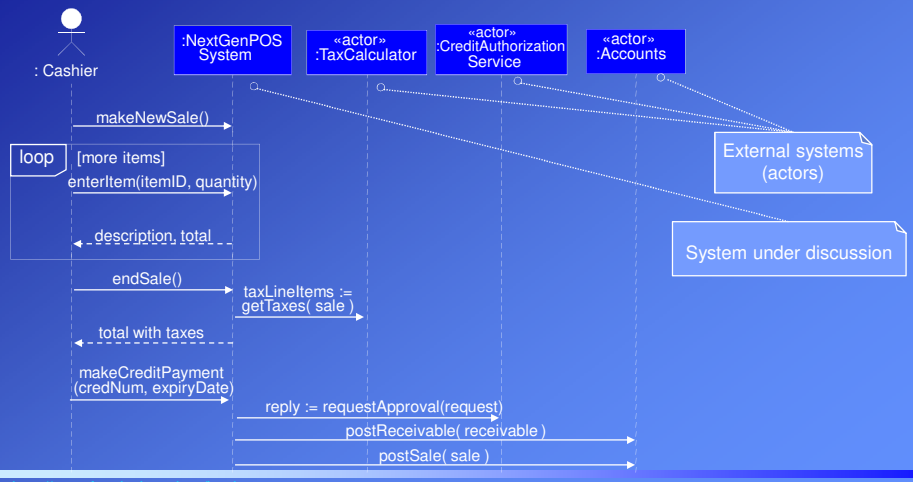
2nd Iteration, Other GRASP patterns

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:

- Collaboration with external systems such as Credit Card Authorization system, Accounting system 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 different parameters such as date, product, total of sale, customer type.

  Customers can have loyalty cards, which provide them some benefits.

Domain Analysis in the second iteration:

Analysis in the second iteration may take shorter because there are not so much new conceptual classes.

Old domain models from the first iteration are not used.

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

GRASP 2

Previously we have seen 5 GRASP patterns.

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

Now we will discus the remaining 4 GRASP patterns:

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

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

Polymorphism (GRASP)

Remember, polymorphism is one of the fundamental properties of the object oriented programming. (What is polymorphism in programming level?)

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

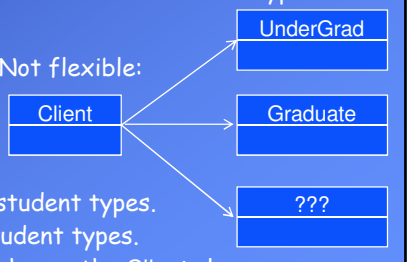
If a program is designed using if-then-else or switch-case statements, then when a new variation or a type (class) arises, it requires modification of the case logic often in many places.

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

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 the 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 an 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 the super type (base) of other objects.

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.

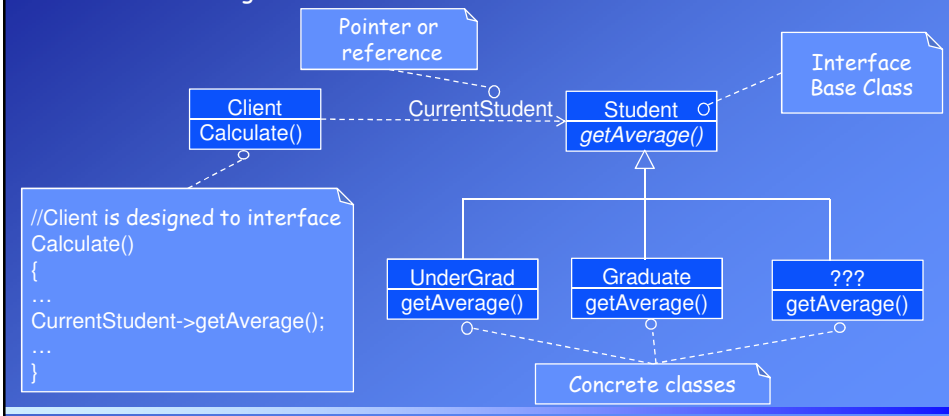
OOP Principle: Design to interface and not concrete classes.
Program to an interface, not an implementation.

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

One of the important 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.

