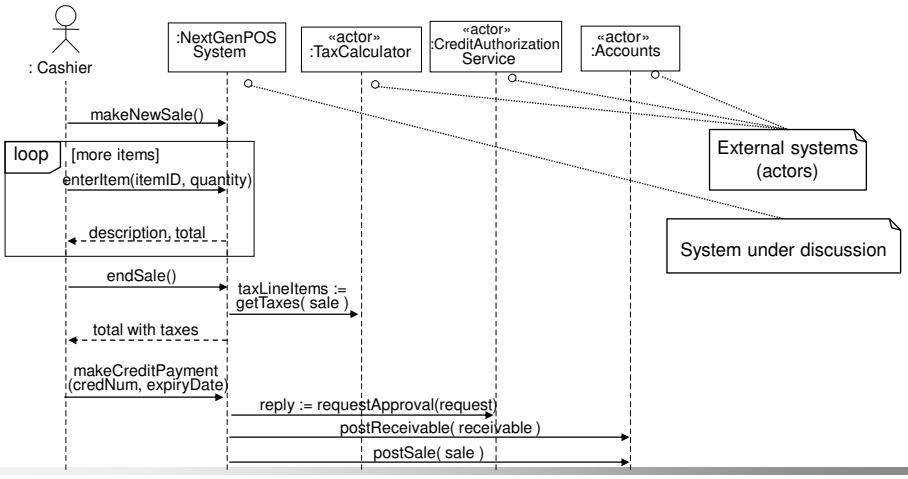
2nd Iteration, GRASP Patterns 2

In the first iteration, we implemented operations of the basic flow of the "UC1- Process Sale" use case.

In the second, some alternative flows will be implemented.

Use case:



New requirements in the 2nd Iteration:

In the second iteration, we will deal with the following problems:

- Collaboration with external systems, such as Credit Card Authorization, Accounting, etc., will be handled.
  - The main problem with connections to external systems is that they are not under our control.
  - They may have different interfaces.
  - They may change or be replaced. Our system should not be affected by these changes.
- Pricing strategies (discounts):
  - The store can apply different discounts according to various parameters such as date, product, total of sale, and customer type.
  - Customers can have loyalty cards, which provide them with some benefits.
  - Discount strategies may change.

Writing use cases and domain analysis in the second iteration:

Use cases:

We can continue to work on the use cases we wrote in previous iterations if we have not entirely realized (designed) them.

For example, in our exemplary system, we do not need to write new use cases in the second iteration.

However, in later iterations, it will be necessary to write new use cases such as UC2: Handel Returns.

Domain Analysis in the second iteration:

In our exemplary system, analysis in the second iteration may take shorter because there are few new conceptual classes.

Old domain models from the first iteration (3.17) are not used.

We use the source code or design model (5.19) of the first iteration as the source (domain) model for the second iteration because they include our decisions.

Usually, a UML tool is used to reverse engineer diagrams from the source code of the last iteration.

New conceptual classes, such as tax and credit cards, will be added to this model.

GRASP 2

Previously, we have seen 5 GRASP patterns.

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

Now, we will discuss the remaining 4 GRASP patterns:

- Pure Fabrication
- Indirection
- Polymorphism
- Protected Variations

Later, we will discuss widely used GoF patterns, which also cover topics handled by GRASP patterns.

Pure Fabrication (GRASP)

In OOD, we are usually inspired by the real world (domain).

To achieve the goal of the low representational gap, we create software classes from real-world classes by assigning responsibilities to them.

For example, Sale, Payment, Student, Course, and Book classes.

But sometimes, assigning responsibilities only to domain layer software classes leads to problems in terms of poor cohesion, coupling, or low reuse potential.

Problem:

What object should be responsible when you do not want to violate High Cohesion and Low Coupling or other goals, but the solutions offered by Expert (for example) are inappropriate?

Solution:

Assign a highly cohesive set of responsibilities to an **artificial** class that does not represent a problem domain (real-world) concept to support high cohesion, low coupling, and reuse.

