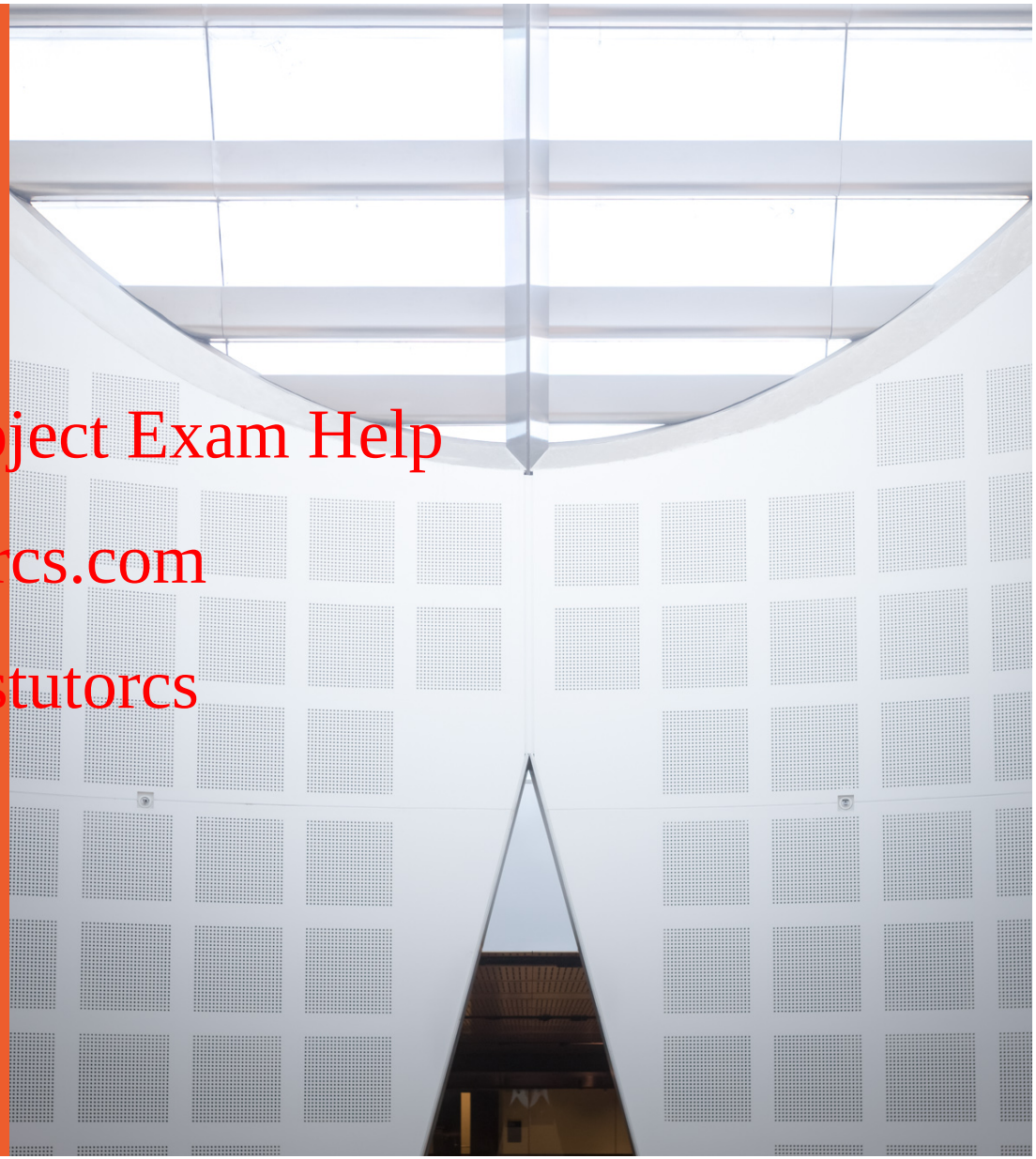
Software Design and Construction 1
SOFT2201 / COMP9201

Software Design Principles

Dr. Xi Wu

School of Computer Science

THE UNIVERSITY OF SYDNEY



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

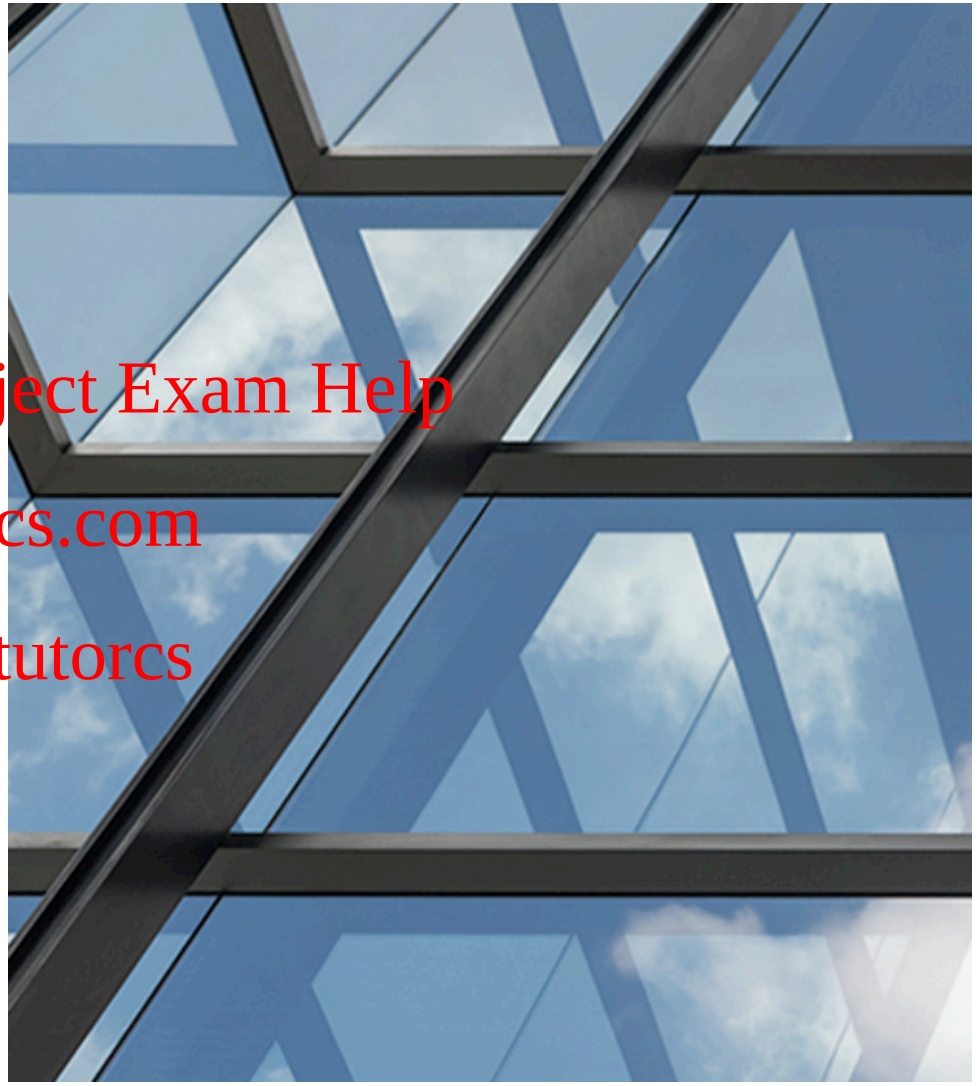
WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

