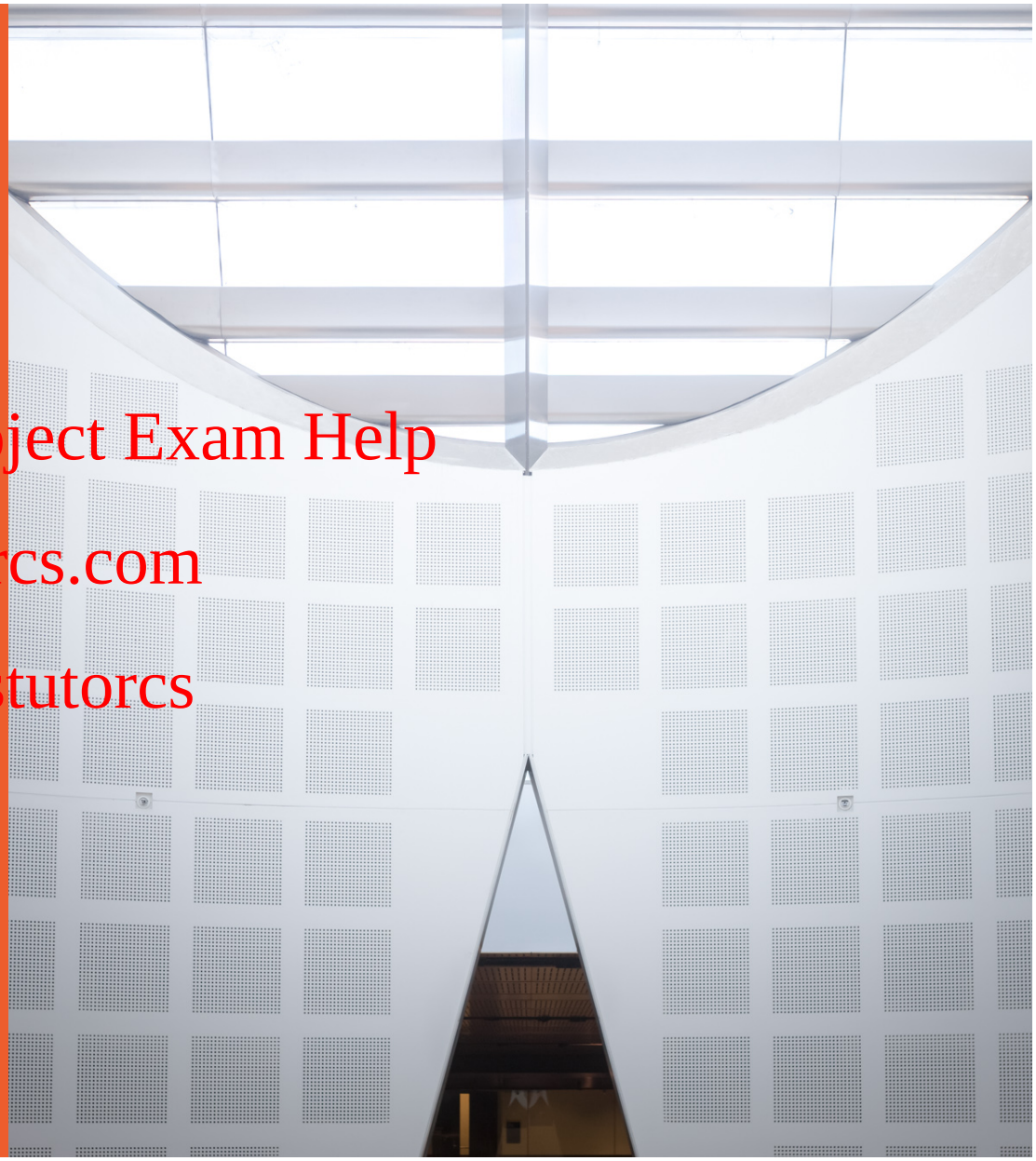
Software Design and Construction 1
SOFT2201 / COMP9201

Adapter and Observer

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.

Agenda

- Structural Design Pattern
  - Adapter
- Behavioural Design Pattern
  - Observer

Structural Design Patterns

THE UNIVERSITY OF SYDNEY

Structural Design Patterns

- How classes and objects are composed to form larger structures
- Structural *class* patterns use inheritance to compose interfaces or implementations
- Structural *object* patterns describe ways to compose objects to realise new functionality
  - The flexibility of object composition comes from the ability to change the composition at run-time

Structural Patterns (GoF)

Pattern Name	Description
Adapter	**Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface that clients expect.**
Façade	Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.
Decorator	Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)
Composite	Compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently
Proxy	Provide a placeholder for another object to control access to it

