COMP 3711

OOD

More GRASP PATTERNS

Larman Chapter 25

Nine **GRASP** Principles

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

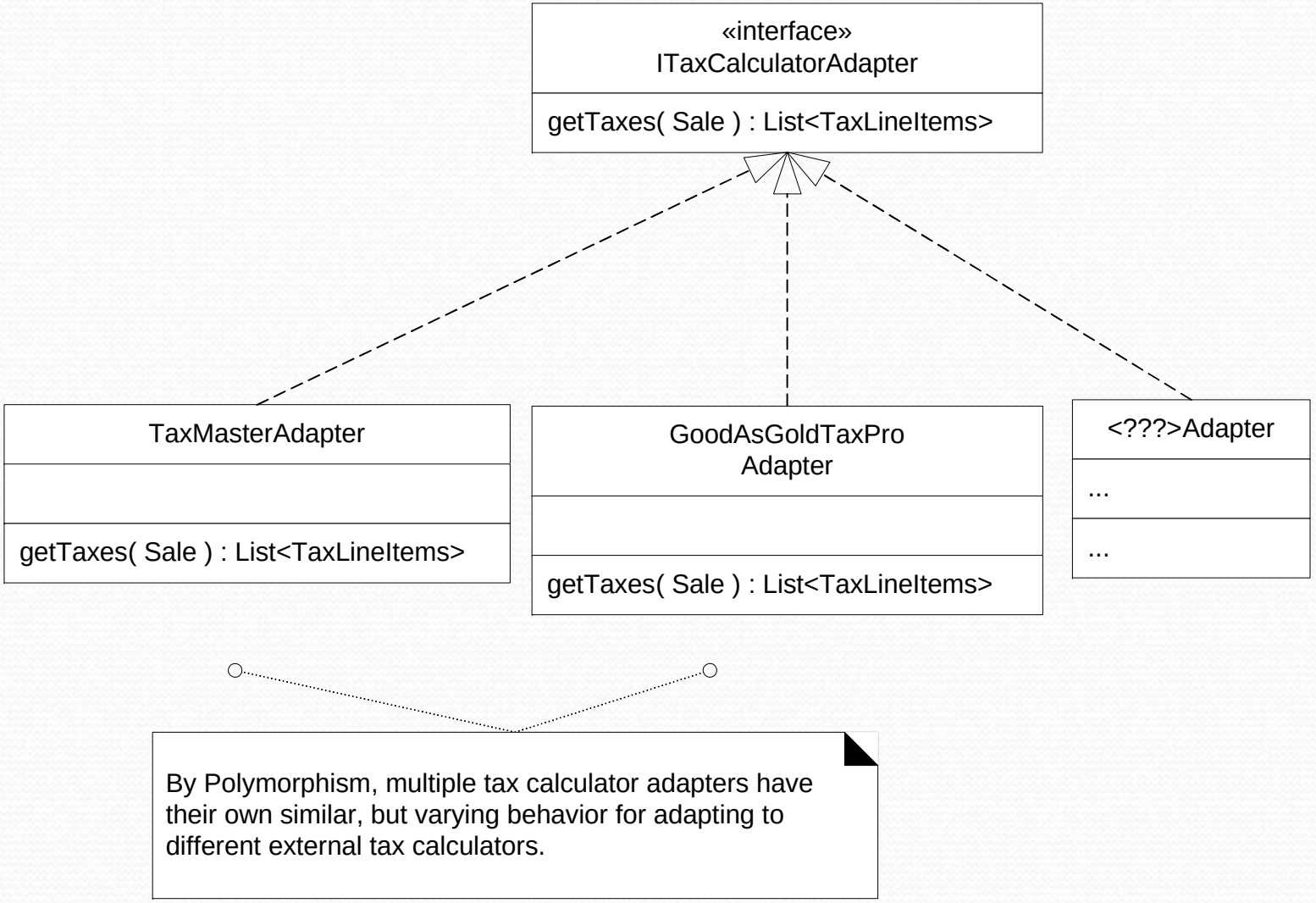
The Last 4 principles

Polymorphism Pattern

- "Giving the same name to services in different objects when the services are similar or related." [Coad95]

- NextGen POS example :
  - Implement getTaxes polymorphic operation, assigned the responsibility for adaptation to different tax calculation objects (similar but with varying behaviour to adapt to each external interfaces)

Example – Polymorphic getTaxes



Larman fig. 25.1

Polymorphism Pattern

- Polymoprhism often is implemented with abstract superclasses or interfaces
- Consider making an interface corresponds to the public method signatures of the abstract superclass and declare the abstract superclass to implement the interface
- Beware not over-do for “future-proofing” against possible unknown variation that may never happen

Pure Fabrication Pattern

- What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling or other goals, and solutions offered by other principles are not appropriate?

