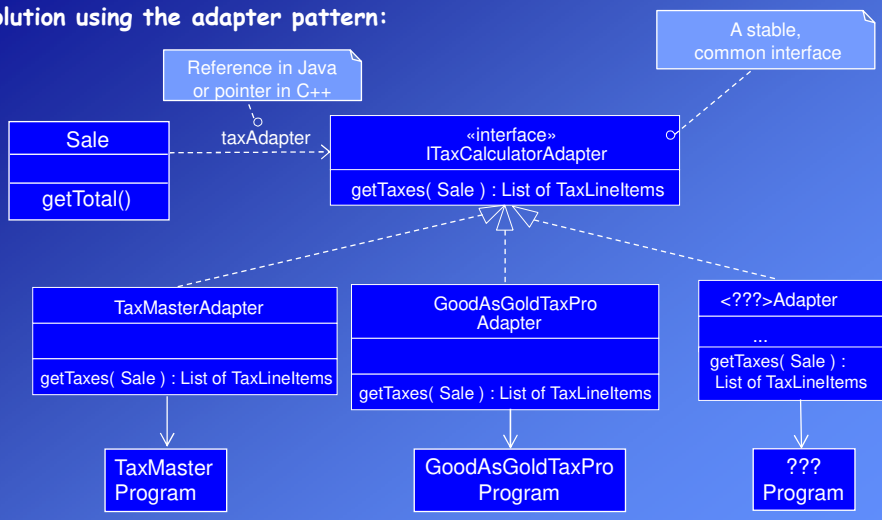
Depending on some conditions, the Sale class will sometime connect to the "Tax Master" program and sometimes to the "Good as Gold Tax Pro" program to calculate the tax.

In the future, other programs may also be added to the system.

©2012 - 2023 Feza BUZLUCA

Example (contd): External tax calculators with different interfaces

Solution using the adapter pattern:



The Sale class knows only the getTaxes(Sale) method.

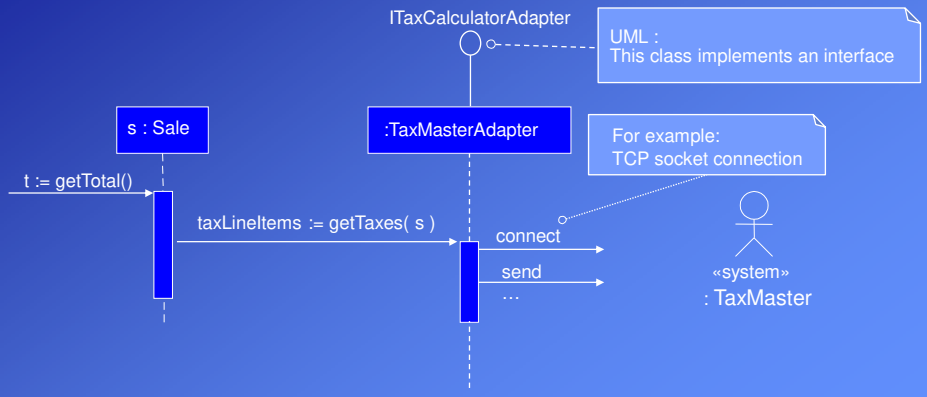
©2012 - 2023 Feza BUZLUCA

Example (contd): External tax calculators with different interfaces

The Sale class always sends the same getTaxes message to calculate the tax regardless of the external system used for calculation.

The current adaptor converts and directs this call to the tax calculator program.