How do we know if the design of a software is good?



THE UNIVERSITY OF SYDNEY

Agenda

- Design Smells
- SOLID
  - A list of excellent design principles
- GRASP
  - Designing objects with Responsibilities

Design Smells

Design Smells

"*Structures in the design that indicate violation of fundamental design principles and negatively impact design quality*" – Girish Suryanarayana et. Al. 2014

- Poor design decision that make the design fragile and difficult to maintain

- Bugs and unimplemented features are not accounted

http://www.codeops.tech/blog/linkedin/what-causes-design-smells/

Girish Suryanarayana, et. al. (2014). "Refactoring for software design smells: Managing technical debt"

Common Design Smells

- **Missing abstraction:** bunches of data or encoded strings are used instead of creating an abstraction
- **Multifaceted abstraction:** an abstraction has multiple responsibilities assigned to it
- **Insufficient modularization:** an abstraction has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both
- **Cyclically-dependent modularization:** two or more abstractions depend on each other directly or indirectly (creating tightly-coupling abstractions)
- **Cyclic hierarchy:** a super-type in a hierarchy depends on any of its subtypes

Girish Suryanarayana, et. al. (2014). "Refactoring for software design smells: Managing technical debt"

Summary of Common Symptoms Design Smells

- Rigidity (difficult to change)
- Fragility (easy to break)
- Immobility (difficult to reuse)
- Viscosity (difficult to do the right thing)
- Needless Complexity (overdesign)
- Needless Repetition (mouse abuse)
- Opacity (disorganized expression)

Girish Suryanarayana, et. al. (2014). "Refactoring for software design smells: Managing technical debt"

Symptoms Design Smells

- **Rigidity (difficult to change):** the system is hard to change because every change forces many other changes to other parts of the system
  - A design is rigid if a single change causes a cascade of subsequent changes in dependent modules
- **Fragility (easy to break):** Changes cause the system to break in places that have no conceptual relationship to the part that was changed
  - Fixing those problems leads to even more problems
  - As fragility of a module increases, the likelihood that a change will introduce unexpected problems approaches certainty
- **Immobility (difficult to reuse):** It is hard to detangle the system into components that can be reused in other systems

Symptoms Design Smells

- **Viscosity:** Doing things right is harder than doing things wrong
  - *Software:* when design-preserving methods are more difficult to use than the others (hacks), the design viscosity is high (easy to do the wrong thing but difficult to do the right thing)
  - *Environment:* when development environment is slow and inefficient.
    - *Compile times are very long, developers try to make changes that do not force large recompiles, even such changes do not preserve the design*
- **Needless Complexity:** when design contains elements that are not useful.
  - When developers anticipate changes to the requirements and put facilities in software to deal with those potential changes.

Symptoms Design Smells

- **Needless Repetition:**
  - Developers tend to find what they think relevant code, copy and paste and change it in their module
  - Code appears over and over again in slightly different forms, developers are missing an abstraction
  - Bugs found in repeating modules have to be fixed in every repetition

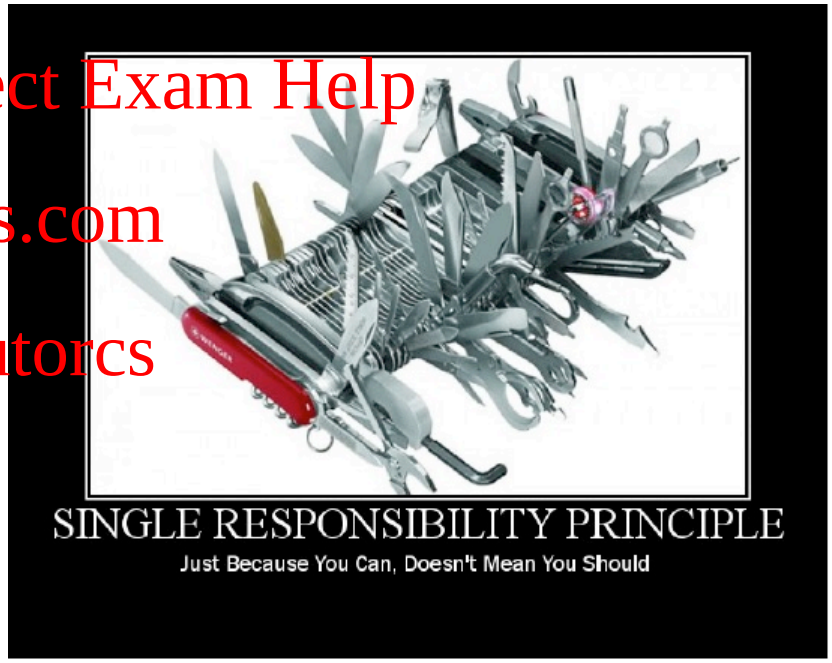
- **Opacity:** tendency of a module to be difficult to understand
  - Code written in unclear and non-expressive way
  - Code that evolves over time tends to become more and more opaque with age
  - Developers need to put themselves in the reader's shoes and make appropriate effort to refactor their code so that their readers can understand it