The new artificial class is a f**abrication** of the imagination.

The responsibilities assigned to this fabrication must support high cohesion and low coupling so the fabrication design is very clean or pure, hence a **pure fabrication**.

Example: Saving a Sale Object in a Database

Responsibility: It is necessary to save Sale instances in a relational database.

Who will get the responsibility?

Information Expert: "Assign this responsibility to the Sale class itself because the Sale has the data that needs to be saved."

However, there are some implications:

- The task requires many database-oriented operations, none related to the concept of sale-ness, so the Sale class becomes incohesive.
- The Sale class has to be coupled to the relational database interface (such as JDBC) so its coupling goes up.

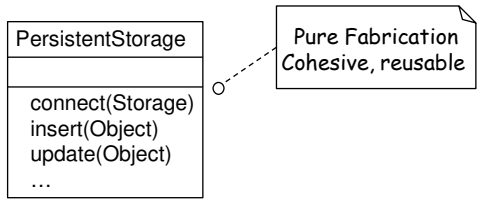
  The reusability potential of this Sale class is low. In another Project, Sale may not be saved in a database.
- Saving objects in a relational database is a general task for which many classes (e.g., Customer, Payment, etc.) need support.

  Placing these responsibilities in the Sale class causes poor reuse or lots of duplication in other classes that do the same thing.

Solution with the Pure Fabrication:

A reasonable solution is to create a new class that is solely responsible for saving objects in some kind of persistent storage medium (PersistentStorage).

This class is Pure Fabrication.



Advantages:

- The Sale remains well-designed, with high cohesion and low coupling.
- The PersistentStorage class is relatively cohesive, with the sole purpose of storing or inserting objects in a persistent storage medium.
- The PersistentStorage class is a very generic and reusable object.

Warning: Do not put unrelated responsibilities (functions) in the same class.

It must be "**pure**".

Discussion:

There are two common methods to identify software classes:

1. Representational decomposition.
2. Behavioral decomposition.

By **representational decomposition,** the software class relates to or represents a thing in a domain (real world).

For example, Sale, Book, Customer, etc.

Representational decomposition is a common strategy in object design and supports the goal of the low representational gap.

However, sometimes (because of cohesion, coupling, and reusability), we group some related behavior or methods in an artificial class.

These artificial classes are inspired by **behavioral decomposition**.

A Pure Fabrication is a function-centric or behavioral object.

Warning:

Behavioral decomposition into Pure Fabrication objects is sometimes overused by designers who are new to object design and more familiar with procedural (imperative) programming.

The usage of pure fabrication needs to be balanced with the ability to design with representational decomposition.

The representational (domain) classes should take the main responsibilities.

Artificial classes (fabrication) should support the representational software classes in fulfilling their responsibilities.

Main classes: Representational classes

Helper (Supporting) classes: Artificial (behavioral) classes

Indirection (GRASP)

Sometimes, objects must interact with other objects or external systems, which may change (or be replaced) in the future.

Direct coupling to such objects or systems may result in modifications in our objects.

Problem:

Where should we assign a responsibility to avoid direct coupling between two (or more) things?
How can we decouple objects so that low coupling is supported and reuse potential remains higher?

Solution:

Assign the responsibility to an **intermediate object** to mediate between other components or services so that they are not directly coupled.
The intermediary creates an **indirection** between the other components.

Example:

The "Pure Fabrication" PersistentStorage class is also an example of assigning responsibilities to support Indirection.

The PersistentStorage acts as an intermediary between the Sale and the database.

The change in the database (ideally) will not affect the Sale.

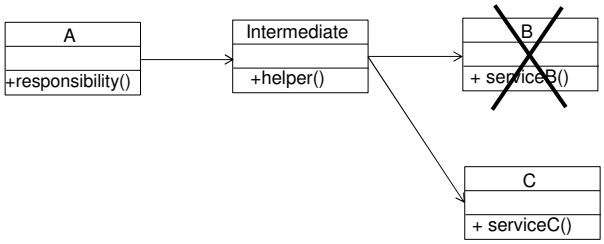
Example: The intermediate class

Problems:

Class A gets services from class B.