Adapter Pattern

Class, Object Structural

THE UNIVERSITY OF SYDNEY

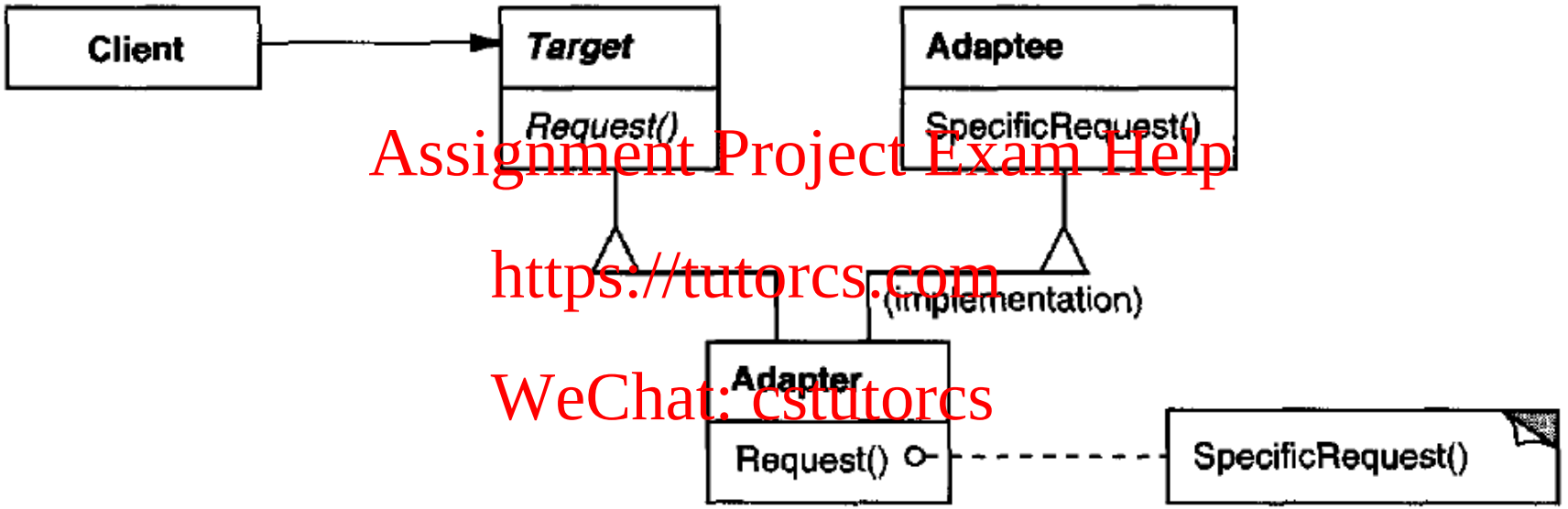
Adapter

- Intent
  - Convert the interface of a class into another interface that clients expect.
  - Lets classes work together that couldn't otherwise because of incompatible interfaces.
- Known as
  - Wrapper
- Motivation
  - Sometimes existing code that has the functionality we want doesn't have the right interface we want to use

Adapter

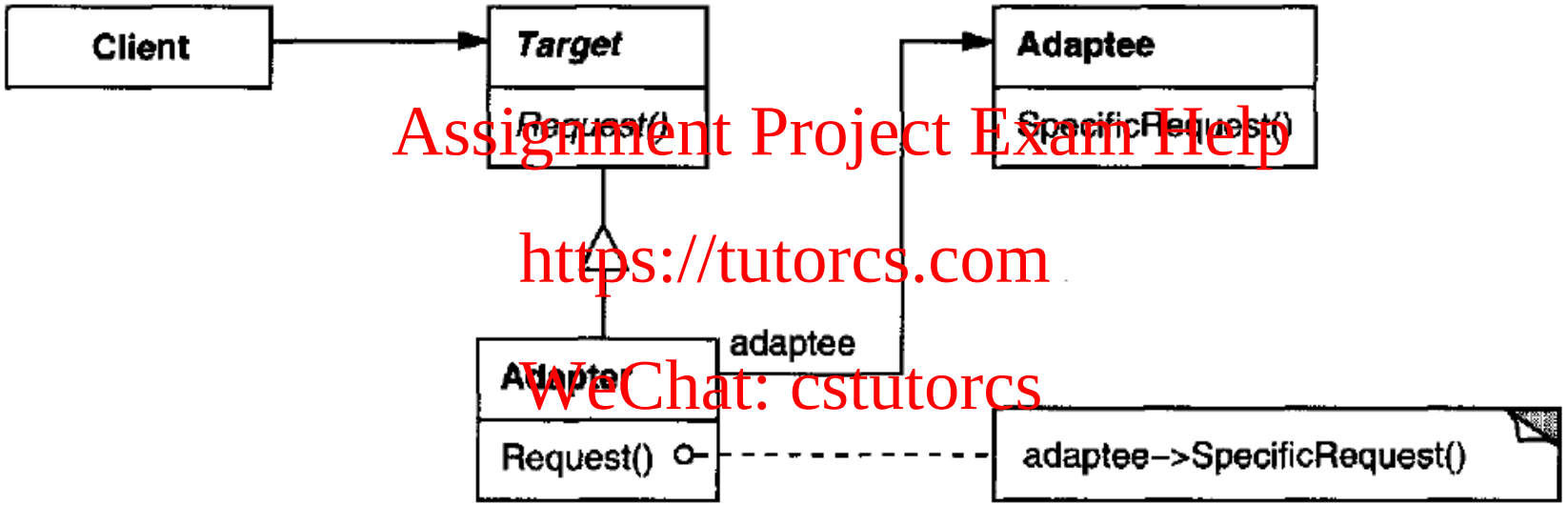
- Applicability
  - To use an existing class with an interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, i.e., classes that don't necessarily have compatible interfaces
  - (**Object adapter only**) Adapt an existing interface, which has several existing implementations.
- Benefits
  - Code reuse

Class Adapter – Structure



- Request multiple inheritance to adapt the Adaptee to Target, supported by C++

Object Adapter – Structure



Adapter – Participants

- **Target**
  - Defines the domain-specific interface that Client uses
- **Client**
  - Collaborates with objects conforming to the Target interface.
- **Adaptee**
  - Defines an existing interface that needs adapting.
- **Adapter**
  - Adapts the interface of Adaptee to the Target interface
- **Collaborations**
  - Clients call operations on an Adapter instance. In turn, the Adapter calls Adaptee operations that carry out the request

Class Adapter – Consequences

- When we want to adapt a class *and* all its subclasses, a class adapter won't work
  - It adapts Adaptee to Target by committing to a concrete Adaptee class

- Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee

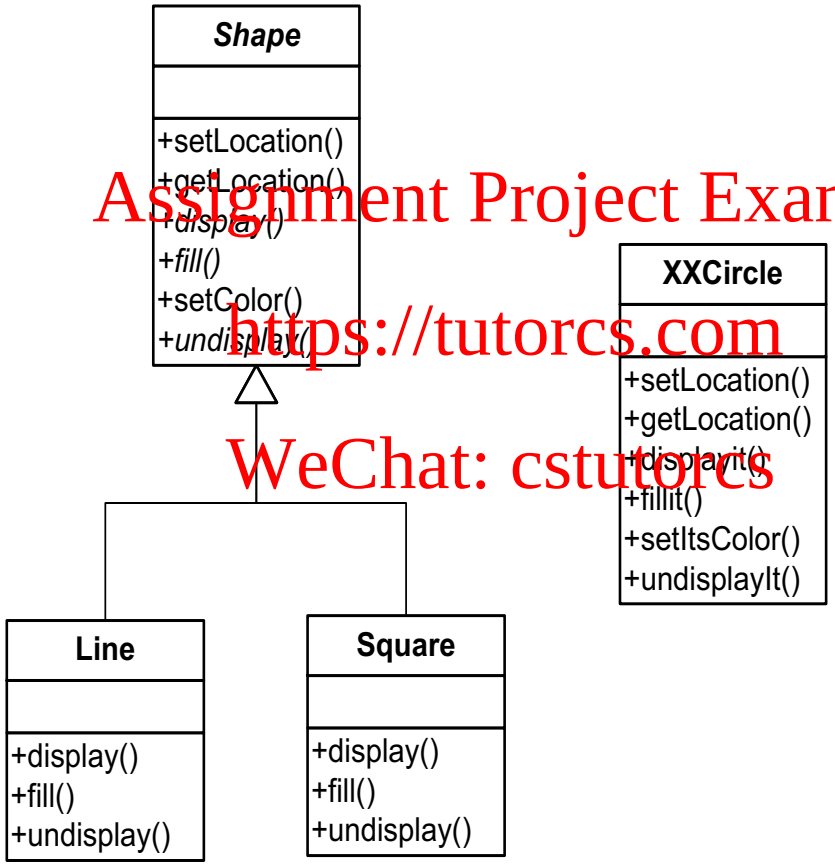
- Introduces only one object, and no additional pointer indirection is needed to get to the Adaptee (for programming language such as C++)

Object Adapter – Consequences

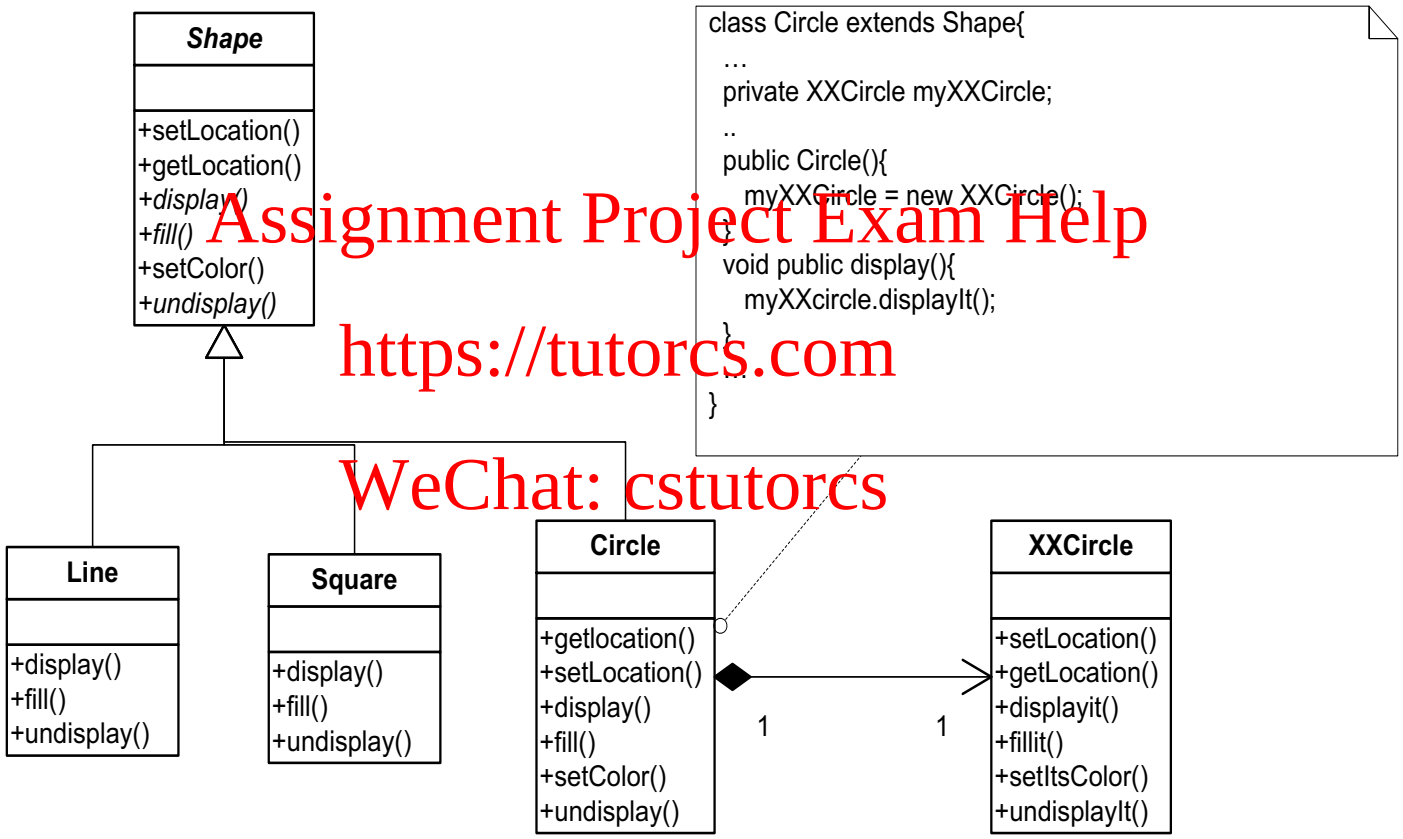
- Lets a single Adapter work with many Adaptees – i.e., the Adaptee itself and all of its subclasses (if any).

- Makes it harder to override Adaptee behavior. It will require sub-classing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself

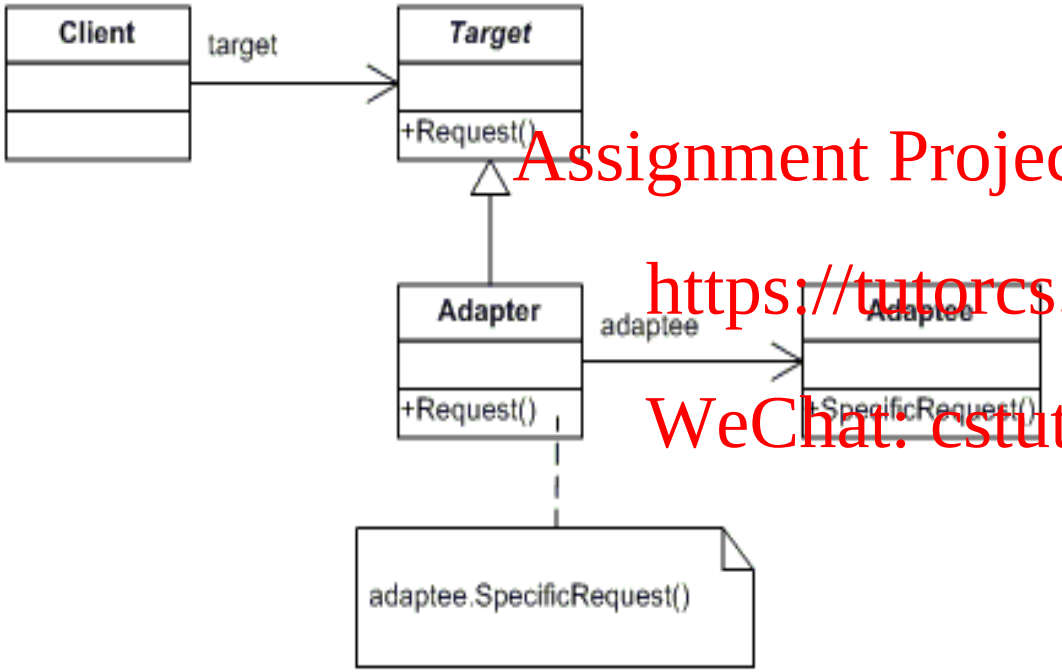
Adapter Example -- Problem



Adapter Example -- Solution

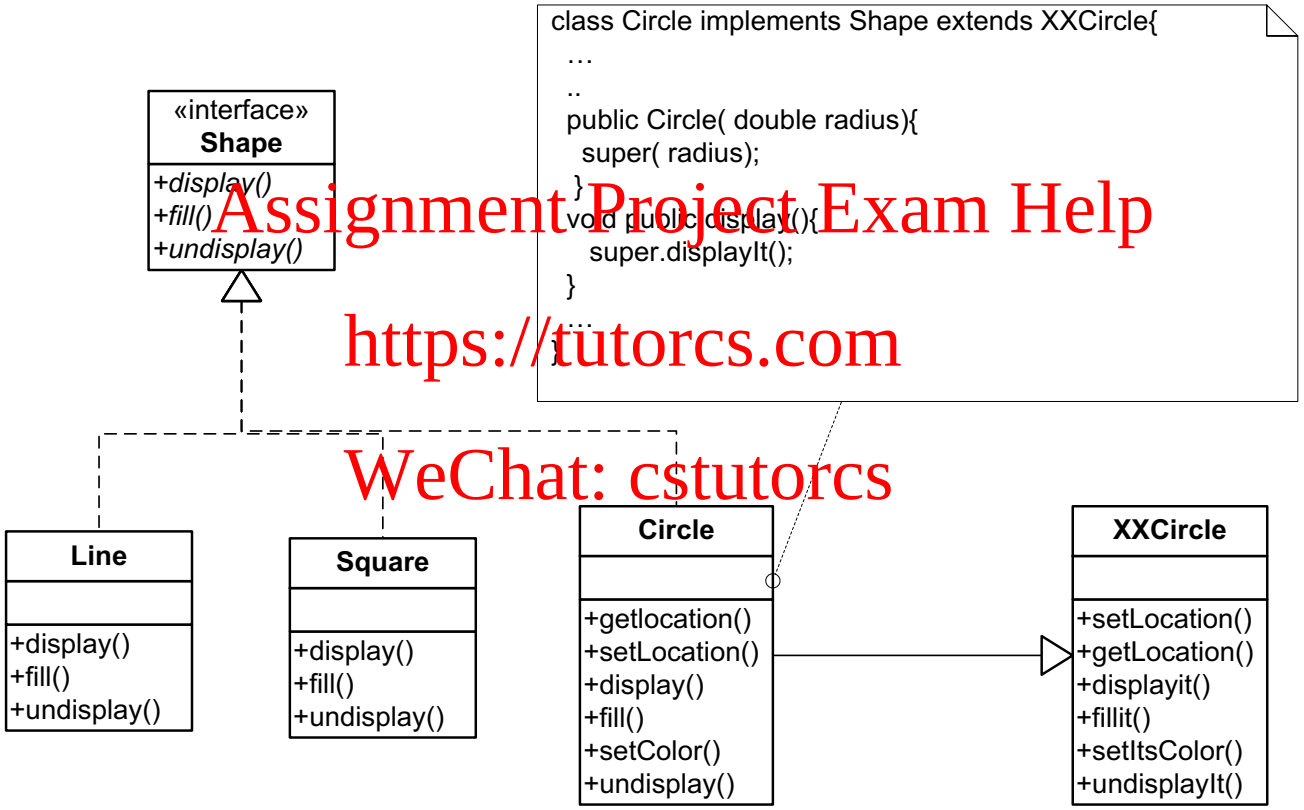


Adapter pattern: general structure



- **Target (Shape)**
  - defines the domain-specific interface that Client uses.
- **Adapter (Circle)**
  - adapts the interface **Adaptee** to the **Target** interface.
- **Adaptee (XXCircle)**
  - defines an existing interface that needs adapting.
- **Client (ShapeApp)**
  - collaborates with objects conforming to the Target interface.

Different Implementations of Adapter



Object Adapter and Class Adapter

- Object Adapter
  - Relies on object composition to achieve adapter
- Class Adapter
  - Relies on class inheritance to achieve adapter

Two Reuse Mechanisms

- Inheritance and Delegation
  - Inheritance: reuse by subclassing;
    - "is-a" relationship (**white-box reuse**)
  - Delegation: reuse by composition;
    - "has-a" relationship (**black-box reuse**)
    - A class is said to delegate another class if it implements an operation by resending a message to another class
- Rule of thumb – design principles #1
  - **Favour object composition over class inheritance**

Observer Pattern

Object Behavioural

THE UNIVERSITY OF SYDNEY

Observer

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Known as**
  - Dependents, Publish-Subscribe
- **Motivation:**
  - A collection of cooperating classes (consistency between related objects)
  - Consistency while maintaining loosely-coupled, and highly reusable classes