SOLID Design Principles

THE UNIVERSITY OF SYDNEY

SOLID: Single Responsibility

Every class should have a single responsibility and that responsibility should be entirely met by that class



SOLID: Open/Closed

Open for extension but not for mutilation

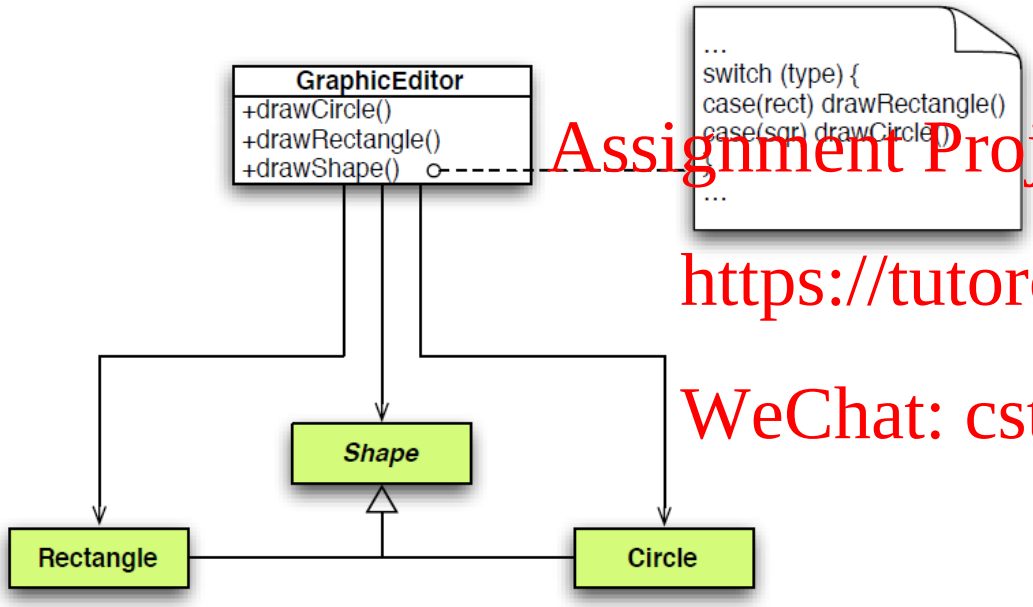
Have you ever written code that you don't want others to mess with?

Have you ever wanted to extend something that you can't?

The Open/Closed principle is that you should be able to extend code without breaking it. That means not altering superclasses when you can do as well by adding a subclass.

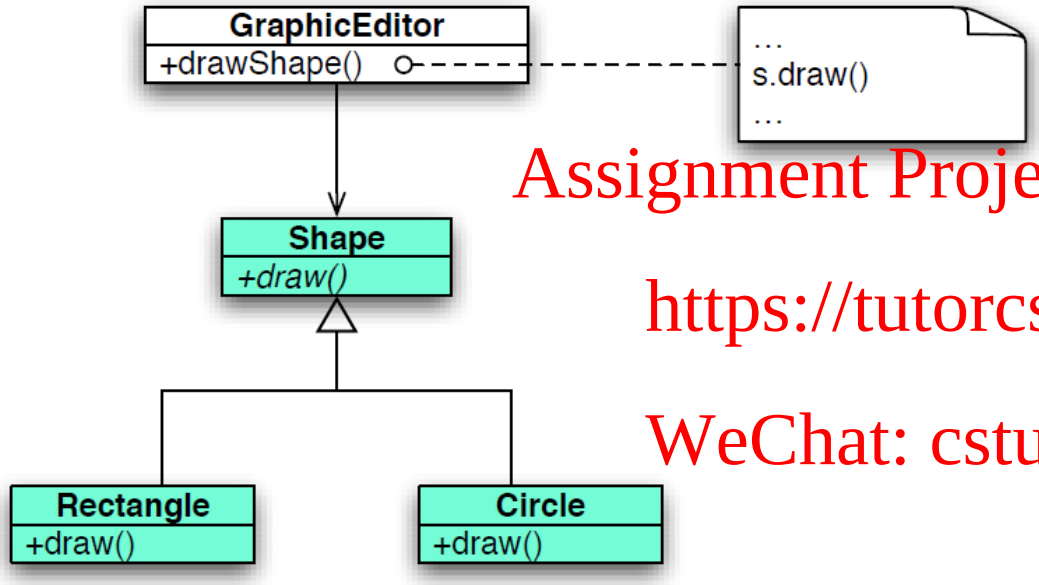
New subtypes of a class should not require changes to the superclass

SOLID: Open/Closed



This really isn't a good design: every time a new object is introduced to be drawn, the base class has to be changed

SOLID: Open/Closed



This is much better: each item has its own draw method which is called at runtime through polymorphism mechanisms

SOLID: Liskov Substitution Principle

- In 1987 Barbara Liskov introduced her idea of strong behavioral subtyping, later formalized in a 1994 paper with Jeannette Wing and updated in a 1999 technical report as

> *Let q(x) be a property provable about objects x of type T . Then q(y) should be provable for objects y of type S where S is a subtype of T .*

- This defines required behaviors for mutable (changeable) objects: if S is a subtype of T , then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

- What's "desirable"? One example is correctness. . .