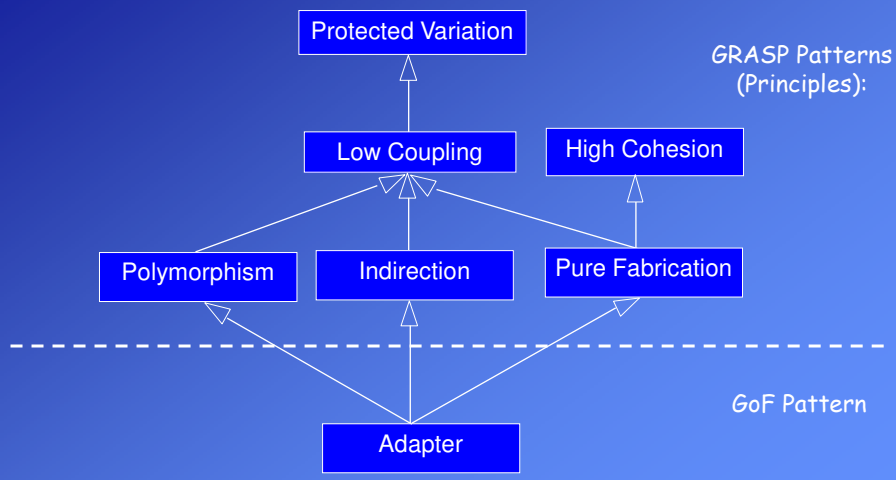
The sequence diagram below presents the case if the pointer taxAdapter in Sale points to the TaxMasterAdapter.



Relation between Design Principles and GoF Patterns:

A GoF pattern may include many design principles.

The relation between the Adapter Patterns and GRASP patterns (principles) is shown below:



The Abstract Factory Pattern (Creational)



Object (1) Object (2)

(1) http://www.kellyskindergarten.com
(2) http://www.pamsclipart.com

Before the GoF Abstract Factory pattern, we will see a simplification called Simple Factory, Concrete Factory, or just **Factory**.

Example: Creation of the adapters in the NextGen POS System

In the prior Adapter pattern solution for external tax calculators with varying interfaces (8.16), we face two new problems:

1. **Who creates the adapters?**
2. And how to determine **which type of adapter to create?**

Discussion:

If some domain object (for example, Sale) creates them (as the Creator pattern suggests), we will encounter the following problems.

- The domain objects (Sale) must know external systems (coupling).
- Adding or removing an external calculator will affect the Sale.
- Change in rules (or in conditions) about adapter usage (when, which adapter) will affect the Sale.
- This responsibility lowers the cohesion of the domain object because connectivity with external software components is not its primary job (separation of concerns).

The solution to the adapter-creation problem:

Warning: We will not use the Creator (GRASP) pattern here.

We will apply **the Factory** pattern, in which a Pure Fabrication object called "factory" is defined to create objects (in this example, the adapter objects).

Advantages of Factory objects:

- Separate the responsibility of complex creation into cohesive helper objects.
- Hide potentially complex creation logic.
- Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

Definition: The Factory Pattern

Problem:

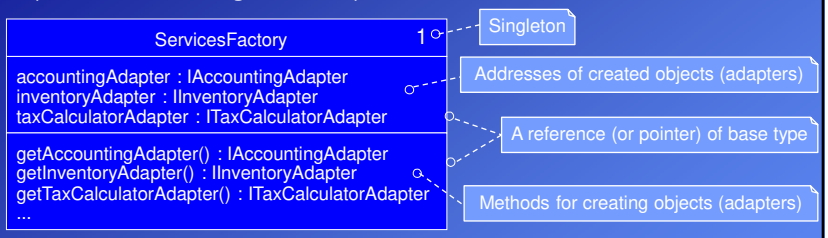
Who should be responsible for creating objects when there are special considerations, such as complex creation logic and a desire to separate the creation responsibilities for better cohesion?

Solution:

Create a Pure Fabrication object called a Factory that handles the creation.

Attention: Factory objects are defined not only to create adapters.
As the definition presents, this pattern can be applied to create different types of objects with a complex creation logic.

Example: ServicesFactory is the factory object that creates necessary adapters for the external systems (accounting, inventory, tax calculation) in the NextGen POS system.



When the domain object (Sale) needs to access an external tax calculator, it will call the getTaxCalculatorAdapter method of the ServicesFactory object.

This method (of the factory) will determine the appropriate adaptor according to the current conditions. It will also create (if necessary) the adapter and return its address to the domain object (Slide 8.23).