In the future, class B may change or may be replaced.

We want to protect class A from the changes in class B.



Polymorphism (GRASP)

Remember, polymorphism is one of the fundamental properties of object-oriented programming. (What is polymorphism at the programming level?) ***OOP*!**

Sometimes alternatives or different behavior of a program are based on type (class).

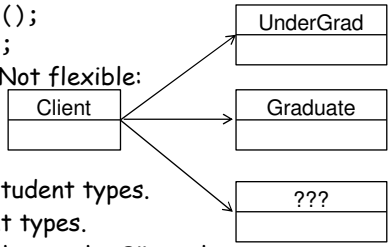
If a program is designed using if-then-else or switch-case statements, then when a new variation or a type (class) arises, it often requires modification of the case logic in many places.

This approach makes it difficult to extend a program with new variations easily.

Example: A part of a class (Client) that operates on different student types.

```
if (studentType == underGrad) doSomething();
if (studentType == grad) doAnotherthing();
        else  doSomethingElse();
…
```

Not flexible:



The class Client must be aware of all different student types.

It is strongly affected by the changes in student types.

If we add (or remove) a student type, we must change the Client class.

Definition of the polymorphism pattern:

Problem:

How to handle alternatives based on type? How to create pluggable software components?

Pluggable software components: In client-server relationships, how can you replace one server component with another without affecting the client?

Solution:

When related alternatives or behaviors vary by type (class), assign responsibility for the behavior using polymorphic operations.
Do not test for the type of object.

With the help of polymorphism, one object (client) can send messages to other objects without being aware of (without knowing) their actual type (class).

The calling (client) object knows only other objects' süper type (base).

Polymorphism provides two advantages:

1. We can change the behavior of the Client object in run-time.
2. If we add new classes derived from the same base to the system, the Client class does not need to be changed.

OOP Principle: Design to interface and not concrete classes.

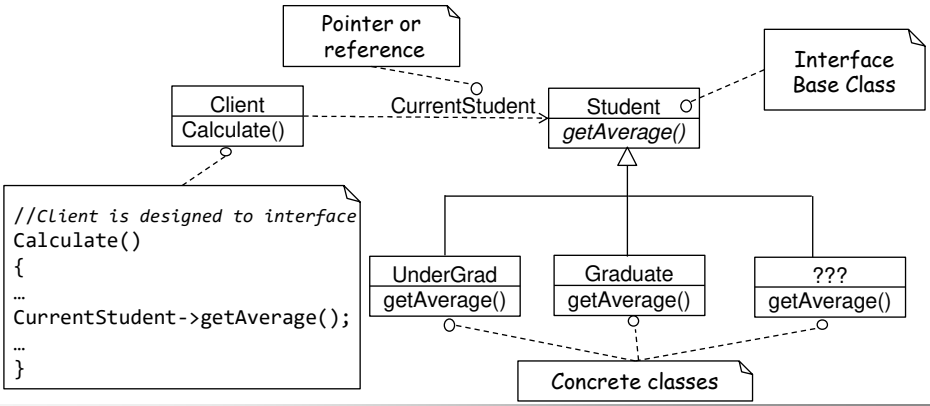
Program to an interface, not an implementation.

Polymorphism is the main mechanism of many principles and design patterns.

One of the essential principles is "design to interface".

Example:

A Client class that gets services from different Student classes.



Example:

We design a graphics library that includes different types of shapes.

A client object (or a system) gets services (calls methods) from this library.