Substitutability

- Suppose we have something like this in our code: after a definition of someRoutine() we have

ALGORITHM
// presomeRoutine must be true here
someRoutine()
// postsomeRoutine must be true here

- But now we want to replace *someRoutine* with *someNewRoutine* with a guarantee of no ill-effects, so now we need the following: someNewRoutine();

// presomeRoutine must be true here
// presomeNewRoutine must be true here
someNewRoutine()
// postsomeNewRoutine must be true here
// postsomeRoutine must be true here

This means, for a routine r substituted by s

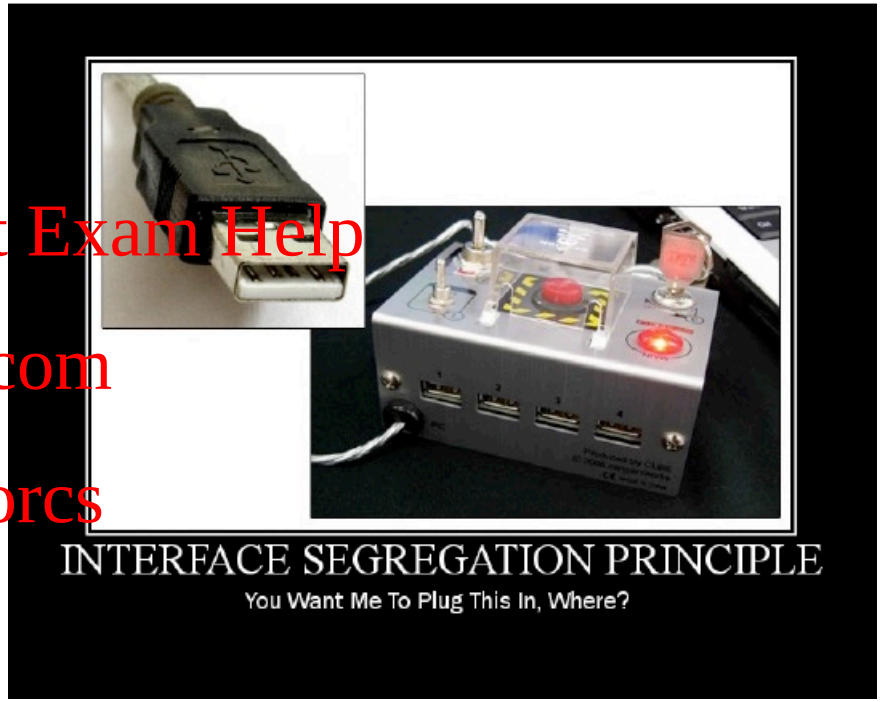
$pre_r \Rightarrow pre_s$ and $post_s \Rightarrow post_r$

Substitutability

- In other words;
  - Pre-conditions cannot get stronger, Post-conditions cannot get weaker
- The aim is not to see if a replacement can do something the original couldn't but can the replacement do everything the original could under the same circumstances.
- Substitutability is asking the question 'Can one substitute one type for another with a guarantee of no ill-effects? We might need to consider substitutability in cases:
  - Refactoring
  - Redesign
  - Porting
- The context is 'changing something in existing use'

SOLID: Interface Segregation

You should not be forced to implement interfaces you don't use!



SOLID: Dependency Inversion

No complex class should depend on simpler classes it uses; they should be separated by interfaces (abstract classes)

The details of classes should depend on the abstraction (interface), not the other way around

Summary of SOLID Principles

Single Responsibility: Every class should have a single responsibility and that responsibility should be entirely met by that class;

Open/Closed: Open for extension but closed for modification; inheritance is used for this, e.g. through the use of inherited abstract base classes;

Liskov Substitutability: If $S <: T$ ("S is a subtype of T") then a T object can be replaced with an S object and no harm done;

Interface Segregation: Client code should not have to implement interfaces it doesn't need;

Dependency Inversion: High level, complex modules should not depend on low-level, simple models — use abstraction, and implementation details should depend on abstractions, not the other way around.

General Responsibility Assignment Software Pattern (GRASP)

Designing objects with responsibilities

THE UNIVERSITY OF SYDNEY

Object Design

- "Identify requirements, create a domain model, add methods to the software classes, define messages to meet requirements…"

- Too Simple!
  - What methods belong where?
  - How do we assign ***responsibilities*** to classes?

- The critical design tool for software development is a mind well educated in design ***principles*** and **patterns**.

Responsibility Driven Design

- Responsibility is a contract or obligation of a class

- What must a class "know"? [knowing responsibility]
  - Private encapsulated data
  - Related objects
  - Things it can derive or calculate

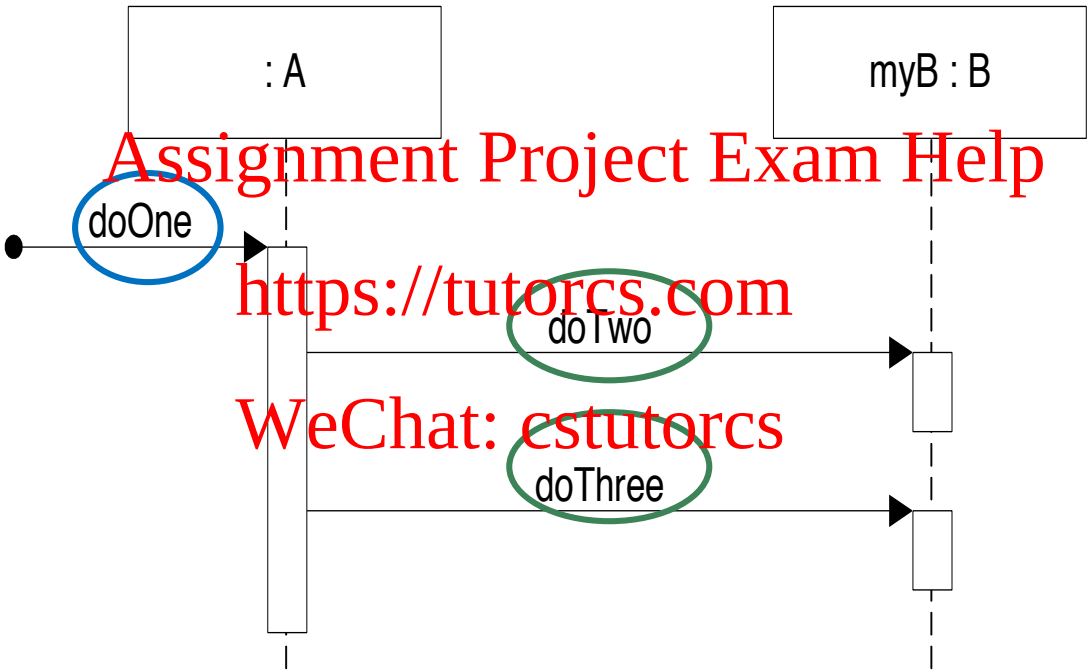
- What must a class "do"? [doing responsibility]
  - Take action (create an object, do a calculation)
  - Initiate action in other objects
  - Control/coordinate actions in other objects

- Responsibilities are assigned to classes of objects during object design

Responsibilities: Examples

- "A *Sale* is responsible for creating *SalesLineItems*" (doing)
- "A *Sale* is responsible for knowing its total" (knowing)
- Knowing responsibilities are related to attributes, associations in the domain model
- Doing responsibilities are implemented by means of methods.