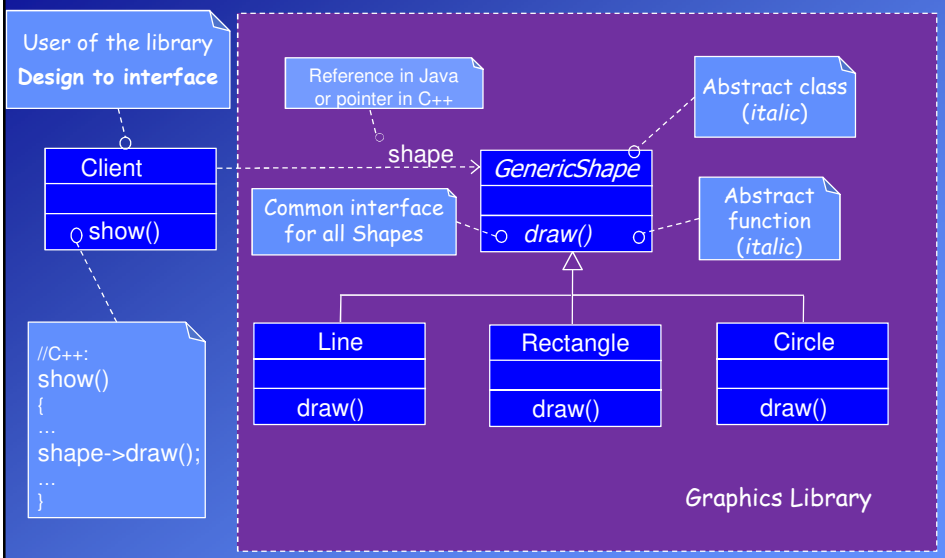
It is possible to add new shapes in to the library (or may be remove some of them).

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

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
};

class Line:public GenericShape{              // Line class
 private:
   int x1, y1, x2, y2;                       // Coordinates of line
 public:
   Line(int x_in,int y_in,int x2_in,int y2_in):  // Constructor
   void draw() const;                        // virtual draw function of line
};

Class Rectangle:public GenericShape{
......
void draw() const;                           // virtual draw of rectangle
.....

class Circle:public GenericShape{            // Circle class
 private:
    int centerX, centerY, radius;
 public:
    Circle(int x_cen,int y_cen,int r);       // Constructor
    void draw() const;                       // virtual draw of circle
};
```

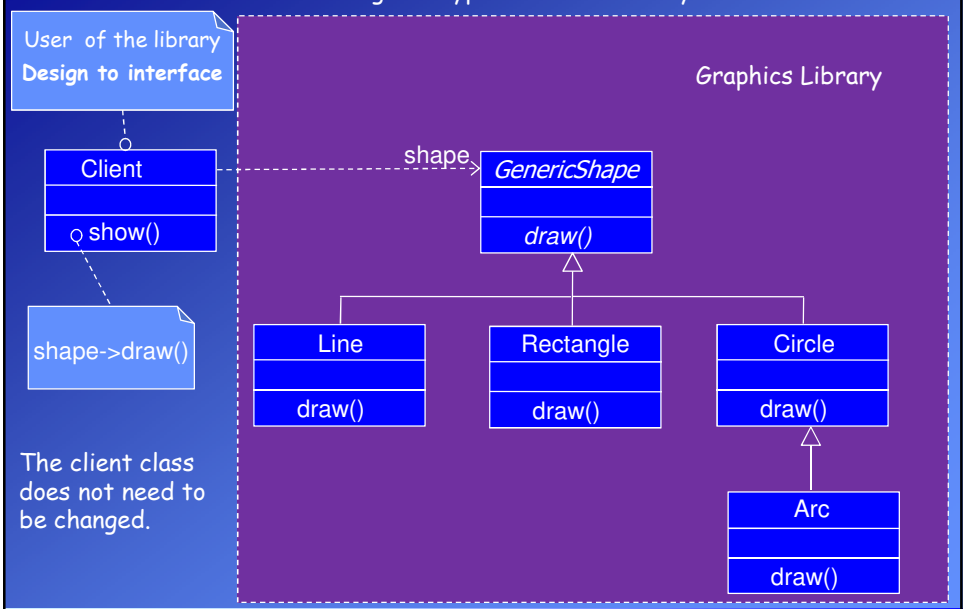