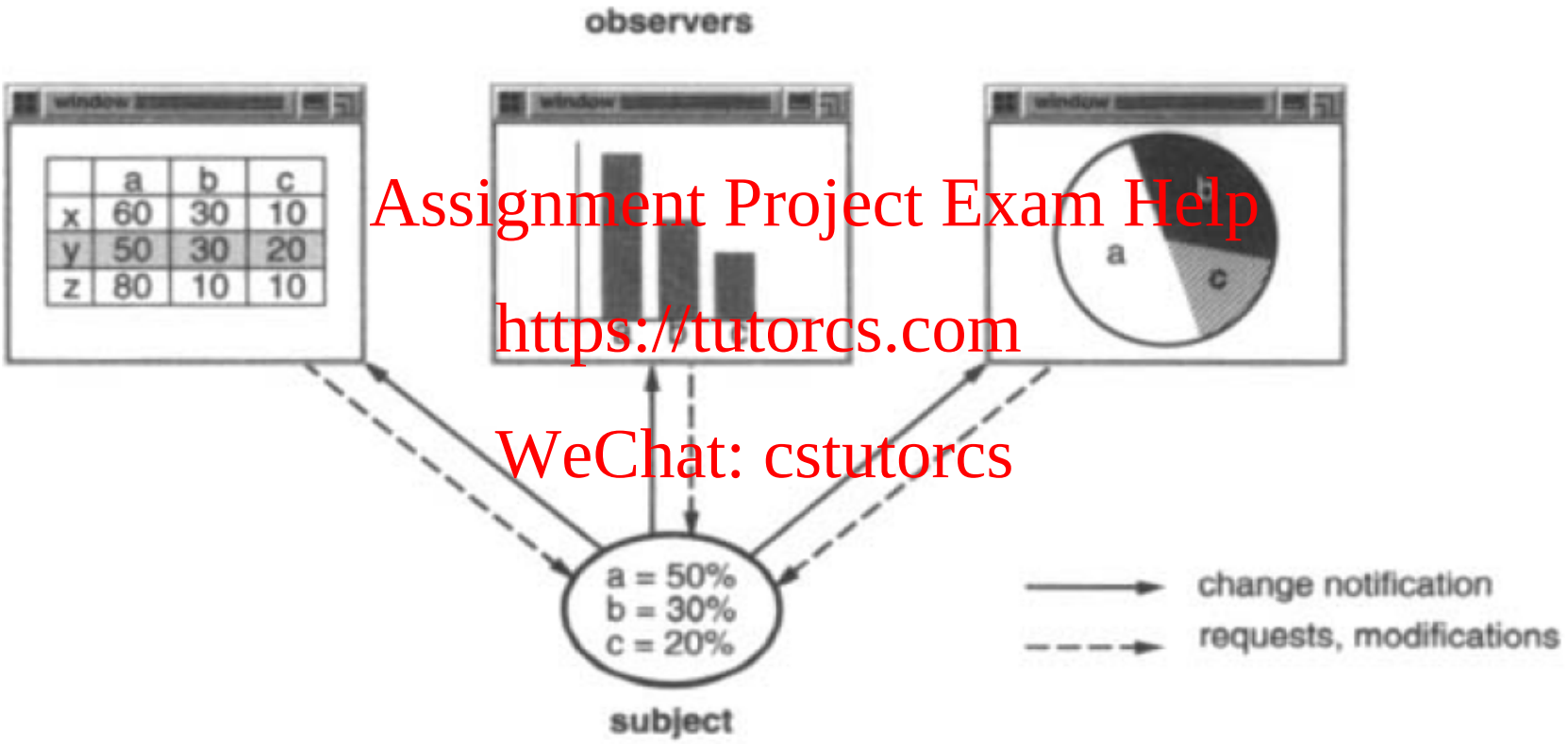
Observer – Applicability

- An abstraction has two aspects, one dependent on the other
- A change to one object requires changing others, and it's not clear how many objects need to be changed
- An object should be able to notify other objects without making assumptions about who these objects are

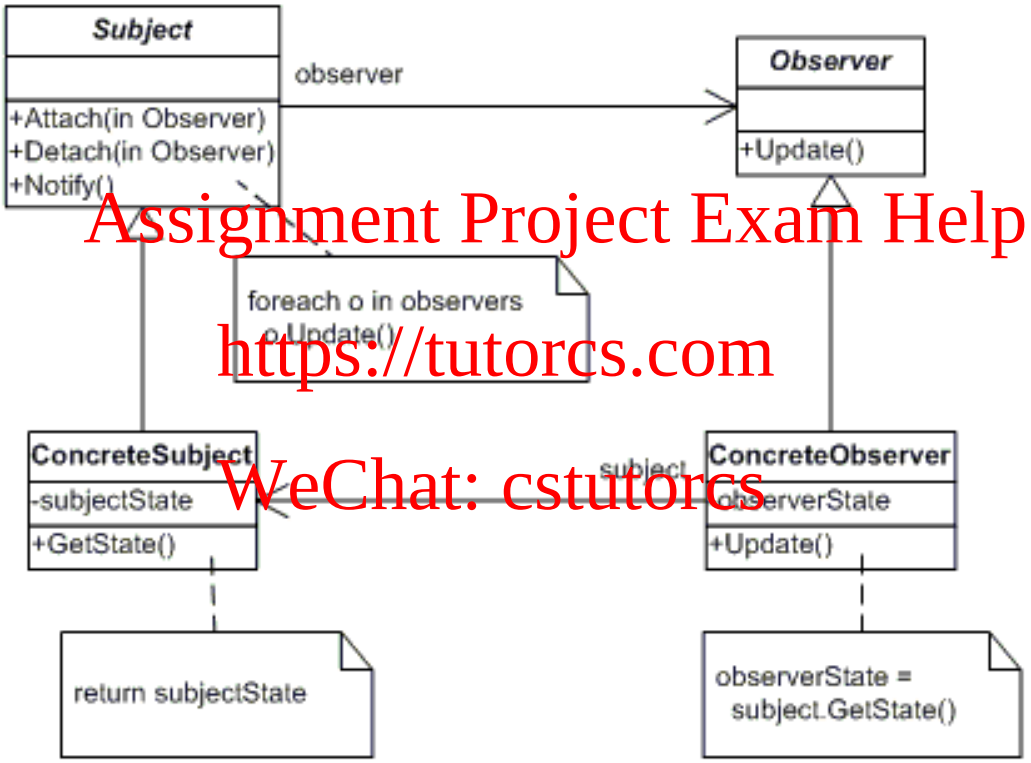
Observer – Publish-Subscribe

- Problem
  - You need to notify a varying list of objects that an event has occurred
- Solution
  - Subscriber/listener interface
  - Publisher: dynamically register interested subscribers and notify them when an event occurs

