Software Engineering II

Introduction to Design Patterns

Dr. Amr S. Ghoneim

Lecture Objectives:

- Introduce the concept of design patterns.
- Discuss why design patterns are important and what advantages they provide.
- Discuss the following patterns :
  - Singleton
  - Immutable
  - Delegator

Introduction to Patterns

- A *pattern* is the outline of a reusable solution to a recurring problem encountered in a particular context.
- Many of them have been systematically documented for all software developers to use.
- A good pattern should
  - Contain a solution that has been proven to effectively solve the problem in the indicated context.
- *Studying patterns is an effective way to learn from the experience of others*

Motivation for Patterns

- Patterns provide common language between developers. For example if a developer tells another developer that he is using a *Singleton*, the other developer should know exactly what this means.

- They help to improve software quality & reduce development time. As They provide proven solution and are based on the basic principles and heuristics of object orientated design like :
  - — As we will see later
  - Program to an interface not to an implementation
  - Favor object composition over inheritance
  - Encapsulates what varies
  - Law of Demeter
  - Reduce coupling

Software Patterns

Software Patterns include the following types:

- Analysis Patterns
- Architectural Patterns
- Assigning responsibilities patterns
- Design Patterns

What's Design Patterns

- Design pattern is a solution to a **recurring design problem** within a particular context

Design *Pattern = Design Problem & Solution pair in a context*

Design Patterns descriptions are often independent of programming language and implementation details.

Design Patterns & Class libraries

- Class Libraries :
  - Reusable code; ex. String class, Math class,…
  - Language dependent.

- Design Patterns :
  - Reusable Design; Design problem & solution
  - Language independent.

Pattern Description

- Four main elements to describe any pattern:
  - The name of the pattern
  - The purpose of the pattern: what problem it solves
  - How to solve the problem
  - The constraints we have to consider in our solution

Pattern Description (Template 1)

Name:

Context:

The general situation in which the pattern applies

Problem:

A short sentence or two raising the main difficulty.

Forces:

- The issues or concerns to consider when solving the problem

Solution:

- The recommended way to solve the problem in the given context.

—'to balance the forces'

Antipatterns: (Optional)

- Solutions that are inferior or do not work in this context.

Related patterns: (Optional)

- Patterns that are similar to this pattern.

References:

- Who developed or inspired the pattern.

Pattern description (GOF Template)

Name:

Intent:
The general situation in which the pattern applies

Problem:
A short sentence or two raising the main difficulty.

Solution:
The approach to solve the problem .

Structure:
Class diagram

Participants:
Entities involved in the pattern.

Consequences:
Effect the pattern has on your system

Implementation:
Example ways to implement the pattern.

Patterns categorization

Patterns are classified into three categories

- *Creational Patterns*
  - — *Concerned with creation of objects*
- *Structural Patterns*
  - — *Concerned with composition of classes or objects into larger structures*
- *Behavioral Patterns*
  - — *Determines the ways in which the classes interact and distribute responsibilities (determines the flow of control in a complex program)*

Examples - GOF Patterns

Creational

- Abstract Factory
- Factory Method
- Builder
- Prototype
- Singleton

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioural

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

Some Creational Patterns:

- *Singleton*
  - — *Ensures that one instance of a class will be created*

- *Immutable Pattern*
  - — *Ensures that the state of the object never changes after creation*

The Singleton Pattern

- ***Context***:
  - — It is very common to find classes for which only one instance should exist (*singleton)* .
  - — Examples: Company or university class, Main Window class in GUI.

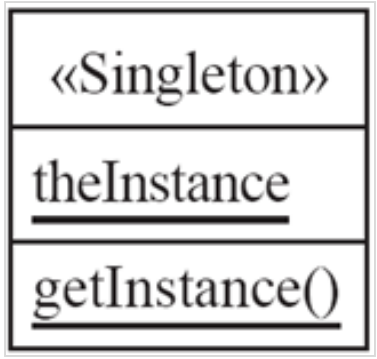
- ***Problem***:
  - — How do you ensure that it is never possible to create more than one instance of a singleton class. And provide a global point of access to it.