- Often poor cohesion and coupling problems arise when assigning responsibilities only to classes in domain layer

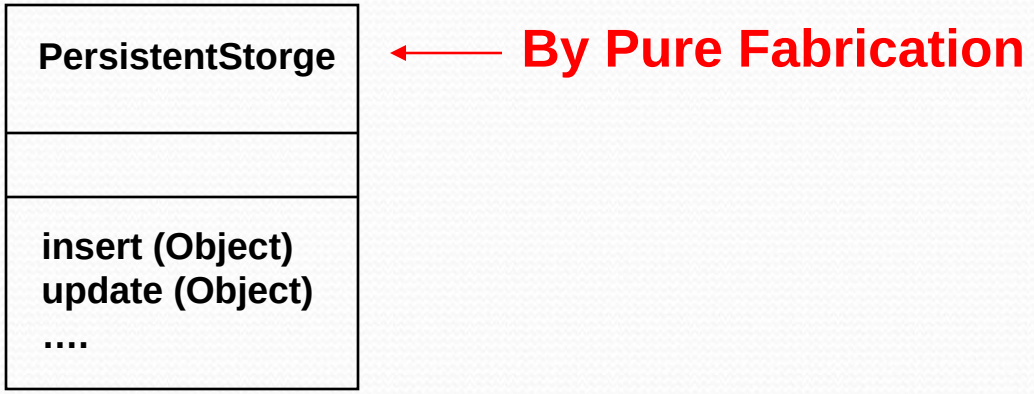
Pure Fabrication Pattern

- One of the solutions to address the above problem is to assign a “pure fabrication” class of high cohesive and low coupling set of responsibilities, by using imagination.

- “Making something up when we are desperate”

Pure Fabrication Pattern

- NextGen POS example:
  - Instead of placing the responsibility in the Sale class for saving the Sale instances in the database, which is what we would do based on Information Expert, create a new "pure fabrication" class that is solely responsible for saving objects in some kind of persistent storage medium.



Pure Fabrication Pattern

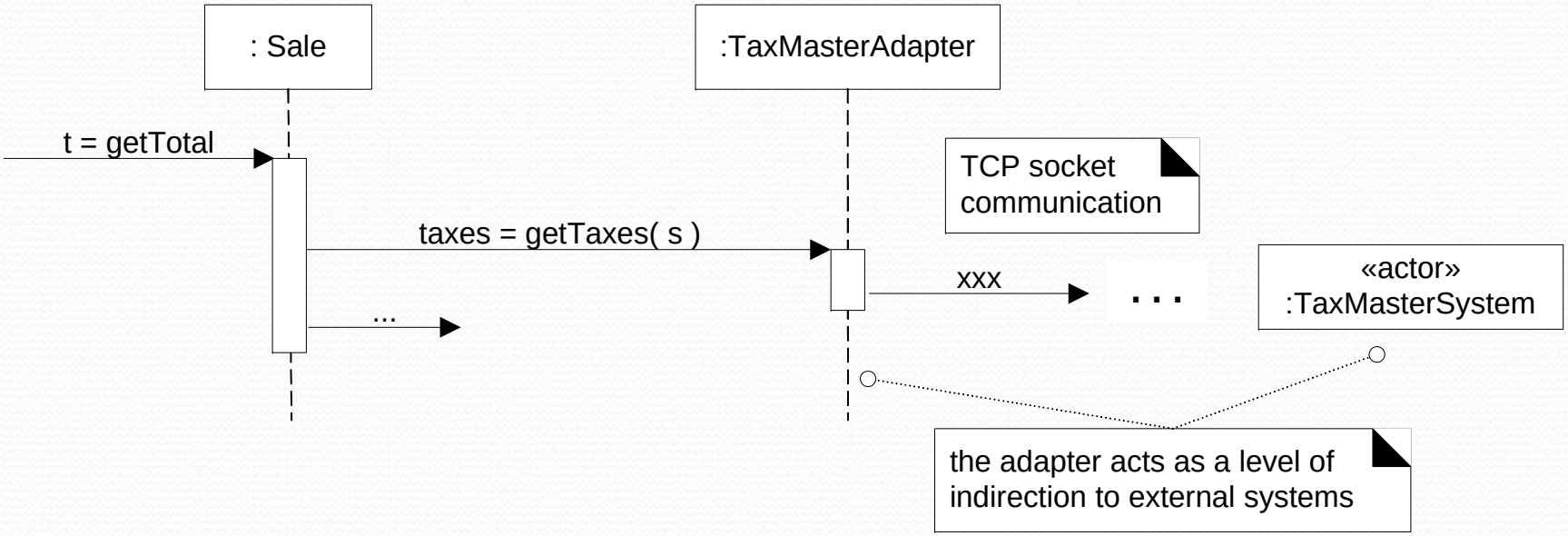
- Benefits of pure fabrication:
  - Sale remains well-designed with high cohesion and low coupling
  - PersistantStorage class itself is cohesive with the sole purpose of storing objects in the database
  - PersistentStorage class is generic enough as an resuable object