Observer – Example (Data Representation)



Observer – Structure



Observer – Participants

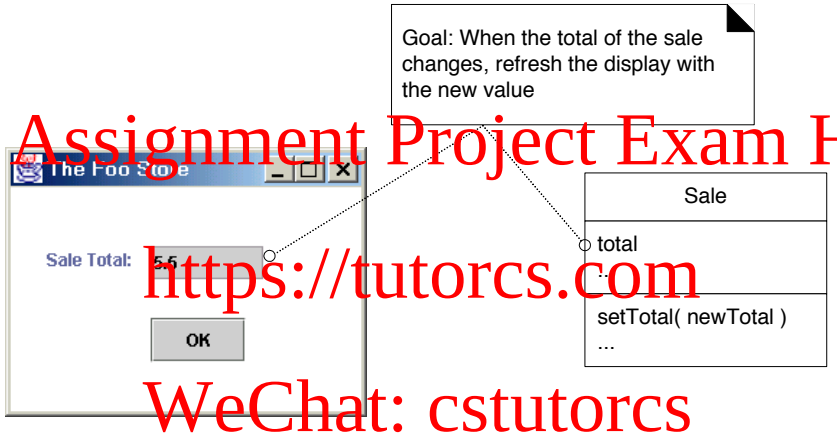
Participant	Goals
Subject	Knows its observers. Any number of observer objects may observe a subject. Provides an interface for attaching and detaching observer objects
Observer	Defines an updating interface for objects that should be notified of changes in a subject
ConcreteSubject	Stores state of interest to ConcreteObserver objects Sends notifications to its observers when its state changes
ConcreteObserver	Maintains a reference to a ConcreteSubject object Stores state that should stay consistent with the subject's. Implements the observer's updating interface to keep its state consistent

Observer – Consequences

- Abstract coupling between Subject and Observer
  - *Subject* only knows its *Observers* through the abstract *Observer* class (it doesn't know the concrete class of any observer)
- Support for broadcast communication
  - Notifications are broadcast automatically to all interested objects that subscribe to the *Subject*
  - Add/remove Observers anytime
- Unexpected updates
  - Observers have no knowledge of each other's presence, so they can be blind to the cost of changing the subject
  - An innocent operation on the subject may cause a cascade of updates to Observers and their dependents

Observer – NextGen PoS System

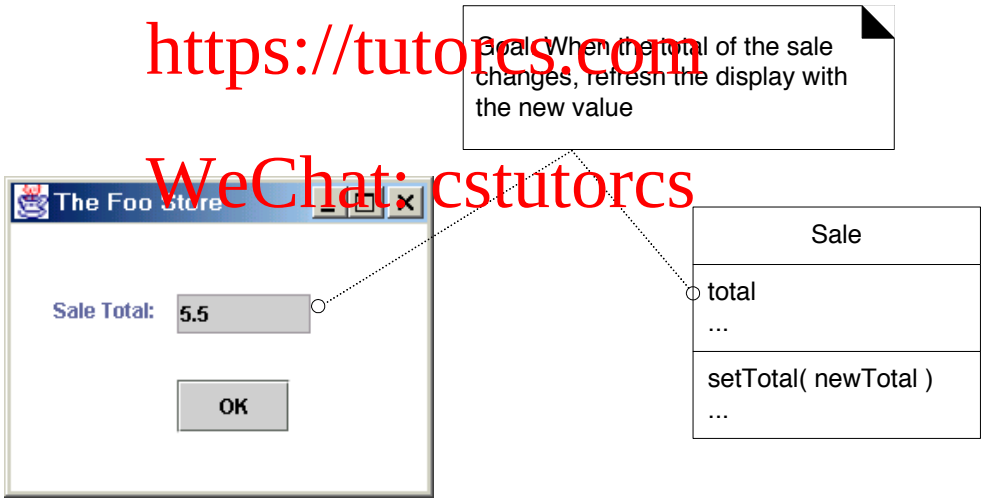
- PoS system requirement (iteration 2):