- ***Forces***:
  - — The use of a public constructor cannot guarantee that no more than one instance will be created.
  - — The singleton instance must also be accessible to all classes that require it, therefore it must often be public.

The Singleton Pattern

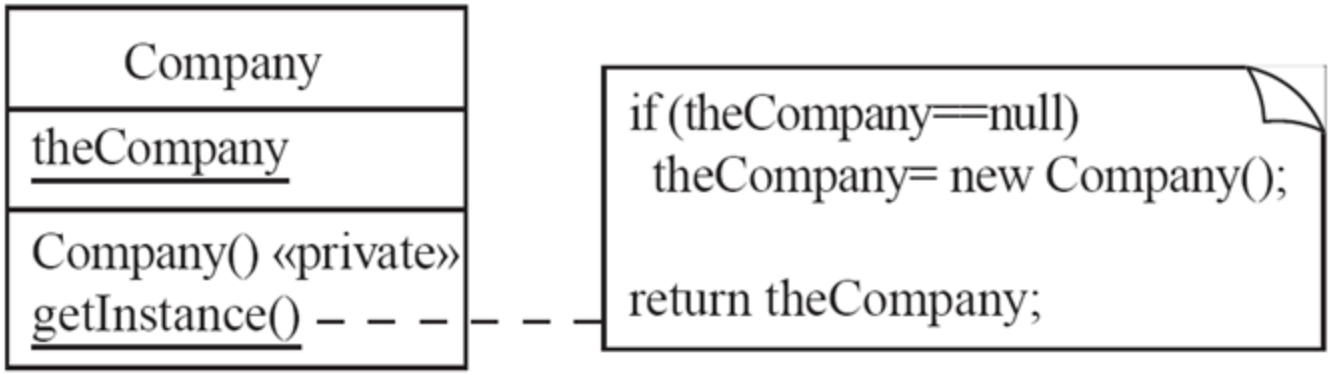
- ***Solution:***

- *Have the constructor private to ensure that no other class will be able to create an instance of the class singleton.*
- *Define a public static method, The first time this method is called , it creates the single instance of the class "singleton" and stores a reference to that object in a static private variable .*



The Singleton Pattern

- ***Example:***



Immutable Pattern

- ***Context***:
  - — An immutable object is an object that has a state that never changes after creation

- ***Problem***:
  - — How do you create a class whose instances are immutable?

- ***Forces***:
  - — There must be no loopholes that would allow ‘illegal’ modification of an immutable object

Immutable Pattern

➢ ***Solution***:

— Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.

— Instance methods which access properties must not change instance variables.

Some Structural Patterns

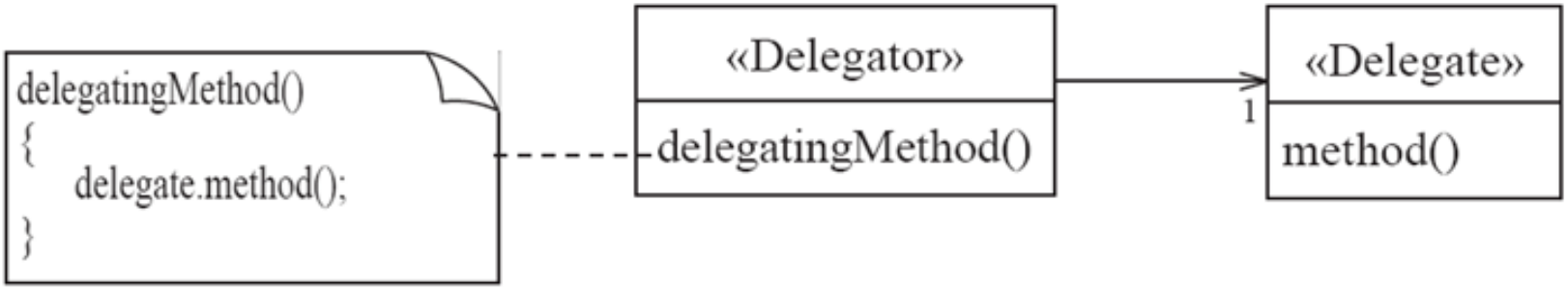
- *Delegation pattern*
  - *— For reusing methods.*
- *Adapter Pattern*
  - *— Convert programming interface of one class into another (reusing an existing unrelated class)*
- *Façade Pattern*
  - *— Defines a higher level interface that makes subsystems easier to use*
- *Read-only interface Pattern*
  - *— To give privileges to some classes to be able to modify attributes of objects that are otherwise immutable.*
- *Abstraction-Occurrence Pattern*