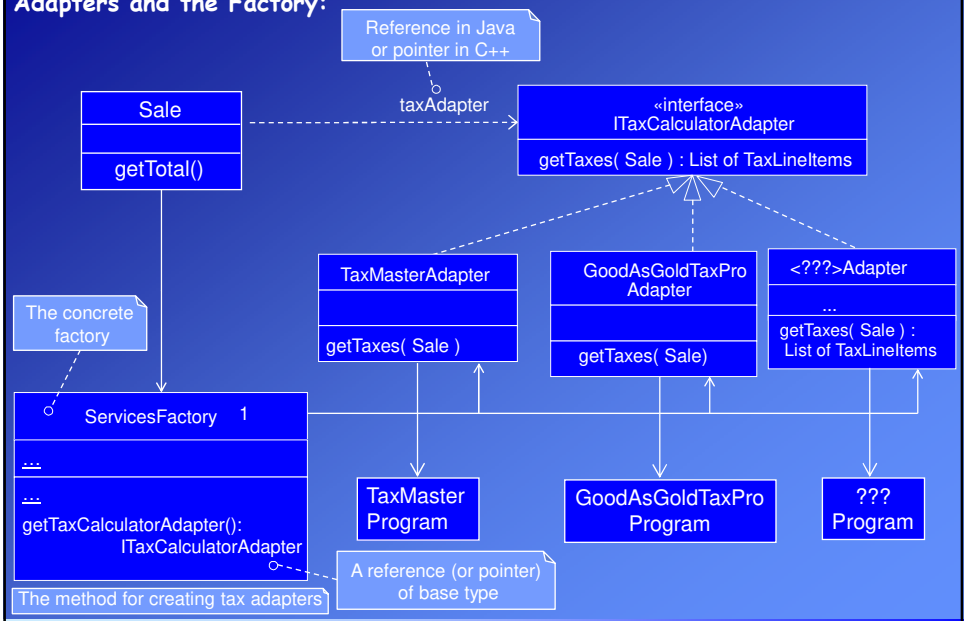
Advantages:

- The Sale object does not know from which external calculator it is being served.
- If a new adapter is added to the system or the creation logic changes, only the factory object is affected (we know where to look).

Factories are often accessed with the **Singleton pattern** that is explained later.

©2012 - 2023 Feza BUZLUCA

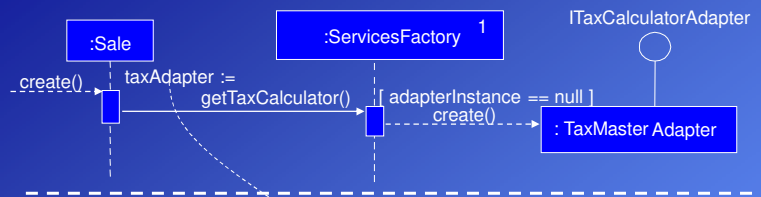
Adapters and the Factory:



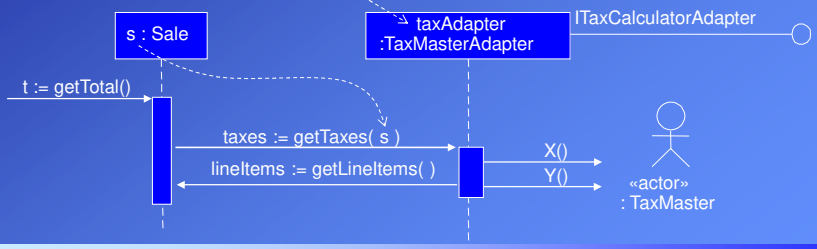
©2012 - 2023 Feza BUZLUCA

Creating and using Adapters:

Sale gets the address of the adapter object from the ServicesFactory.
In this example, the ServicesFactory decides to create the TaxMasterAdapter.



Using the TaskMasterAdapter pointed by the pointer (reference) taxAdapter:



Abstract Factory (GoF)

After the "Concrete Factory", we will discuss the more general Abstract Factory pattern.