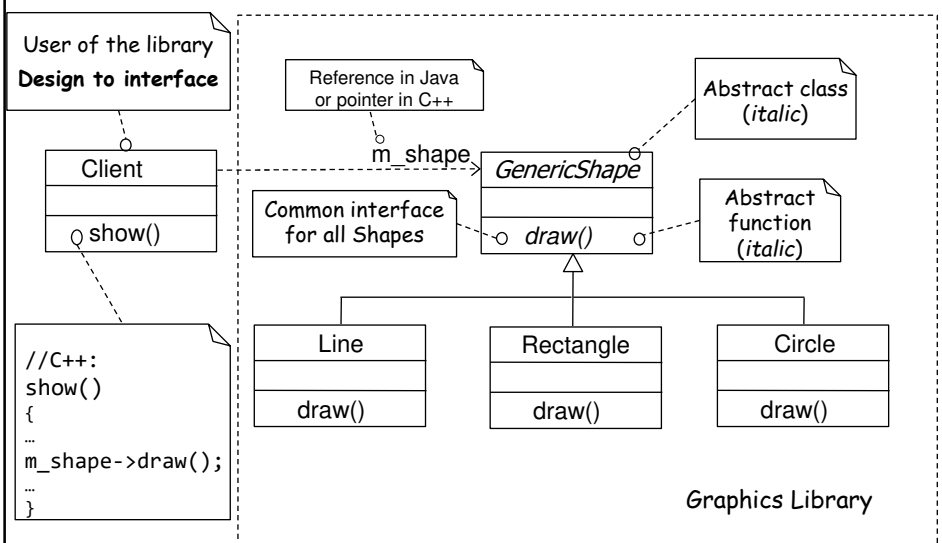
Adding new shapes to the library (or maybe removing some of them) is possible.

During design, the client class does not know the concrete type of the shape (object) to which it will be connected.

The client class is designed according to the common interface of the shapes.

Changes in the graphics library will not affect the client (user) system.

Example: Design of Graphics Library



C++

```
class GenericShape{                              // Abstract base class
public:
  virtual void draw() const =0;     // pure virtual function (polymorphism)
};

class Line:public GenericShape{                  // Line class (concrete)
public:
  Line(int x_in, int y_in, int x2_in, int y2_in);     // Constructor
  void draw() const override;           // concrete draw function of line
private:
  int m_x1, m_y1, m_x2, m_y2;                    // Coordinates of line
};

Class Rectangle:public GenericShape{             // Rectangle class (concrete)
:
  void draw() const override;                    // concrete draw of rectangle
};

class Circle:public GenericShape{                // Circle class (concrete)
public:
  Circle(int x_cen, int y_cen, int r);           // Constructor
  void draw() const override;                    // concrete draw of circle
private:
  int m_centerX, m_centerY, m_radius;
};
```