Doing Responsibilities: Example



GRASP: Methodological Approach to OO Design

General **R**esponsibility **A**ssignment **S**oftware **P**atterns

The five basic principles:

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller

GRASP: Creator Principle

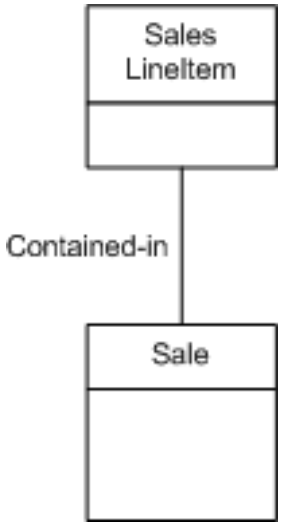
Problem

Who creates an A object

Solution

Assign class B the responsibility to create an instance of class A if one of these is true

- B "contains" A
- B "records" A
- B "closely uses" A
- B "has the Initializing data for" A



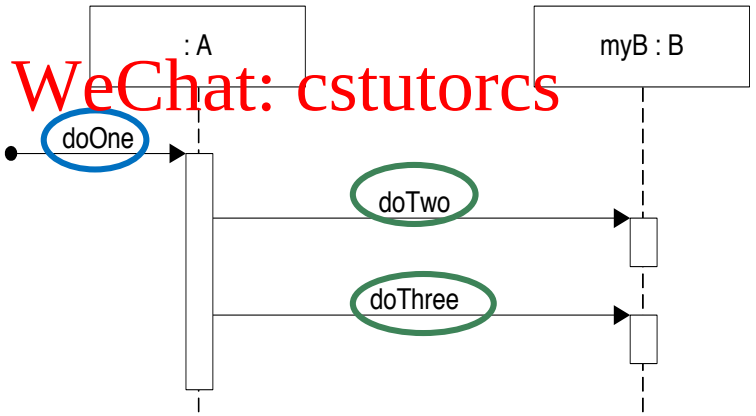
GRASP: Information Expert Principle

Problem

What is a general principle of assigning responsibilities to objects

Solution

Assign a responsibility to the class that has the information needed to fulfill it



Dependency

- A dependency exists between two elements if changes to the definition of one element (the **supplier**) may cause changes to the other (the **client**)

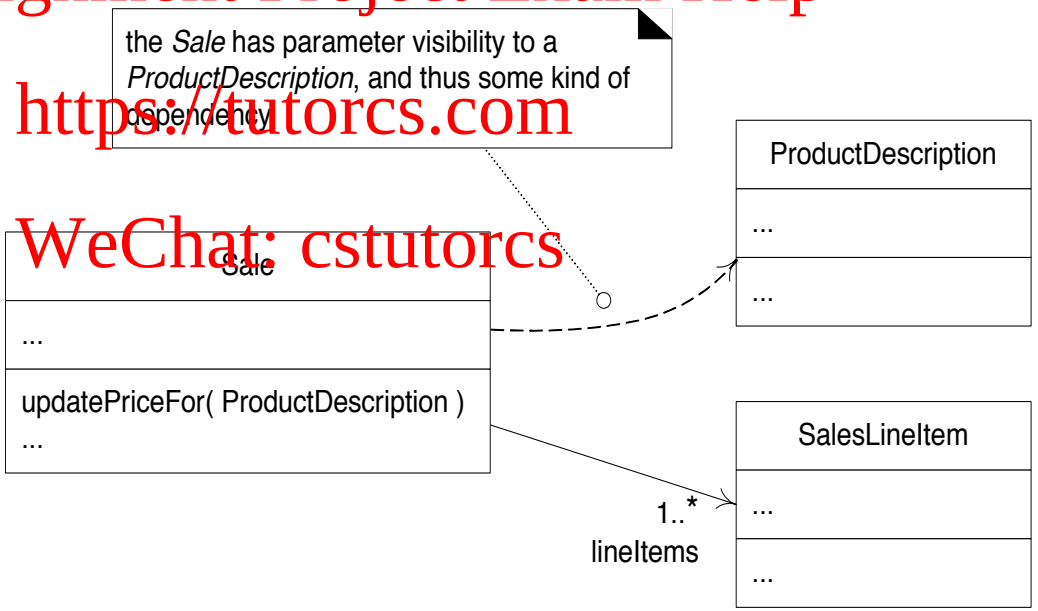
- Various reason for dependency
  - Class send message to another
  - One class has another as its data
  - One class mention another as a parameter to an operation
  - One class is a superclass or interface of another

When to show dependency?

