Object Oriented Methods with UML

A · P · U
ASIA PACIFIC UNIVERSITY
OF TECHNOLOGY & INNOVATION

Introduction to Design Patterns- Lecture 8

- Topics(03/05/16)
  - Design Patterns

Design Pattern

- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software **design**.

- It is a description or template for how to solve a problem that can be used in many different situations.

What is Gang-of-Four DP?

- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

- These authors are collectively known as **Gang of Four (GOF)**. According to these authors design patterns are primarily based on the following principles of object orientated design.

  - Program to an interface not an implementation
  - Favor object composition over inheritance

Issues in Software Design

- Concurrency:
  - How to decompose the system into processes ,tasks and threads.
- Controlling and Handling of Events
  - How to organize the flow of data to control temporal events
- Distribution
  - How the objects communicate with each other.
- Interactive systems
  - Which approach is used to interact with the users.
- Persistence
  - How to handle the persistent data.

Types of Design Patterns

S. N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new opreator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.
4	**J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

Creational patterns

- Abstract Factory
- Builder
- Factory Method
- Object Pool
- Prototype
- Singleton

Structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Private Class Data
- Proxy

Behavioral patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Creational Pattern-Singleton

- Problem:

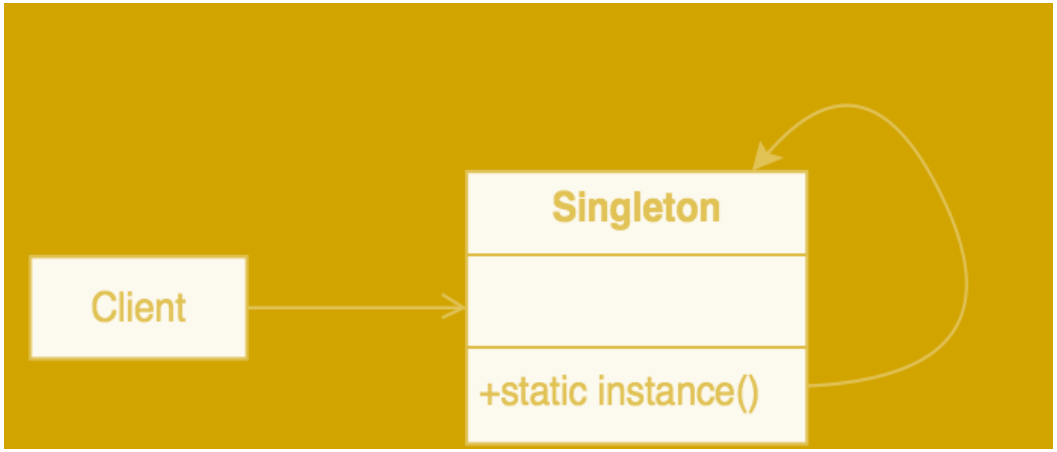
Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion

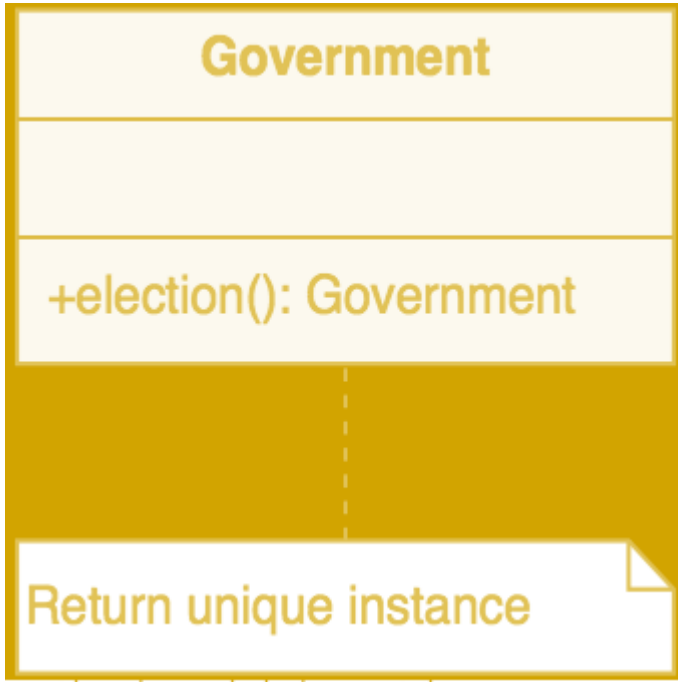
- Make the class of the single instance object responsible for creation, initialization, access, and enforcement.
- Declare the instance as a private static data member.
- Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

Singleton pattern

- Solution :Singleton pattern



Singleton pattern



Check list-Singleton pattern

- Define a private static attribute in the "single instance" class.
- Define a public static accessor function in the class.
- Do "lazy initialization" (creation on first use) in the accessor function.
- Define all constructors to be protected or private.
- Clients may only use the accessor function to manipulate the Singleton.