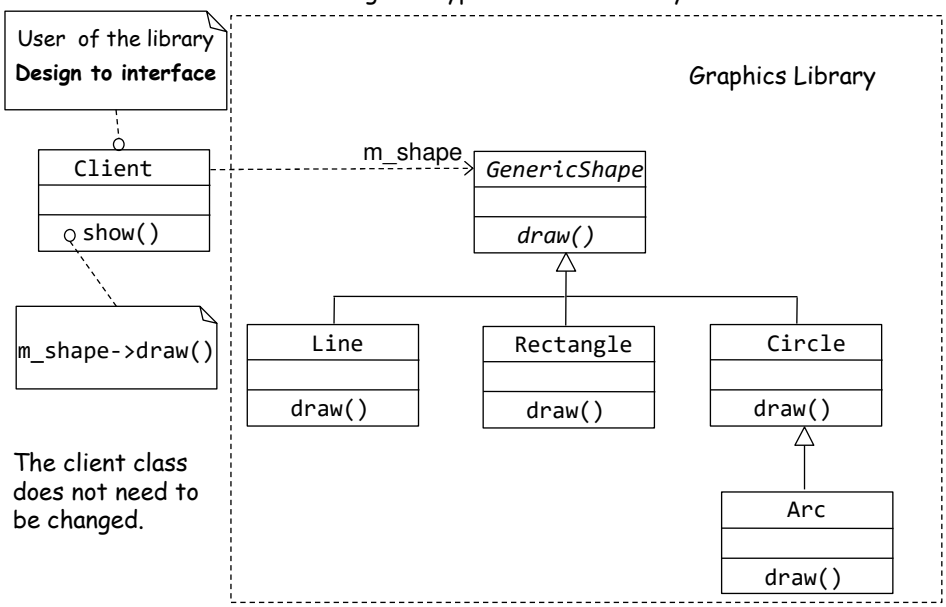
```
// A client (user) class that uses the Shape library
// Designed to interface (Generic Shape, not to Line, Rectangle, etc.)

class Client{
  GenericShape *m_shape;      // The pointer Can point to different shapes
                              // Design to interface
public:
 Client (GenericShape * inputShape): m_shape {inputShape} // Constructor
   {};                                // initial shape

 void setShape(GenericShape * inputShape)
  {
     m_shape = inputShape;           // change the shape in run-time
  }

  void show() const
   {                          // Which draw function will be called?
       m_shape->draw();       // It is unknown at compile-time
   }                          // Polymorphism

};
```

```
/** Test program **/
int main()
{
  // Shape objects
  Circle *circle1 = new Circle{ 100, 100, 20 };
  Rectangle *rectangle1= new Rectangle{ 30, 50, 250, 140 };
  Circle *circle2 = new Circle{ 300, 170, 50 };

  // Client object
  Client testClient{ rectangle1 };   // Connect to rectangle1
  testClient.show();                 // get a service from the shape

  testClient.setShape(circle2);      // change the shape to circle2
  testClient.show();                 // get a service from the shape

  testClient.setShape(circle1);
  testClient.show();
        ……
```

See Example generic_shape.cpp

The of the `Client`'s behavior (`show()`) changes in run time.

Adding new type Arc to the library:



The client class does not need to be changed.

Example: Third-Party (External) Tax Calculators in the NextGen System
(Polymorphism, Pure Fabrication, Indirection)

Problems:

Multiple external third-party tax calculators must be supported in the NextGen POS application.

- The system needs to integrate with **different calculators** according to some conditions.

  For example, if the total is above 500TL, it uses the external "Tax Master" program; otherwise, it uses the "Good As Gold" program.
- Each tax calculator has a **different interface**.

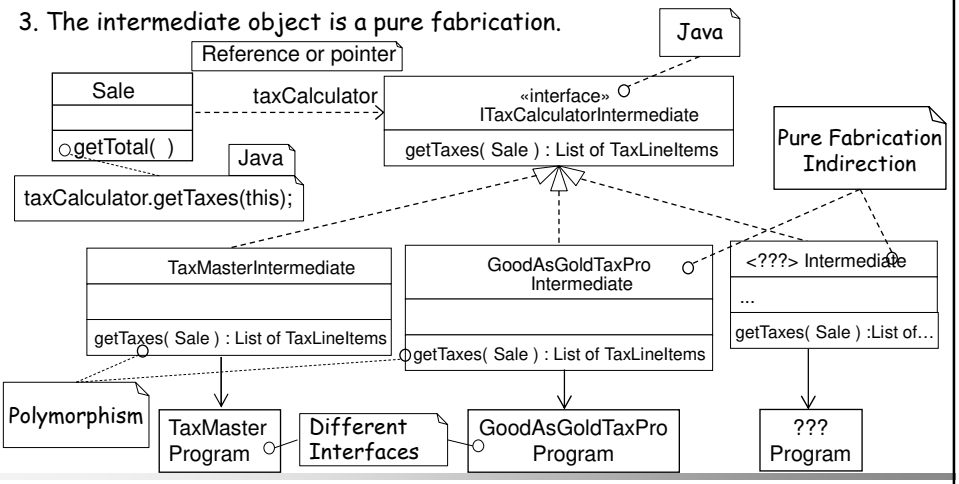
  One product may support a raw TCP socket protocol, another offers a SOAP interface, and a third offers a Java RMI interface.
- In the future, a new calculator program may be integrated into the system, or an existing calculator may be removed.

