Agile Methods: Perspective on Design

- Agile methods have an approach to software design that includes
  - identifying aspects of bad design
  - avoiding those aspects via a set of principles

Good Design Principles(SOLID)

1. The Single Responsibility Principle(SRP)
2. The Open-Closed Principle(OCP)
3. The Liskov Substitution Principle(LSP)
4. The Interface Segregation Principle(ISP)
5. The Dependency Inversion Principle(DIP)

1.Single Responsibility Principle

- **A class should have only one reason to change(responsibility)**
  - If a class has more than one responsibility, changes to one can impact the other
  - If a class has a single responsibility, you can limit the impact of change with respect to that responsibility to this one class
- Example
  - Class Rectangle with draw() and area() methods
    - draw() is only used by GUI apps, separate these responsibilities into different classes

Figure 8-1. More than one responsibility

[View full size image]

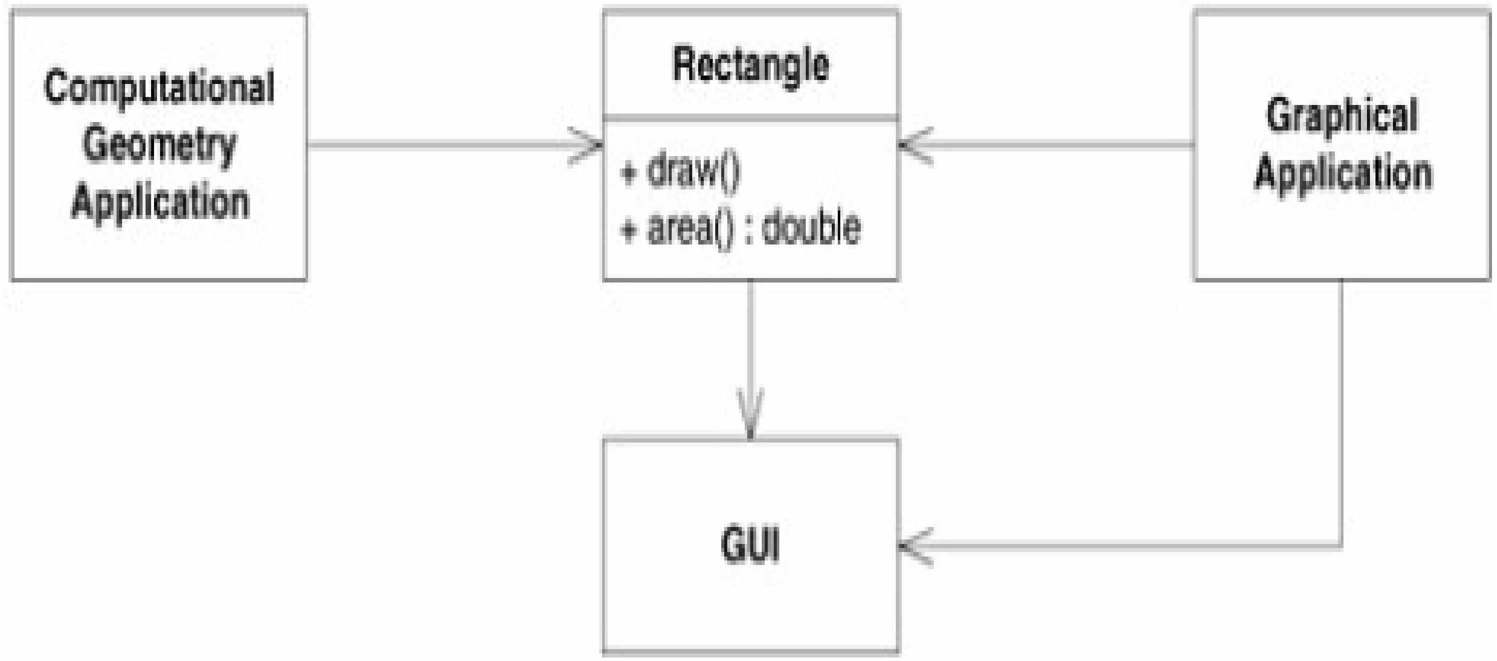
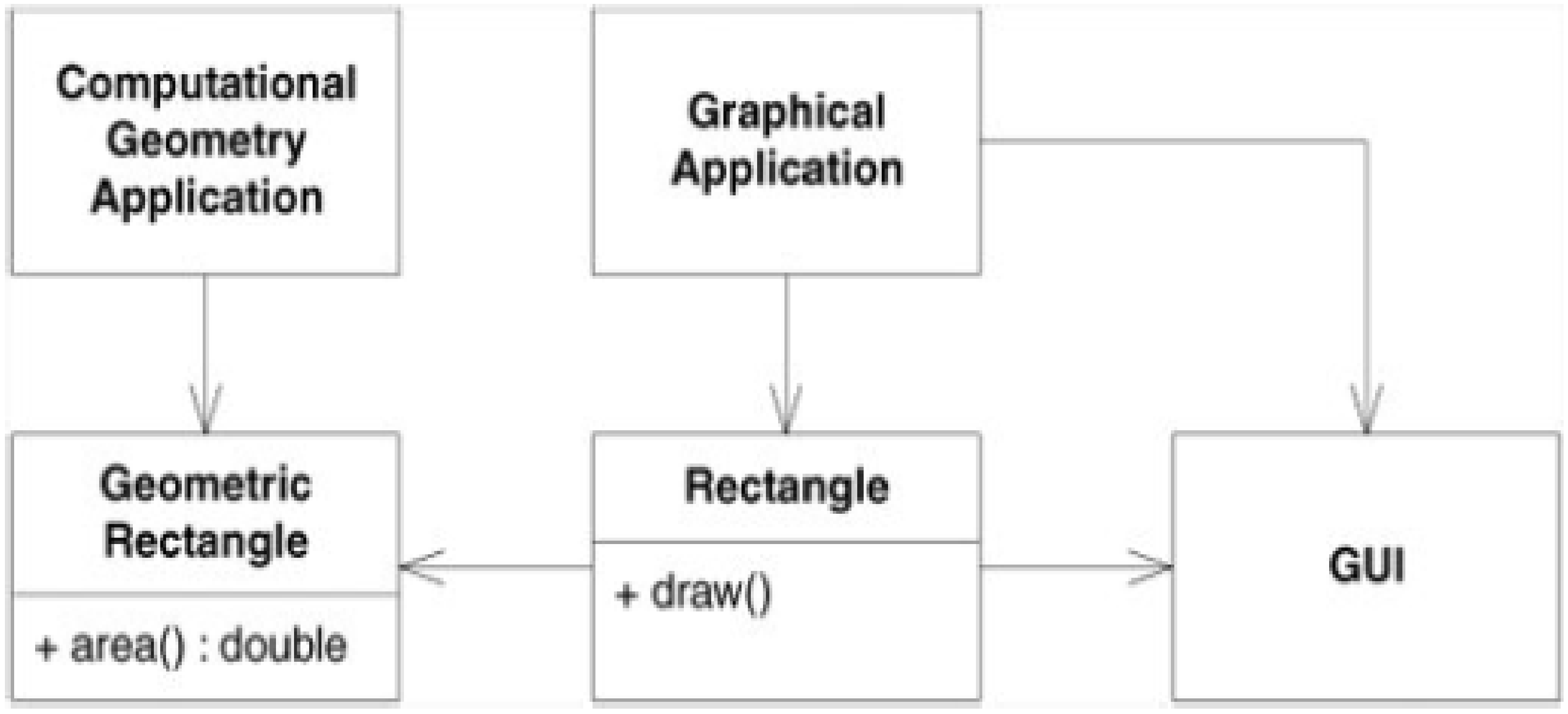