Sample Code-Singleton

- Implementation -Singleton

Structural pattern-Adapter

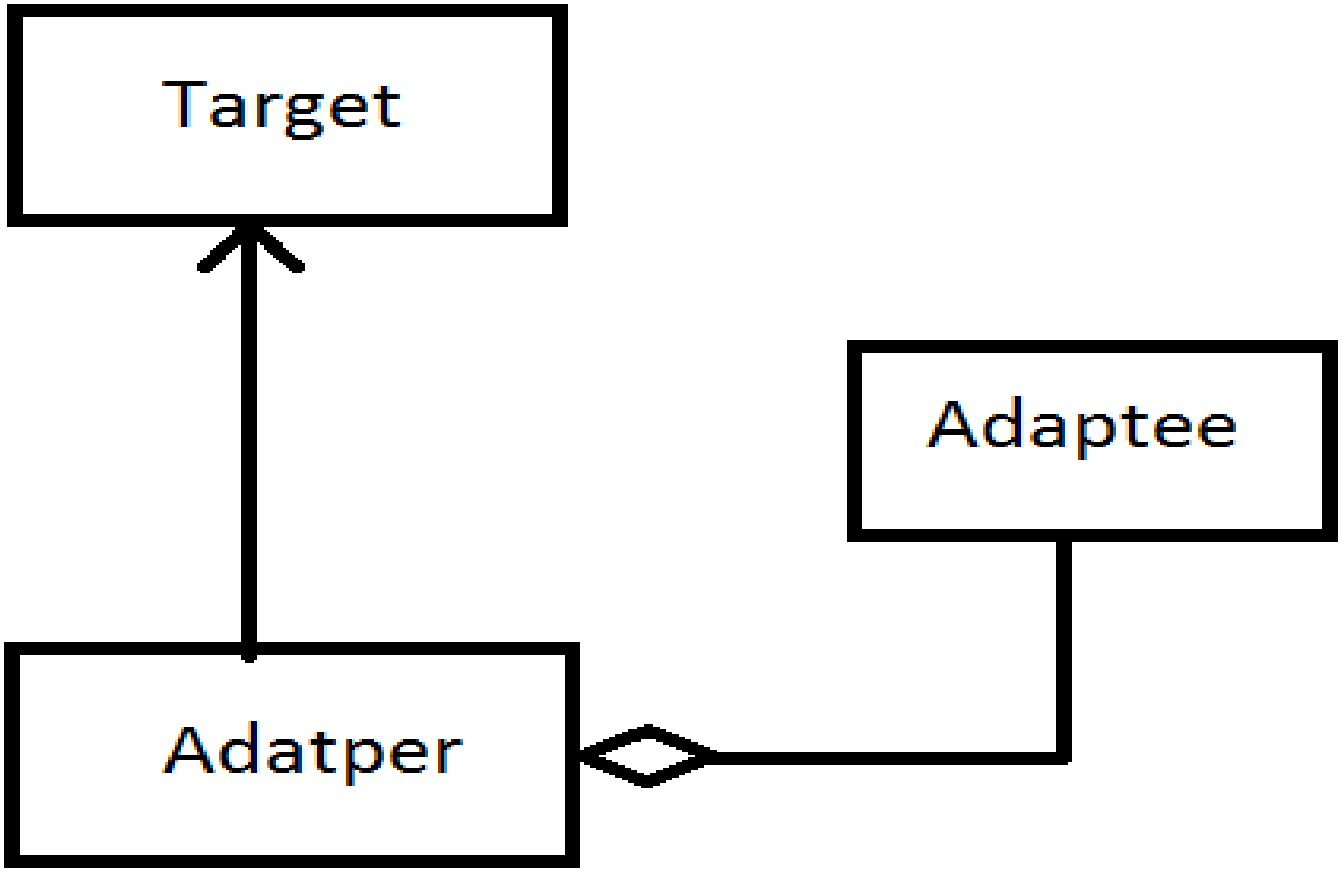
- **Problem**

An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Discussion

- Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system.
- Clients call methods on the Adapter object which redirects them into calls to the legacy component.
- This strategy can be implemented either with inheritance or with aggregation.

Discussion



Discussion

The Adapter could also be thought of as a "wrapper".

Class description:

AbstractPlug : Abstract Target class

Plug : Concrete Target class

AbstractSwitchBoard : Abstract Adaptee class

SwitchBoard : Concrete Adaptee class

Adapter : Adapter class

Check list-Adapter Pattern

- Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
- Identify the interface that the client requires.
- Design a "wrapper" class that can "impedance match" the adaptee to the client.
- The adapter/wrapper class "has a" instance of the adaptee class.
- The adapter/wrapper class "maps" the client interface to the adaptee interface.
- The client uses (is coupled to) the new interface

AbstractPlug Code

```
class AbstractPlug
{ public:
        void virtual RoundPin(){    }
        void virtual PinCount(){    }
 };
```

// Concrete Target

```
class Plug : public AbstractPlug
{ public: void RoundPin()
{ cout << " I am Round Pin" << endl;
 }
 void PinCount()
{ cout << " I have two pins" << endl;
 }
 };
```

// Abstract Adaptee

- *// Abstract Adaptee* class AbstractSwitchBoard

```
{ public:
void virtual FlatPin() {}
void virtual PinCount() {}
}
```

// Concrete Adaptee

```
class SwitchBoard : public AbstractSwitchBoard
{
public: void FlatPin()
{ cout << " Flat Pin" << endl; }
void PinCount()
{ cout << " I have three pins" << endl; }
 };
```

// *Adapter* class

```
Adapter : public AbstractPlug
 {
public: AbstractSwitchBoard *T;
Adapter(AbstractSwitchBoard *TT)
 { T = TT; }
void RoundPin()
 { T->FlatPin(); }
void PinCount()
{ T->PinCount(); }
};
```

// Client code


```
void _tmain(int argc, _TCHAR* argv[])
{
SwitchBoard *mySwitchBoard = new SwitchBoard;
AbstractPlug *adapter = new Adapter(mySwitchBoard);
 adapter->RoundPin();
 adapter->PinCount();
}
```

References

- https://sourcemaking.com/design_patterns/