- Be selective in describing dependency
- Many dependencies are already shown in other format
- Use dependency to depict global, parameter variable, local variable and static-method.
- Use dependencies when you want to show how changes in one element might alter other elements

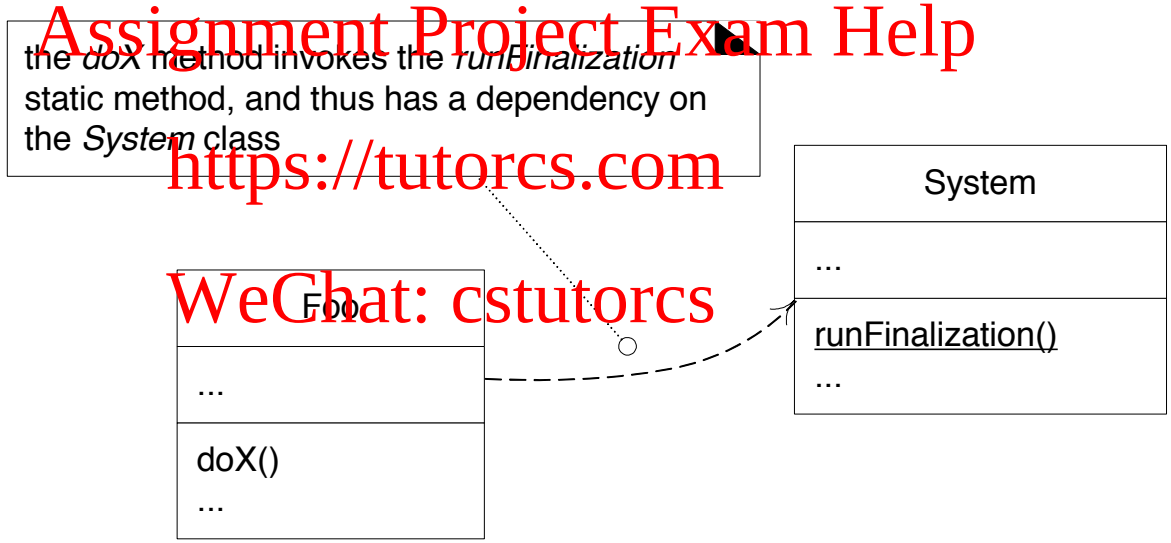
Dependency: Parameter Variable

```
public class Sale{
  public void updatePriceFor (ProductDescription description){
        Money basePrice = description.getPrice();
        //…
  }
}
```

Dependency: static method

```
public class Foo{
      public void doX(){
        System.runFinalization();
        //..
      }
}
```



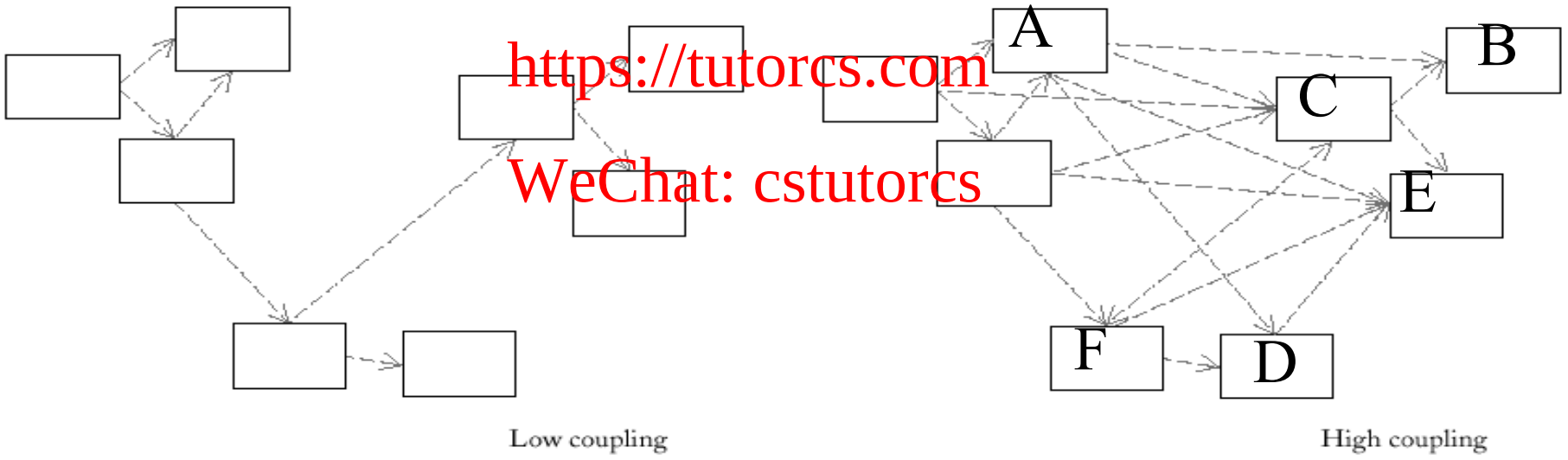
Dependency labels

- There are many varieties of dependency, use keywords to differentiate them

- Different tools have different sets of supported dependency keywords.
  - <<call>> the source calls an operation in the target
  - <<use>> the source requires the targets for its implementation
  - <<parameter>> the target is passed to the source as parameter.

Coupling

- How strongly one element is connected to, has knowledge of, or depends on other elements
- Illustrated as dependency relationship in UML class diagram



GRASP: Low Coupling Principle

Problem

How to reduce the impact of change, to support low dependency, and increase reuse?

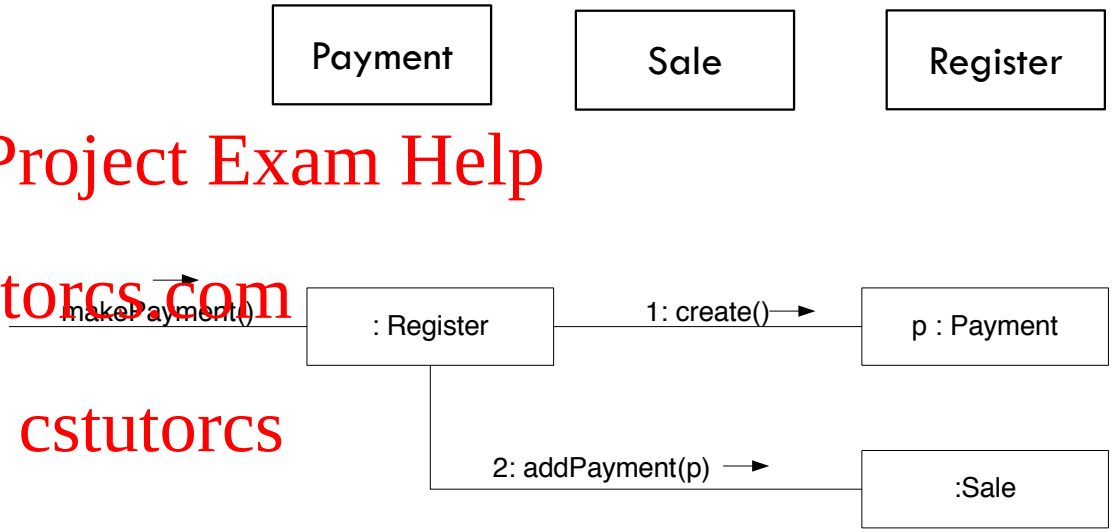
Solution

Assign a responsibility so that coupling remains low

Coupling – Example (NextGen POS)

We need to create a *Payment* instance and associate it with the *Sale*.
What class should be responsible for this?