Figure 8-2. Separated responsibilities



Cohesion and Coupling

- Cohesion: how strongly-related and focused are the various responsibilities of a module

- Coupling: the degree to which each program module relies on each one of the other modules

Strive for low coupling and high cohesion!

- Cohesion is an indication of **relative functional strength of a module**.
- Cohesion is a qualitative indication of the degree to which a module focuses on just one thing.

- Coupling is an indication of **relative interdependence among modules.**
- Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

- A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.

- Coupling is an indication of interconnection among modules in a software structure.
- That is, coupling depends on the interface complexity between modules.

Types of Coupling

- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling

- **Content coupling/Pathological Coupling** occurs when one module modifies the data values or instructions in another module.

  That is, when a module uses/alters data in another.

- **Common/Global-data Coupling** the modules are bound together by global data structures. **.**

  Two module communicating via **global** data.

- **Control coupling** involves passing control flags between modules so that one module controls the sequence of processing steps in another module.

  That is communicating with a control flag.

Stamp/Data-structure Coupling is similar to common coupling except that global data items are shared selectively among the routines that require the data.

The data structure holds **more** information than the recipient needs.

Communicating via a data structure **passed** as a parameter.

- **Data coupling** involves the use of parameter lists to pass items between the routines
- Communicating via parameter passing. The parameters passed are only those that the recipient needs.

- The most desirable form of coupling between modules is a combination of stamp and data coupling.

Types of cohesion

- Coincidental cohesion
- Logical cohesion
- Temporal cohesion
- Communicational cohesion
- Sequential cohesion
- Functional cohesion
- Procedural cohesion
- Informational cohesion

- **Coincidental cohesion** occurs when the elements within a module has no apparent relationship to one another.
- Module elements are unrelated.

- **Logical cohesion** :In this the elements within the modules perform similar activities which are executed from outside the module.
- i.e. by a flag that selects operation to perform.

- **Temporal cohesion** : In this the elements within the modules contain unrelated activities that can be carried out at the same time.

- **Communicational cohesion** is one in which the elements within the modules perform different functions, yet each function references the same input or output information.
- unrelated operations need same data or input

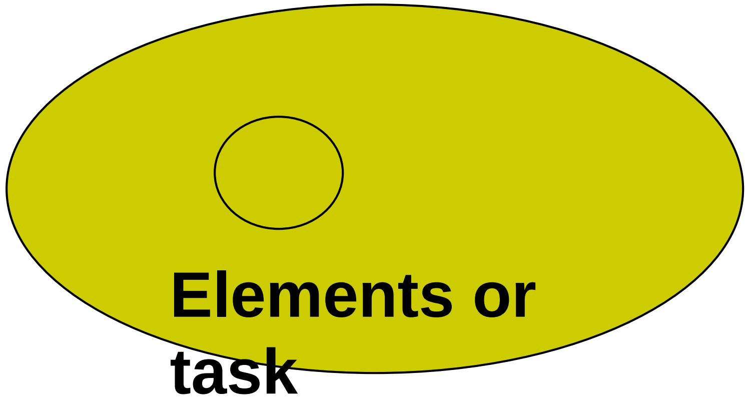
- **Sequential cohesion** of elements occurs when the output of one element is the input for the next element.
- operations on same data in significant order; output from one function is input to next

- **Functional cohesion** : In this elements within the modules contribute to the execution of one and only one problem related task.
- All elements contribute to a single, well-defined task, i.e. a function that performs exactly one operation.

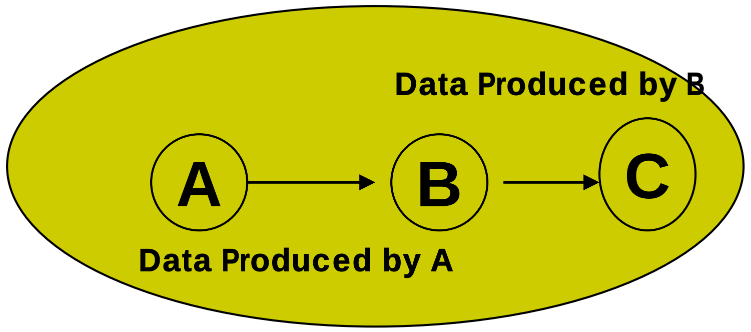
- **Procedural cohesion**: In this ,the elements within the modules are involved in different and possibly unrelated activities**.**
- Elements involved in different but sequential activities, each on different data.

- **Informational cohesion** of elements in a module occurs when the module contains a complex data structure and several routines to manipulate the data structure.

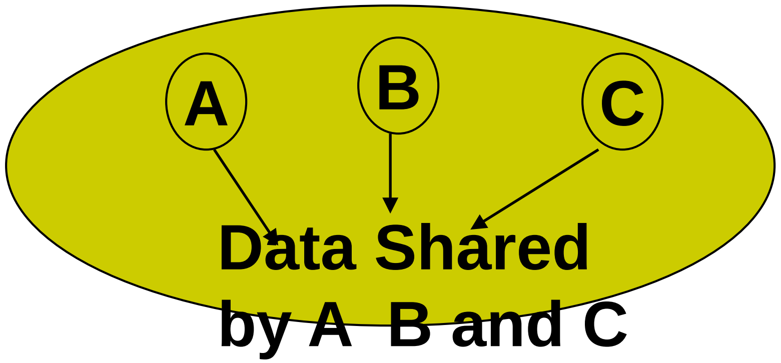
- a module performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure



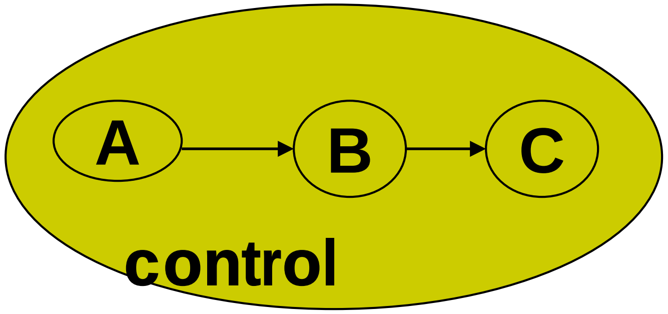
Functional cohesion



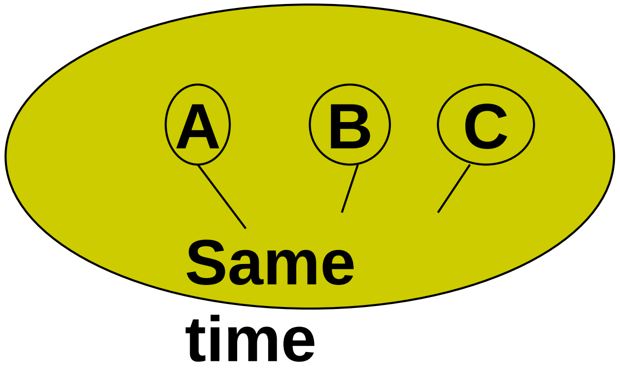
Sequential cohesion



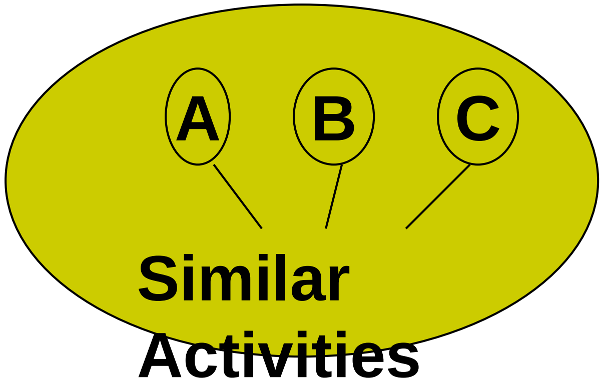
Communicational
Cohesion



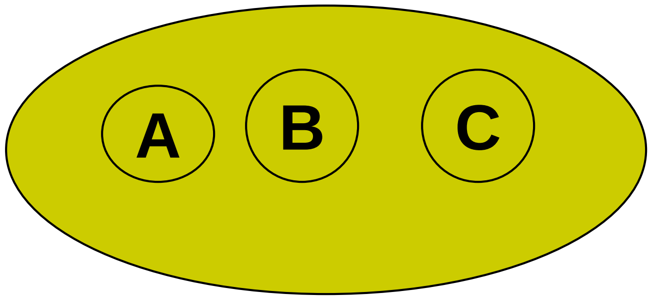
Procedural
Cohesion



Temporal Cohesion



Logical Cohesion



Coincidental Cohesion

Responsibilities are Axes of Change

- More responsibilities == More likelihood of change
- *The more classes a change affects, the more likely the change will introduce errors.*

2. Open-Closed Principle

- **The Open / Closed Principle states that a class should be opened to extension but closed to modification**