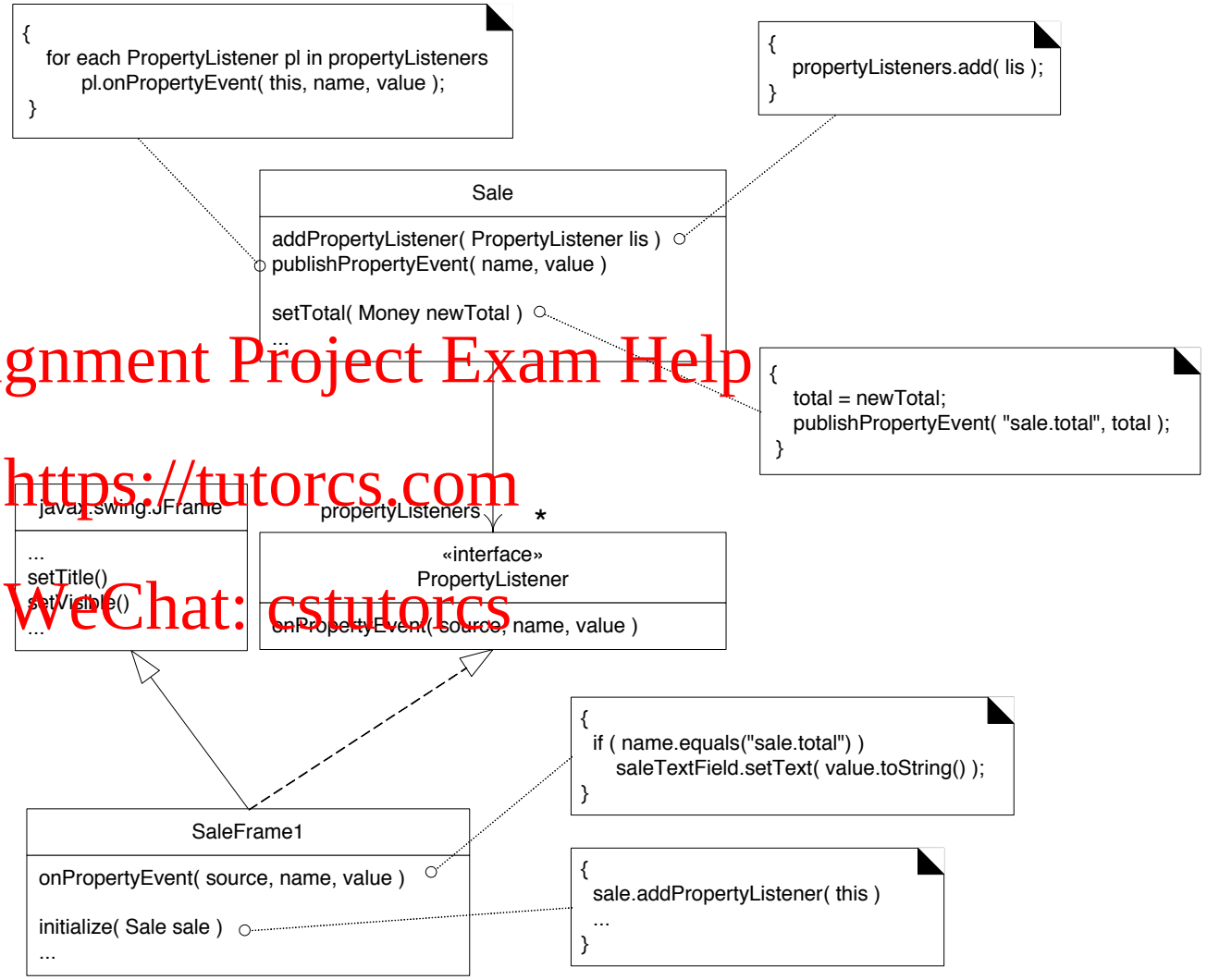
One solution:
When the *Sale* object changes its total, the Sale object sends a message to a window (GUI), asking it to refresh its display
Discuss: good/bad solution? Why/why not?

Observer – NextGen PoS System

- Model-view separation principle (low-coupling of model and UI layers)
  - "model" (e.g., Sale object) should not know/update "view" presentation objects (e.g., window)
  - Allows replacing (or changing) the UI without influencing the model (Sale object)