```
/* A client (user) class that uses the Shape library, Designed to interface */

class Client{
  GenericShape *shape;                       // Can point to different shapes
                                             // Design to interface
public:

 Client (GenericShape * inputShape): shape(inputShape)    // Constructor
   {};                                       // initial shape

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

  void show() const
  {                                          // Which draw function will be called?
     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 circle 2
         testClient.show();                    // get a service from the shape

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

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

Adding new type Arc to the library:



Pure Fabrication (GRASP)

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

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

For example; Sale, Payment, Student, Course, 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 have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?

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 **fabrication** of the imagination.

The responsibilities assigned to this fabrication must support high cohesion and low coupling, so that the design of the fabrication 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."

But 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.

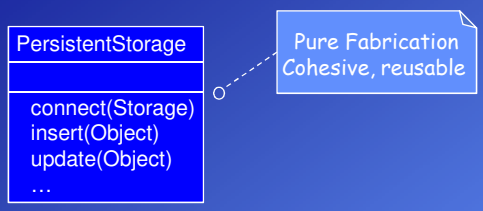
  The reusability potential of this Sale class is low. In another project Sale may not be saved in a database.
- Saving objects in a relational database is a very general task for which many classes need support.

  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 a Pure Fabrication.



Advantages:

- The Sale remains well-designed, with high cohesion and low coupling.
- The PersistentStorage class is itself relatively cohesive, having 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 is related 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 low representational gap.

But sometimes (because of cohesion, coupling, reusability), we group together 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 new to object design and more familiar with procedural (imperative) programming.

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 future.

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

Problem:

Where 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 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 class PersistentStorage class is also an example of assigning responsibilities to support Indirection.

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

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

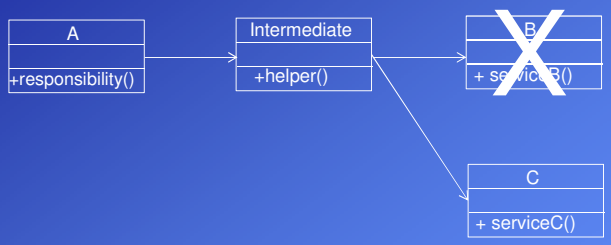
Example: Intermediate

Problems:

A class A gets services from a class B.

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

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



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

Problems:

In the NextGen POS application, there are multiple external third-party tax calculators that must be supported.

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

For example; if total is above 500TL it uses the external "Tax Master" program, otherwise "Good As Gold" program.

Each tax calculator has a **different interface**.

One product may support a raw TCP socket protocol, another may offer a SOAP interface, and a third may offer a Java RMI interface.

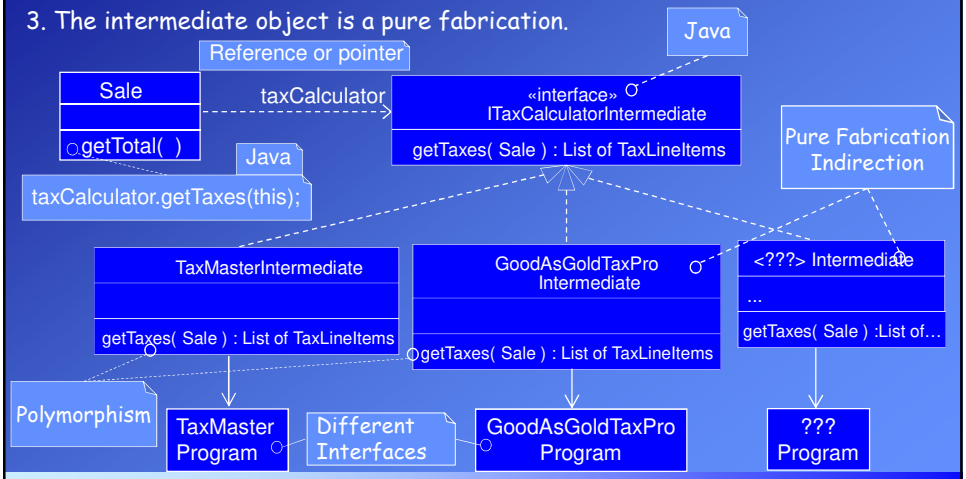
In the future, a new calculator program may be integrated into the system or an existing calculator may be removed.