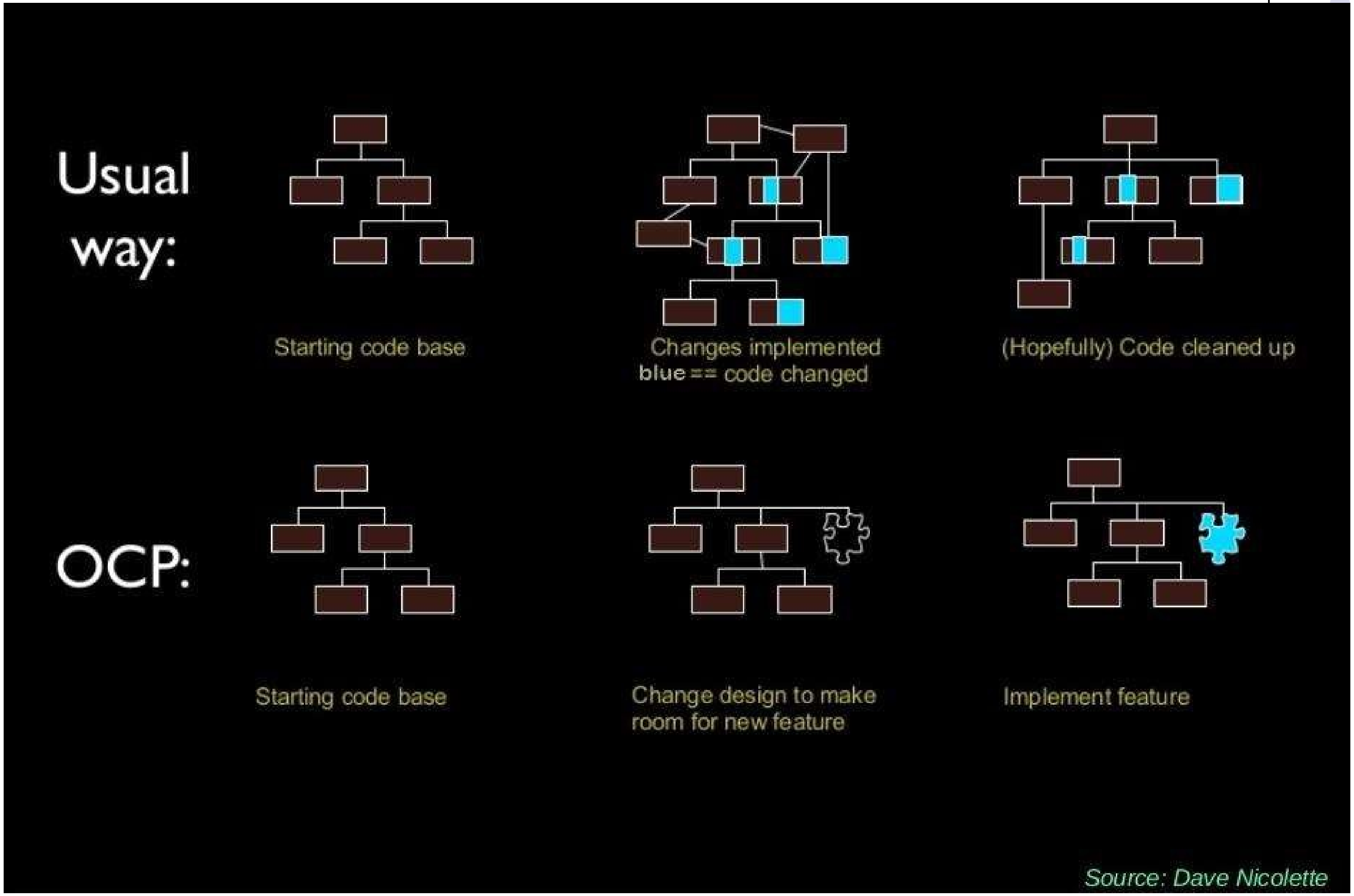
  - Allows clients to code the class without fear of later changes

Open / Closed Principle

"Software entities (classes, modules, methods etc) should be open for extension but closed for modification"

That is such an entity can allow its behaviour to be changed without altering its source code.

Open Closed Principle



Modules that conform to OCP have two primary attributes.