The Delegation Pattern

- ***Context***:
  - — You are designing a method in a class.
  - — You realize that another class has a method which provides the required service.
  - — Inheritance is not appropriate:
    - E.g. because the is-a rule does not apply
- ***Problem***:
  - — How can you most effectively make use of a method that already exists in the other class?
- ***Forces***:
  - — You want to minimize development cost by reusing methods

The Delegation Pattern

- ***Solution:***



- *The delegating method in the delegator class calls a method in the delegate class to perform the required task. An association must exist between the delegator and delegate classes.*

The Delegation Anti-Patterns

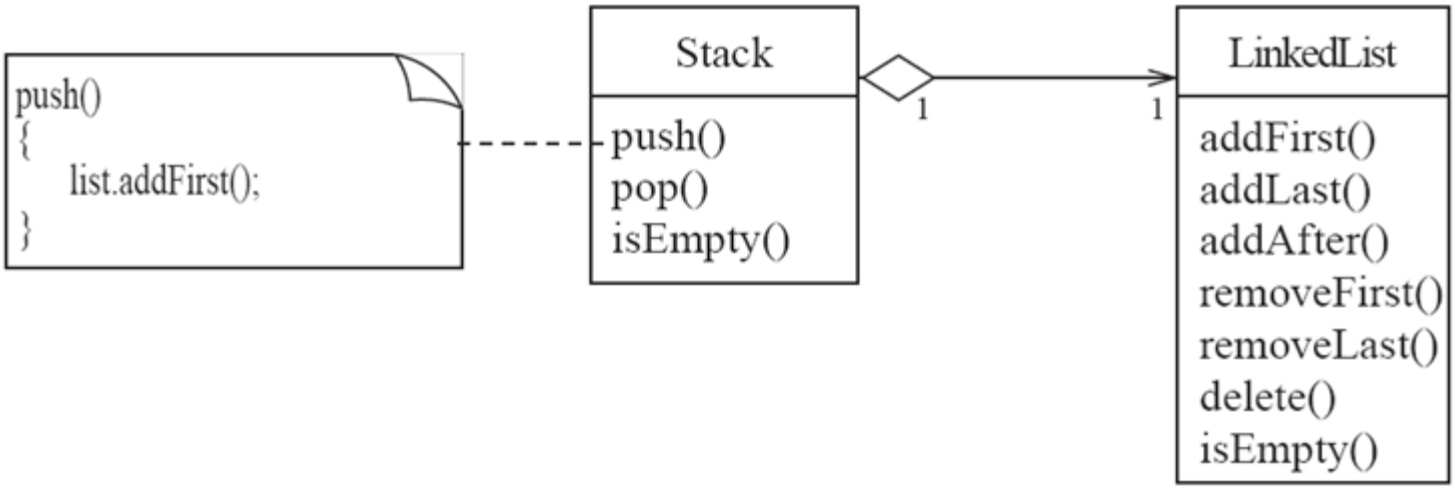
- It's common for people to overuse generalization and inherit the method that's to be reused

  —Ex: What is the problem with making stack a subclass of linked list?

- Duplication of chunks of code.

The Delegation Pattern

- ***Example:*** *The stack class could be created using an existing class in the java collection framework called linkedList using delegation pattern. The push method of stack calls the addfirst method of linkedList,etc.*



The Delegation Pattern

- The delegation pattern brings together **three** design principles that encourage flexible design:
  - *Favoring association over inheritance* when the full power of inheritance is not needed.
  - *Avoiding duplication of chunks of code*.

- Accessing nearby information only; this principle is called "*The Law of Demeter*" .
  - It is about avoiding the chain of messages *(talk to your immediate friends, don't talk to friends of friends)*.
  - It creates loosely coupled systems.