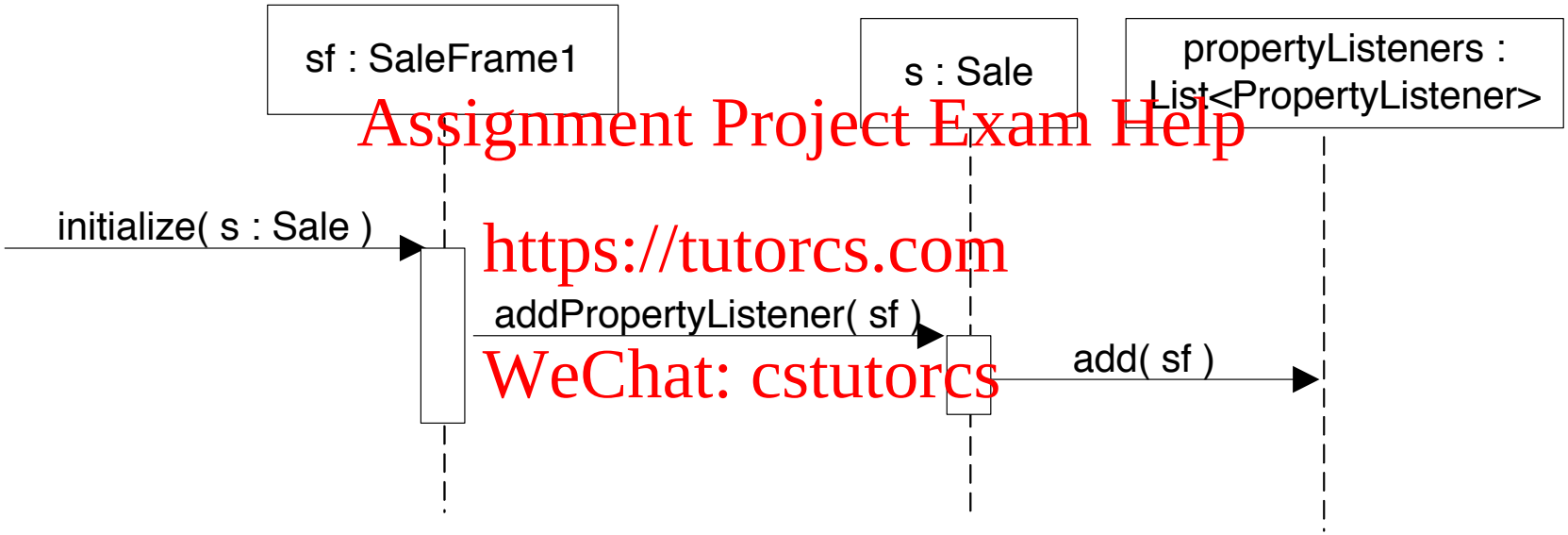
Observer – NextGen POS Solution



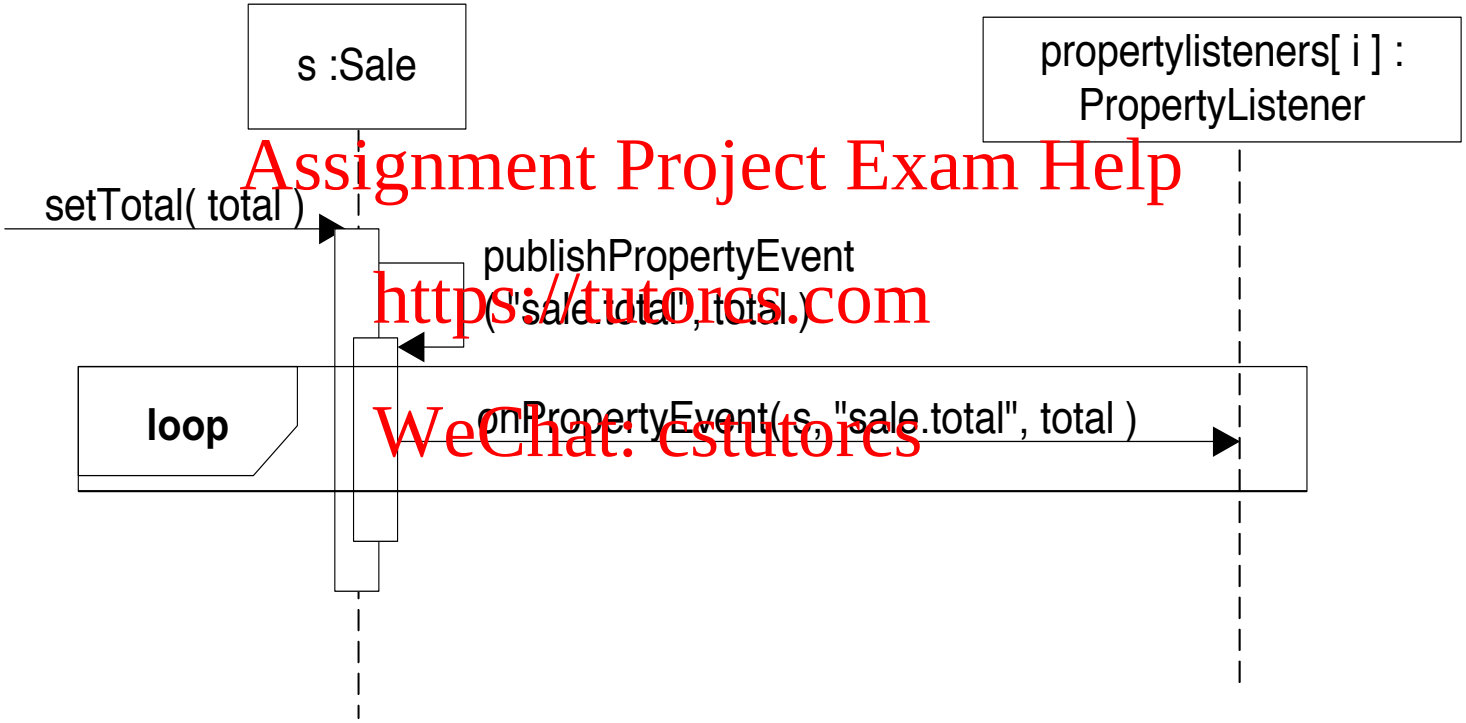
Observer – NexTGen POS Solution

- An interface is defined – **`PropertyListener`** with the operation **`onPropertyEvent`**
- Define an UI object to implement the interface – **`SalesFrame1`**
- When the SalesFrame1 window is initialised, pass it the `Sale` instance from which it is displaying the total
- The **`SaleFrame1`** window registers or subscribes to the **`Sale`** instance for notification of "property events", via the **`addPropertyListener`** message.
- The **`Sale`** instance, once its total changes, iterates across all subscribing **`PropertyListeners`**, notifying each

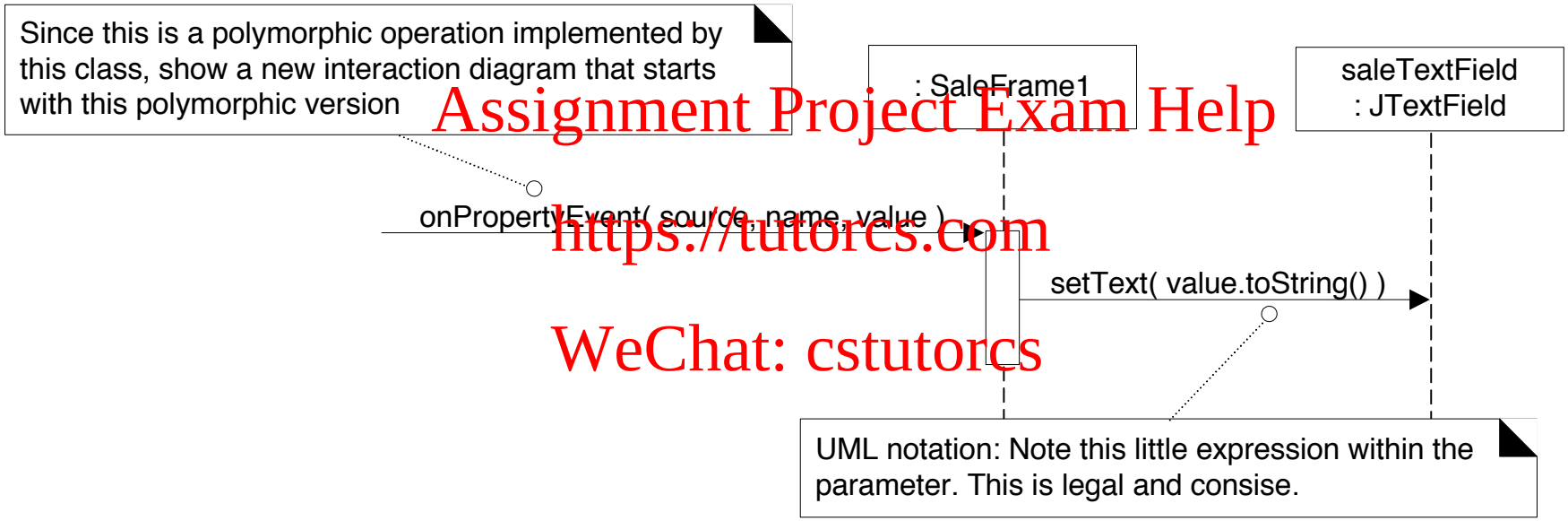
Observer – Subscribe To Publisher



Sale publishes a property events to all subscribers



Subscriber Receives A Notification



Task for Week 9

- Submit weekly exercise on canvas before 23.59pm Sunday
- Well organize time for assignment 3 once it is released
- Attend Helpdesk session if you have any questions/difficulties on implementation perspective
- Prepare questions and ask during tutor Q&A session this week
- Self learning on Next Gen POS system once it is released today

What are we going to learn on week 10?

- Creational Design Pattern
  - Prototype
- Behavioral Design pattern
  - Memento

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.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.