Indirection Pattern

- When to assign a responsibility to avoid direct coupling between two (or more) things?
- How to de-couple objects so that low coupling is supported and reusability possible?
- Solution:
  - Assign responsibility to an intermediate object to mediate between other components or services to avoid direct coupling

Indirection Pattern

- NextGen POS example:
  - Implement polymorphic adaptor object TaxMasterAdapter to add a level of indirection thus avoiding direct coupling
- ”



Larman fig. 25.10

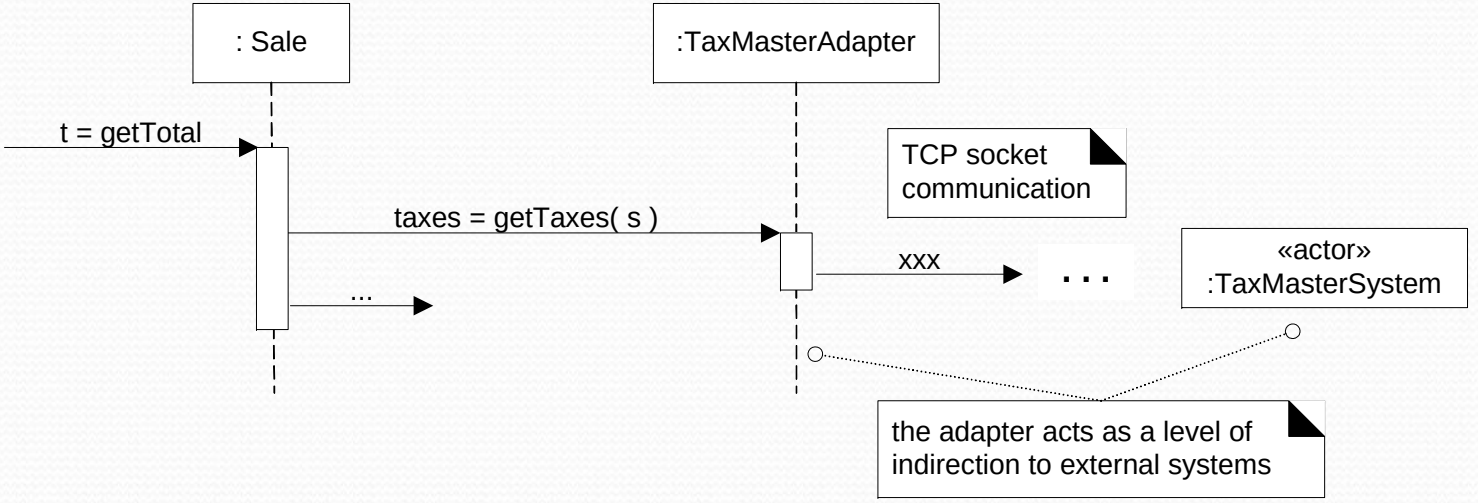
Protected Variations Pattern

- How to design object so that the variations or instability does not have undesirable impact on other objects?

- Solution:
  - Assign responsibility to create a stable interface wound those points of instability or predicted variation ,
  - The Protected Variation concept was first introduced by Cockburn in 1996 and is sometimes referred to information hiding. Other OO principles such as encapsulation, interfaces are motivated by PV.

Protected Variations Pattern

- NextGen POS example:
  - The polymorphic adaptor object TaxMasterAdapter is an example of introducing a level of indirection (interface) protecting the NextGen system form variations in external taxCalculator APIs



See discussions in Larman pp 427-434

Patterns Implementation

- Many mechanisms to implement Protected Variations
- In POS example the following are used:
  - Polymorphism, level of indirection, interface
    1. Polymorphism: Method names are the same
    2. Indirection: Adapter objects
    3. Protected Variations: The stable interface
- Protection within the system from variations in external APIs is achieved.

Do Not Talk To Strangers

- Used to be one of GRASP
- Now special case of Protected Variations
- “Avoid creating designs send messages (or talk) to distant, indirect (stranger) objects. “

Do Not Talk To Strangers

- Within a method, messages should only be sent to the following objects:
  1. The this object (or self).
  2. A parameter of the method.
  3. An attribute of this.
  4. An element of a collection which is an attribute of this.
  5. An object created within the method.
  6. The intent is to avoid coupling a client to knowledge of indirect objects and the object connections between objects.

Example

```
class Register {
   private Sale sale;
   public void slightlyFragileMethod() {

   // sale.getPayment() sends a message to a "familiar" (passes #3)
   // but in sale.getPayment().getTenderedAmount()
   // the getTenderedAmount() message is to a "stranger" Payment

      Money amount = sale.getPayment().getTenderedAmount();

    // ... }
// ... }
```

Example

```
public void moreFragileMethod() {

AccountHolder holder =
    sale.getPayment().getAccount().getAccountHolder();

// ... }
```

Your Design Choice

- Strictly obeying this law requires adding new public operations to the "familiars" of an object These operations provide the desired information and hide how it was obtained.

- Example:

  AccountHolder holder = sale.getAccountHolderOfPayment();

- **`Pick your battles!`**