Actually, the Sale class is responsible for calculating the total and therefore needs the tax.

However, we want to keep our system (Sale) independent from the **varying** external tax calculators.

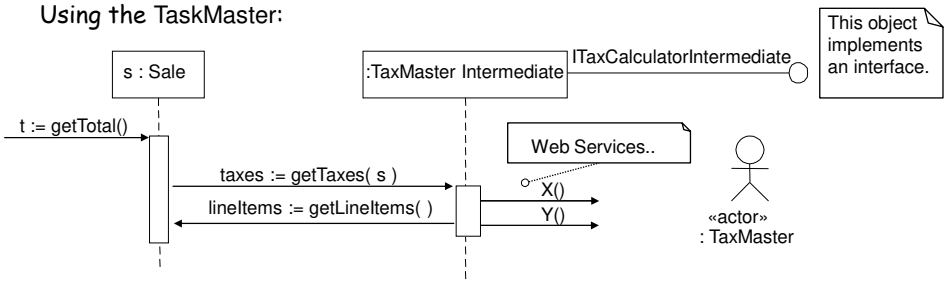
Solution: Polymorphism, Pure Fabrication, Indirection

1. We do not want to couple the Sale to tax calculators directly.
We will put intermediate objects (indirection) between Sale and calculators.
2. The intermediate objects will include a polymorphic method that the Sale calls. This method will then refer to the external calculator.
3. The intermediate object is a pure fabrication.

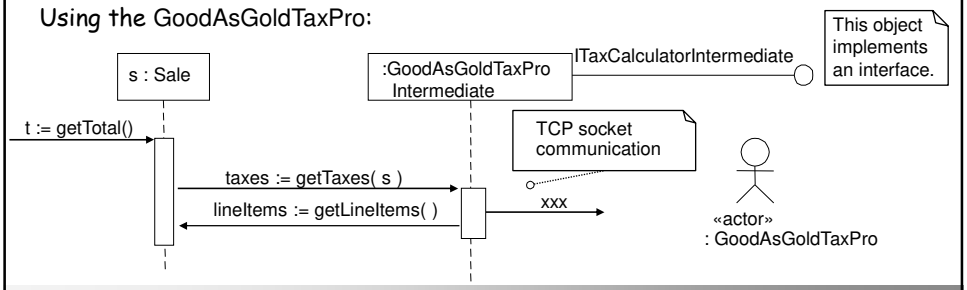


Indirect access to the calculator:

Using the TaskMaster:



Using the GoodAsGoldTaxPro:



Visibility from Intermediate to the Sale:

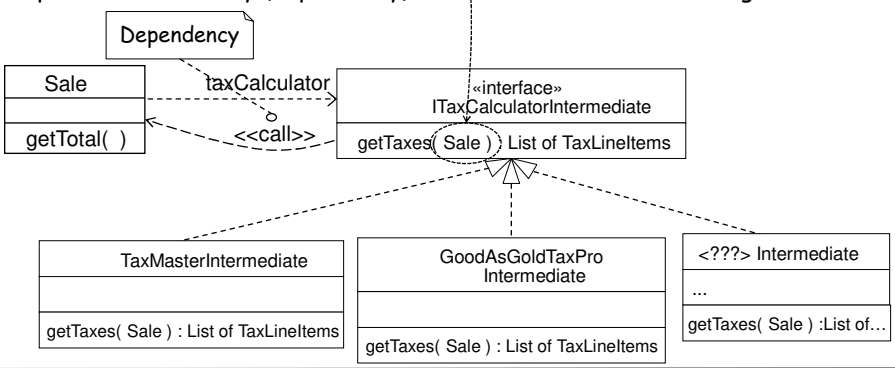
Each getTaxes method takes the Sale object (s) as a parameter so that the calculator can analyze the sale.