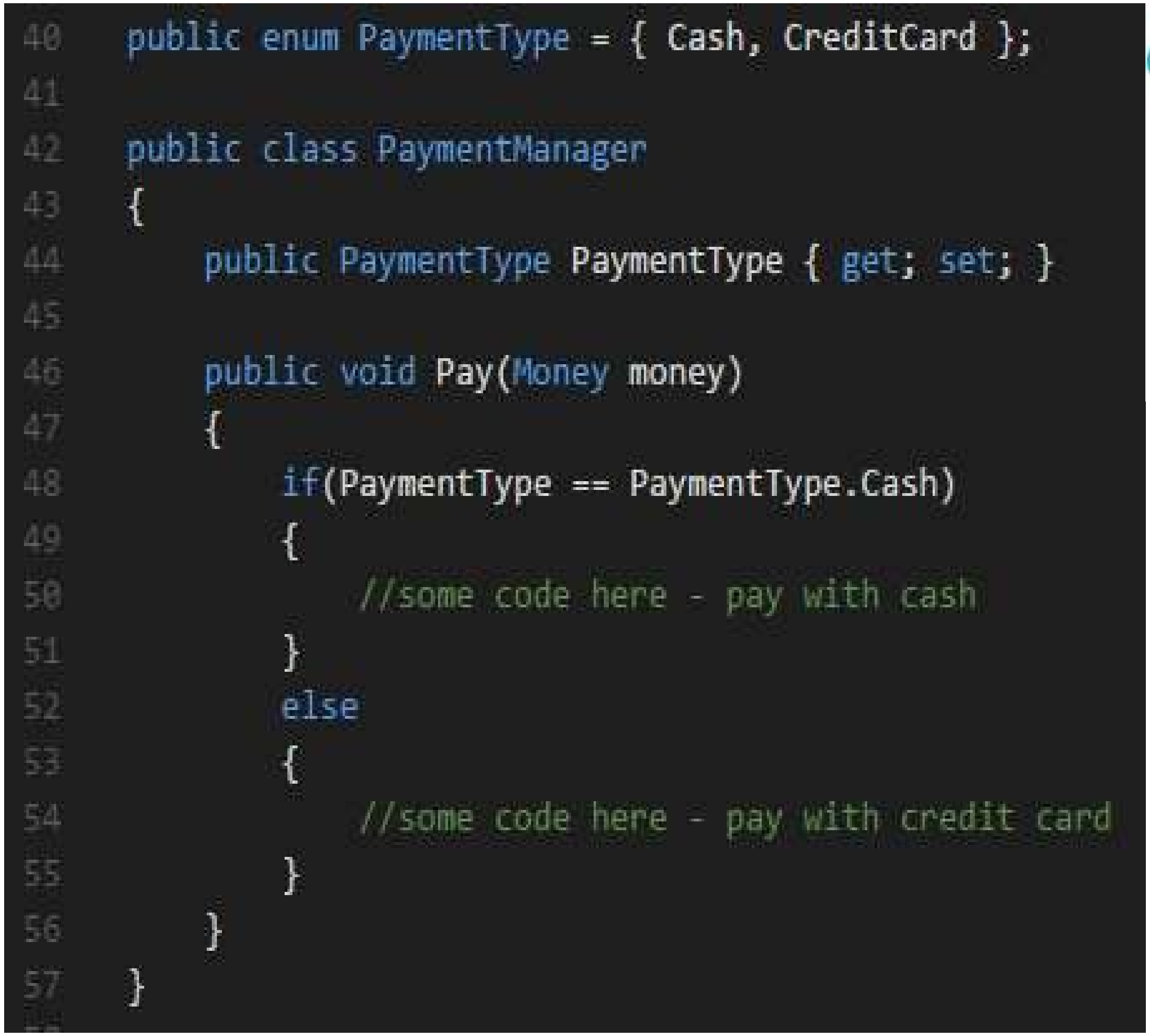
1. Open for extension.

- Behaviour of the module can be extended.
- As the requirements of the application change, we can extend the module with new behaviours that satisfy those changes.

2. Closed for modification.

- Extending the behaviour of a module does not result in changes to the source, or binary, code of the module.

Open / Closed Principle



```
public enum PaymentType = { Cash, CreditCard };

public class PaymentManager
{
    public PaymentType PaymentType { get; set; }

    public void Pay(Money money)
    {
        if(PaymentType == PaymentType.Cash)
        {
            //some code here - pay with cash
        }
        else
        {
            //some code here - pay with credit card
        }
    }
}
```