Actually Sale class is responsible to calculate the total and therefore needs the tax.

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

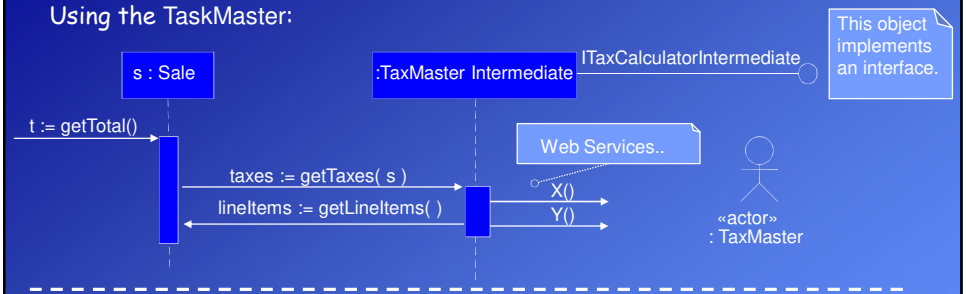
Solution: Polymorphism, Pure Fabrication, Indirection

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

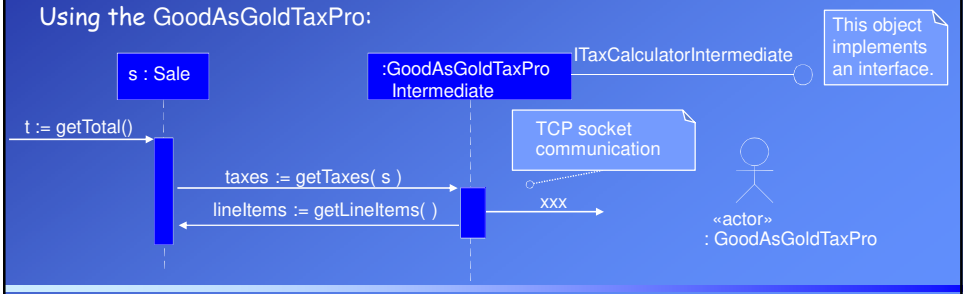


Indirect access to the calculator:

Using the TaskMaster:



Using the GoodAsGoldTaxPro:



Visibility from Intermediate to the Sale:

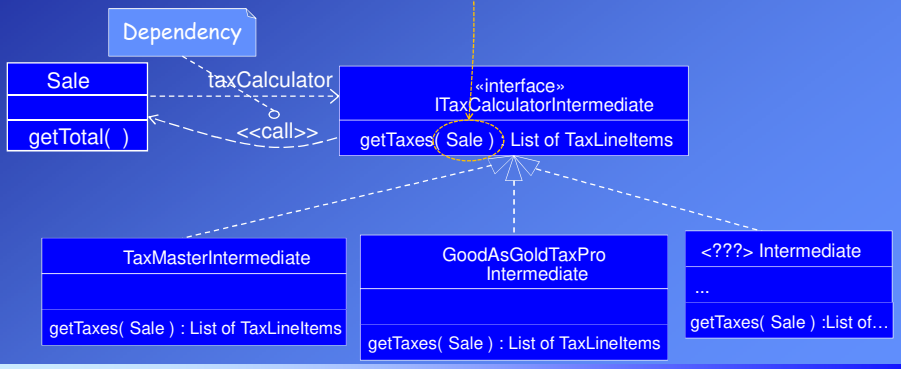
Each getTaxes method takes the Sale object (s) as a parameter, so that the calculator can analyze the sale.

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

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

Example: lineItems := getLineItems( ); (7.22)

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 different external programs and their intermediate objects.

It means coupling between Sale and external systems.

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

To isolate the Sale from the external programs we need another artificial object to fulfill this responsibility (slide 7.24).