The Intermediate may call some methods of the Sale to get the necessary information to calculate taxes.

For example, the total, list of line items, customer, etc.

Example: `lineItems := getLineItems();` (7.23)

This parameter visibility (dependency) can be shown on the UML diagram.



Connection between Sale and Intermediate Objects:

- Where is the decision logic to select the right tax calculator?
- Which object should create the appropriate intermediate object and pass its address to the Sale object?
- Who will establish the visibility from Sale to the appropriate Intermediate?

If Sale decides which intermediate object should be created, it must be aware of (know) all external programs and their intermediate objects.

It means coupling between Sale and external systems.

If an external program or its creation logic changes, we must also change the Sale class.

To isolate the Sale from the external programs, we need another artificial object (Factory) to fulfill this responsibility (slide 7.26).

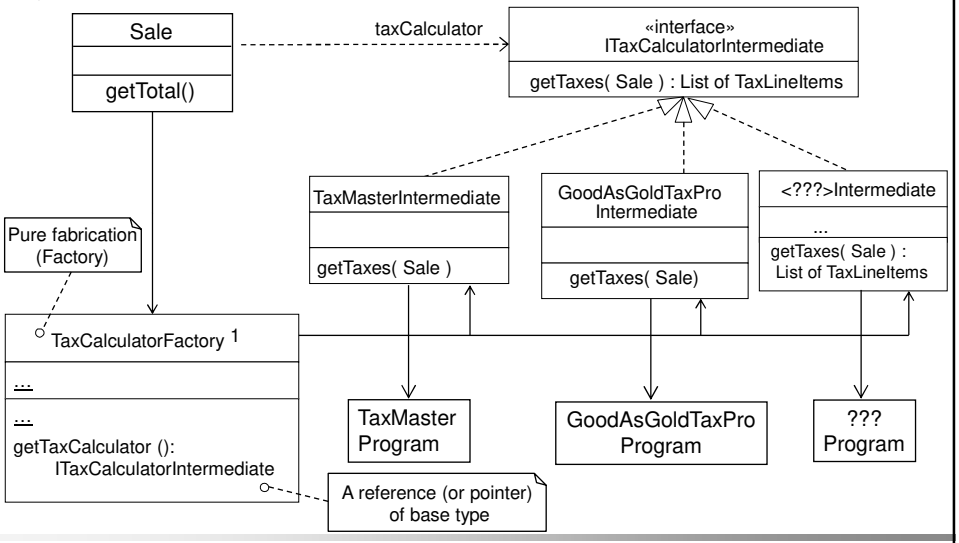
We will discuss the details of the **Factory** GoF design pattern in Chapter 8.

The intermediate objects (providing Polymorphism, Pure Fabrication, Indirection) used in this example are called in software-world "**adapters**".

Adapter is a GoF design pattern.
We will discuss it in chapter 8.

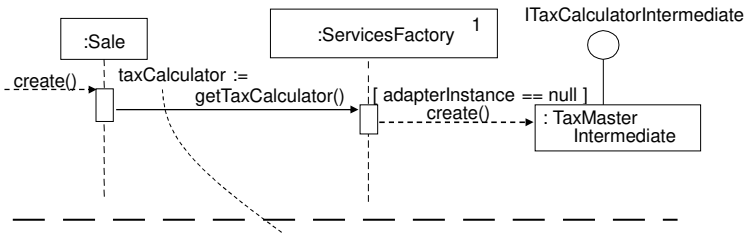
Connection between Sale and Intermediate Objects (cont'd):

The artificial object **TaxCalculatorFactory** creates the necessary intermediate object and returns its address to the Sale.