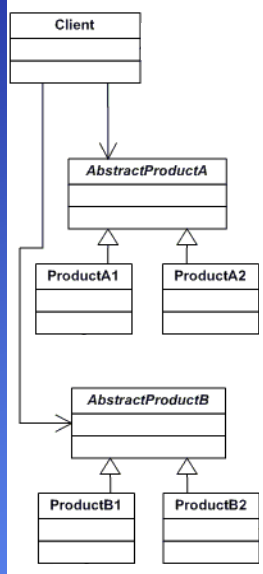
In some cases, objects that are related to each other must be created together.

For example, we have two groups of products, i.e., Group A and Group B.

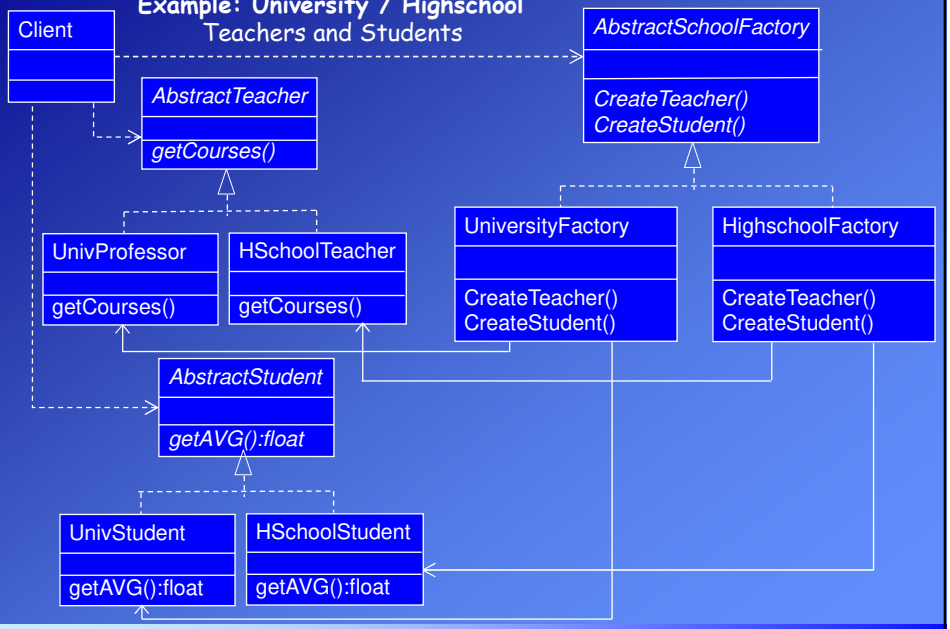
If ProductA1 is selected (created) from Group A, then ProductB1 must be created from Group B.

Similarly, objects ProductA2 and ProductB2 must be created together.

Now we will define two concrete factories derived from an abstract base (abstract factory).



Example: University / Highschool
Teachers and Students



The Singleton Pattern (Creational)

1

The Factory object ServicesFactory in the NextGen POS system raises another new problem in the design:

Who creates the factory itself, and how is it accessed?

Requirements:

- Only one factory instance (object) is needed within the process.
- The methods of this factory may need to be called from various places in the code.

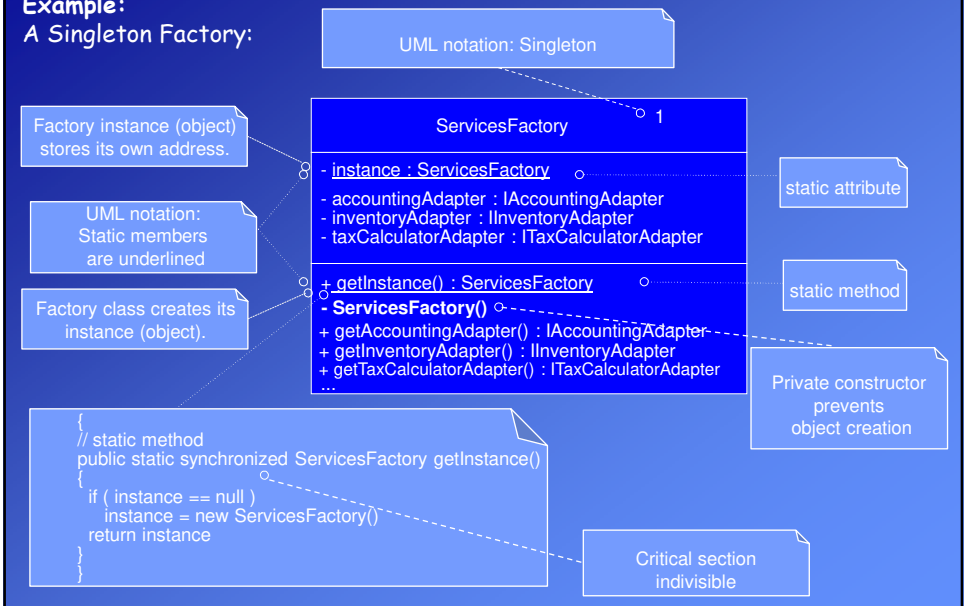
Definition of the Singleton pattern:

Problem: Exactly one instance of a class is allowed (singleton). Objects need a global and single point of access.

Solution: Define a static method of the class that creates and returns the address of the singleton.

Remember: static methods of a class can be called before an object of that class has been created.

Example:
A Singleton Factory:



Using the Factory:

When a domain object needs an adapter, at any point in the code, it can access the singleton factory object and get the address of an adapter.

Example: The Register gets the address of an accounting adapter (in Java)

```
public class Register
{
    public Register( ProductCatalog catalog )   // constructor
    {
      ...
      accountingAdapter = ServicesFactory.getInstance().getAcountingAdapter();
      ...
    }
    // Other methods
}
```

The factory class creates the factory object, if necessary, and returns the address of the factory object.

The factory object creates the adapter object, if necessary, and returns the address of this object.

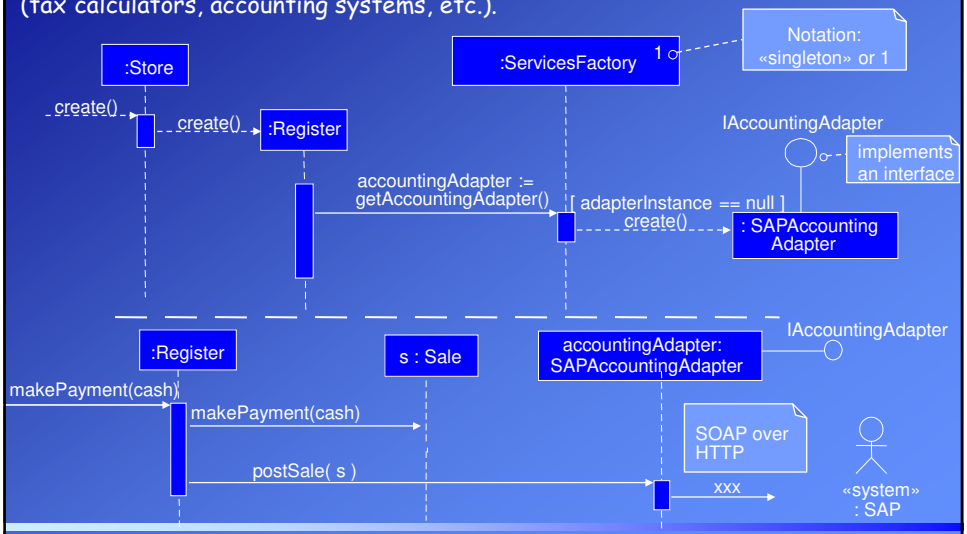
Example: Constructor of the Register in C++

```
Register( const ProductCatalog & catalog )   // constructor
{
    ...
    accountingAdapter = ServicesFactory::getInstance()->getAcountingAdapter();
    ...
}
```

Create the factory object, if necessary — Create the adapter

Summary: External Services with Varying Interfaces Problem

A combination of Adapter, Factory, and Singleton patterns have been used to provide Protected Variations from the varying interfaces of external systems (tax calculators, accounting systems, etc.).



©2012 - 2023 Feza BUZLUCA