Since *Register* record a payment in the real-world domain, the Creator pattern suggests register as a candidate for creating the payment

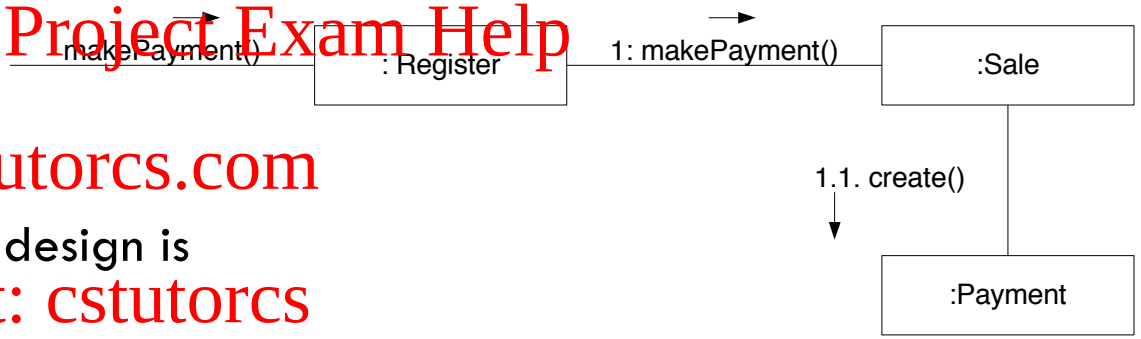


This assignment couple the *Register* class to knowledge of the *Payment* class which increases coupling

Coupling – Example (NextGen POS)

Better design from coupling point of view

It maintains overall lower coupling



From creator point of view, the previous design is better.

In practice, consider the level of couple along with other principles such as Expert and High Cohesion

Cohesion

- How strongly related and focused the responsibilities of an element are

- Formal definition (calculation) of cohesion
  - Cohesion of two methods is defined as the intersection of the sets of instance variables that are used by the methods
  - If an object has different methods performing different operations on the same set of instance variables, the class is cohesive

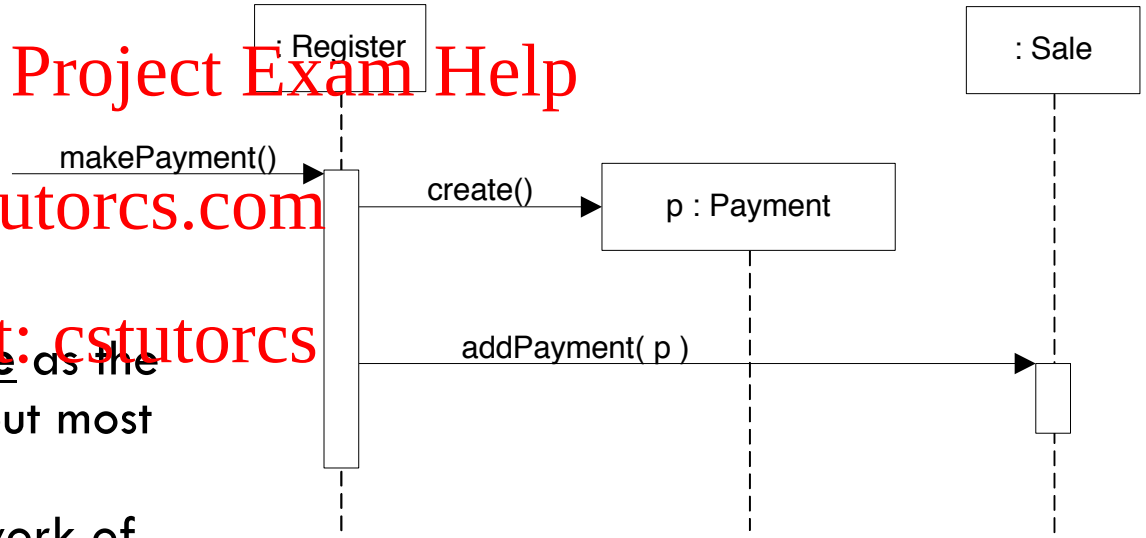
High Cohesion

- Problem
  - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- Solution
  - Assign responsibilities so that cohesion remains high

Cohesion – Example (NextGen POS)

We need to create a (cash) *Payment* instance and associate it with the *Sale*.
What class should be responsible for this?

Since *Register* record a payment in the real-world domain, the Creator pattern suggests register as a candidate for creating the payment