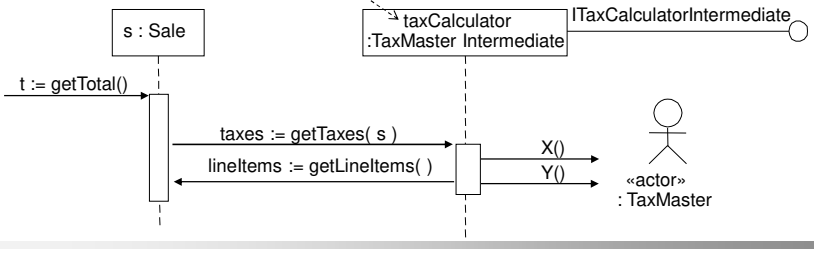


Connection between Sale and Intermediate Objects (cont'd):

Sale gets the address of the intermediate object from the Factory.



Using the TaskMasterIntermediate pointed by the pointer (reference) taxCalculator:



Protected Variations (GRASP)

"Protected Variations" is a fundamental principle of software design.

Almost every software or architectural design method is a specialization of it.

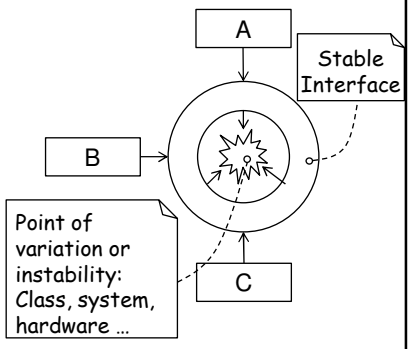
Problem:

How should we design objects, subsystems, and systems so that the variations or instabilities in these elements do not have an undesirable impact on other elements?

Solution:

Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Here, the term "interface" is used in the broadest sense of an access view; it does not only mean something like a Java interface.



Example: The prior external tax calculator problem (see 7.21)

Points of instability or variation:

- Different interfaces of external tax calculators.
- The POS system needs to be able to integrate with many existing tax calculator systems and also with future third-party calculators not yet in existence.

Solution:
By adding a level of *indirection*, an *interface*, and using *polymorphism* with various ITaxCalculatorIntermediate implementations, protection within the system from variations in external APIs is achieved.

Internal objects collaborate with a stable interface; the various adapter implementations hide the variations to the external systems. (See 7.22)

Discussion:

PV is a root principle motivating most of the mechanisms and patterns in programming and design to provide flexibility and protection from variations in data, behavior, hardware, software components, operating systems, and more.

Many design tricks such as encapsulation, polymorphism, data-driven designs, interfaces, virtual machines, configuration files, and operating systems are a specialization of Protected Variations.

The Law of Demeter (Don't Talk to Strangers Principle)

This principle is a special case of Protected Variations.

It means avoiding creating designs that send messages (or talk) to distant, indirect (stranger) objects.

Such designs are fragile to changes in the object structures.

The principle states that within a method, messages should only be sent to the following (familiar) 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.

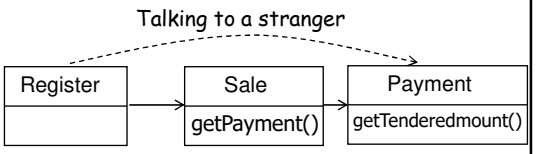
These are direct objects that are a client's "familiars".

Indirect objects are "strangers".

A client should talk to "familiars" and avoid talking to strangers.

Example: Talking to strangers

```
public class Register
{
   private Sale sale;
   public void slightlyfragileMethod()
   {
     Money total = sale.getTotal();          // OK: Because Sale is familiar
     Money amount = sale.getPayment().getTenderedAmount(); // msg to a stranger
     :
   }
```



sale.getPayment() returns the address of the Payment object

getTenderedAmount() belongs to Payment that is stranger to Register

This code traverses structural connections from a familiar object (the Sale) to a stranger object (the Payment) and then sends it a message.

General example: obj1.m1().m2().¨¨.mn();

The farther along a path the program traverses, the more fragile it is because the object structure (the connections) may change.

To solve the problem (providing protection against structural variations), a new public method can be added to a familiar object.

```
Money amount = sale. getTenderedAmountOfPayment();   // A new method to Sale
```