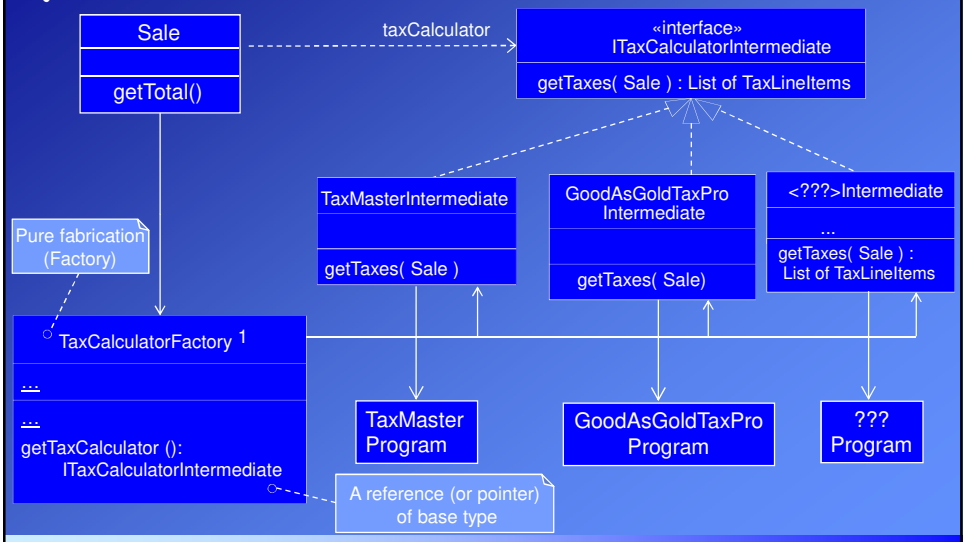
We will discuss the **Factory** GoF design pattern in chapter 8.

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.

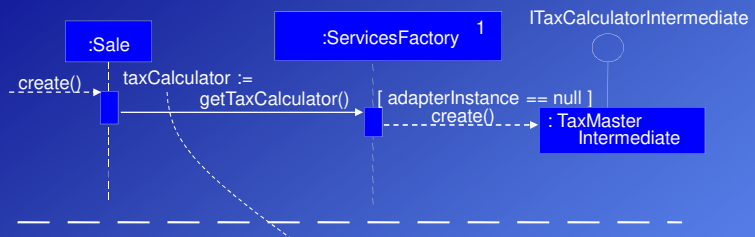
Connection between Sale and Intermediate Objects (cont'd):

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

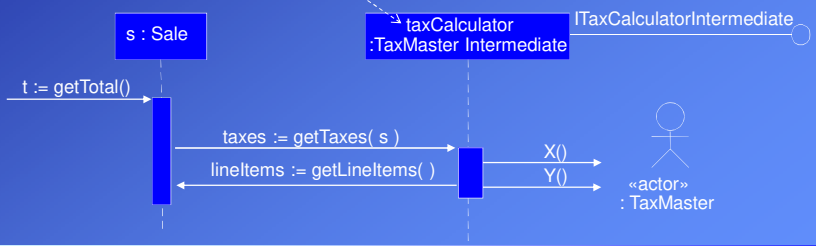


Connection between Sale and Intermediate Objects (cont'd):

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 very important and fundamental principle of software design.

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

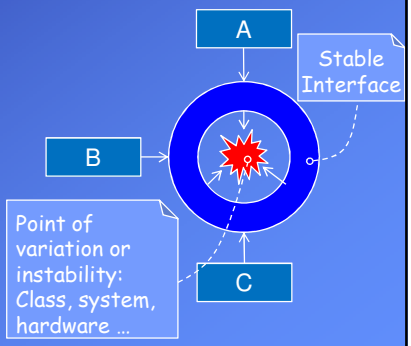
Problem:

How to 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 7.20)

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 7.21)

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, operating systems are 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 to avoid creating designs that send messages (or talk) to distant, indirect (stranger) objects.

Such designs are fragile with respect 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.

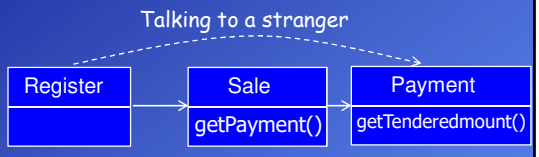
These are direct objects that are a client's "familiars".

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
```