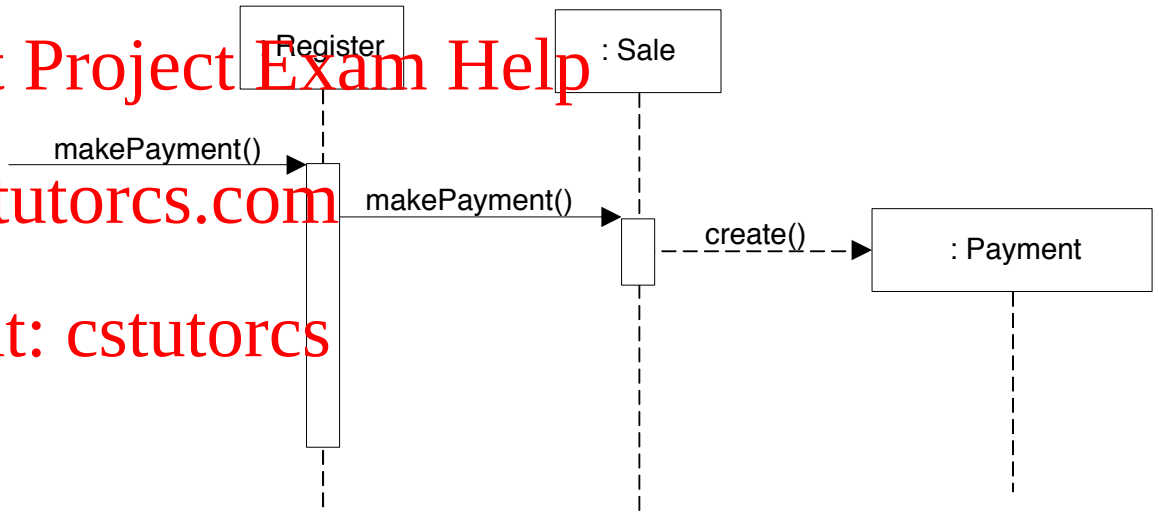
Acceptable but could become **incohesive** as the Register will increasingly need to carry out most of the system operations assigned to it
e.g., Register responsible for doing the work of 20 operations (overburden)

Cohesion – Example (NextGen POS)

Better design from cohesion point of view

The *payment* creation responsibility is delegated to the *Sale* instance

It supports high cohesion and low coupling



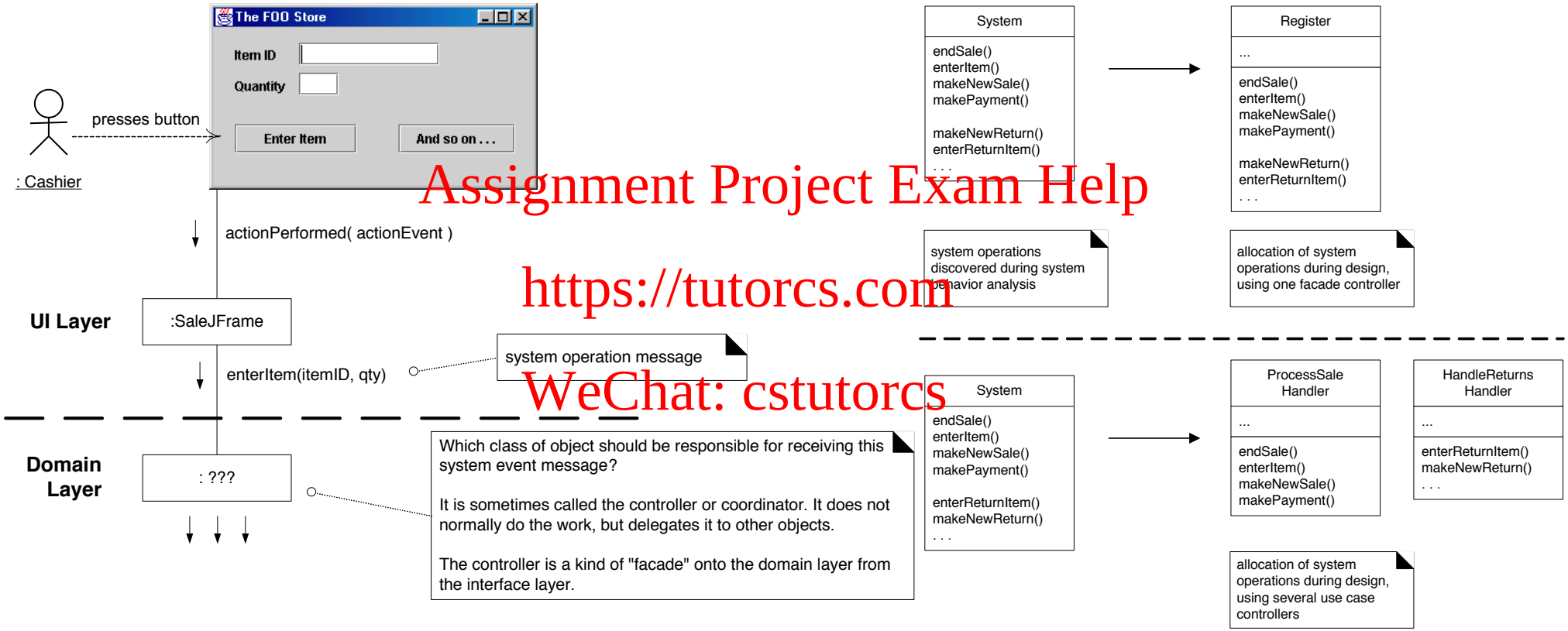
Coupling and Cohesion

- Coupling describes the inter-objects relationship
- Cohesion describes the intra-object relationship
- Extreme case of "coupling"
  - Only one class for the whole system
  - No coupling at all
  - Extremely low cohesion
- Extreme case of cohesion
  - Separate even a single concept into several classes
  - Very high cohesion
  - Extremely high coupling
- Domain model helps to identify concepts
- OOD helps to assign responsibilities to proper concepts

Controller

- Problem
  - What first object beyond the UI layer receives and coordinates ("controls") a system operation
- Solution
  - Assign the responsibility to an object representing one of these choices
    - Represents the overall system, root object, device or subsystem (a façade controller)
    - Represents a use case scenario within which the system operations occurs (a use case controller)

Controller



Task for Week 4

- Submit weekly exercise on canvas before 23.59pm Sunday
- Self learning on Next Gen POS system (Extended Version)
- Submit assignment 1 on canvas before its due.
  - ***All assignments are individual assignments***
  - Please note that: work must be done individually without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Robert Cecil Martin. 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Girish Suryanarayana et al. 2015. Refactoring for software design smells : managing technical debt. Waltham, Massachusetts ;: Morgan Kaufmann. Print.

What are we going to learn next week?

- Design Patterns
  - GoF Design Patterns
- Creational Design Patterns
  - Factory Method Pattern
  - Builder Pattern