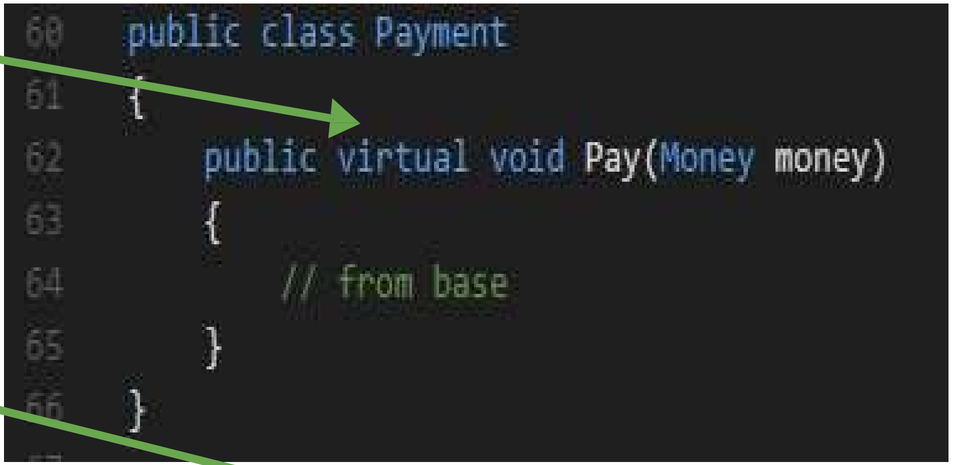
Humm…and if I need to add a new payment type?

You need to modificate this class.

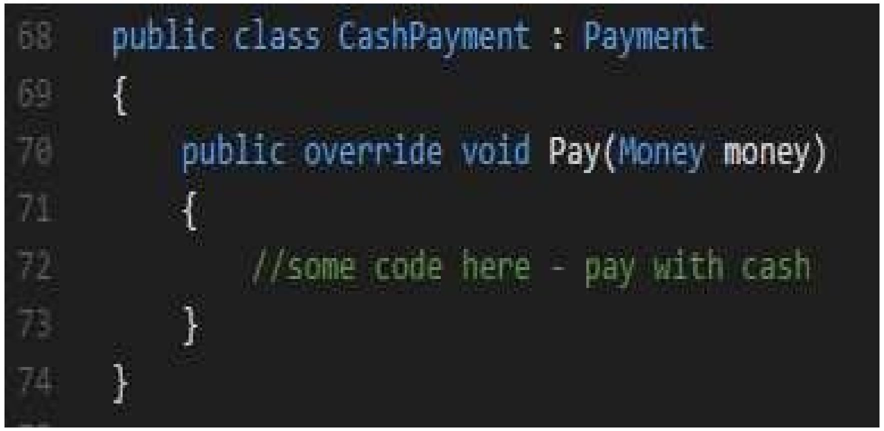
Open / Closed Principle

open for extension

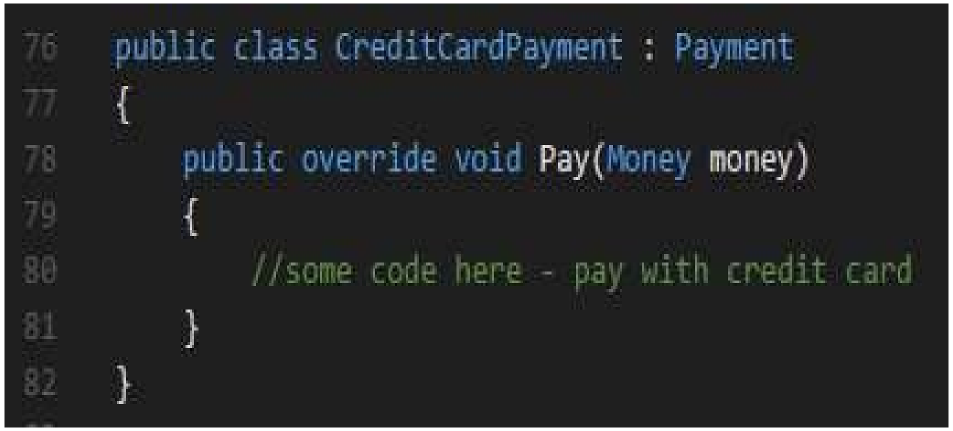
close for modification



```
public class Payment
{
    public virtual void Pay(Money money)
    {
        // from base
    }
}
```



```
public class CashPayment : Payment
{
    public override void Pay(Money money)
    {
        //some code here - pay with cash
    }
}
```



```
public class CreditCardPayment : Payment
{
    public override void Pay(Money money)
    {
        //some code here - pay with credit card
    }
}
```

- Many changes are involved when a new functionality is added to an application.
- Those changes in the existing code should be minimized, since it's assumed that the existing code is already unit tested and changes in already written code might affect the existing functionality in an unwanted manner.

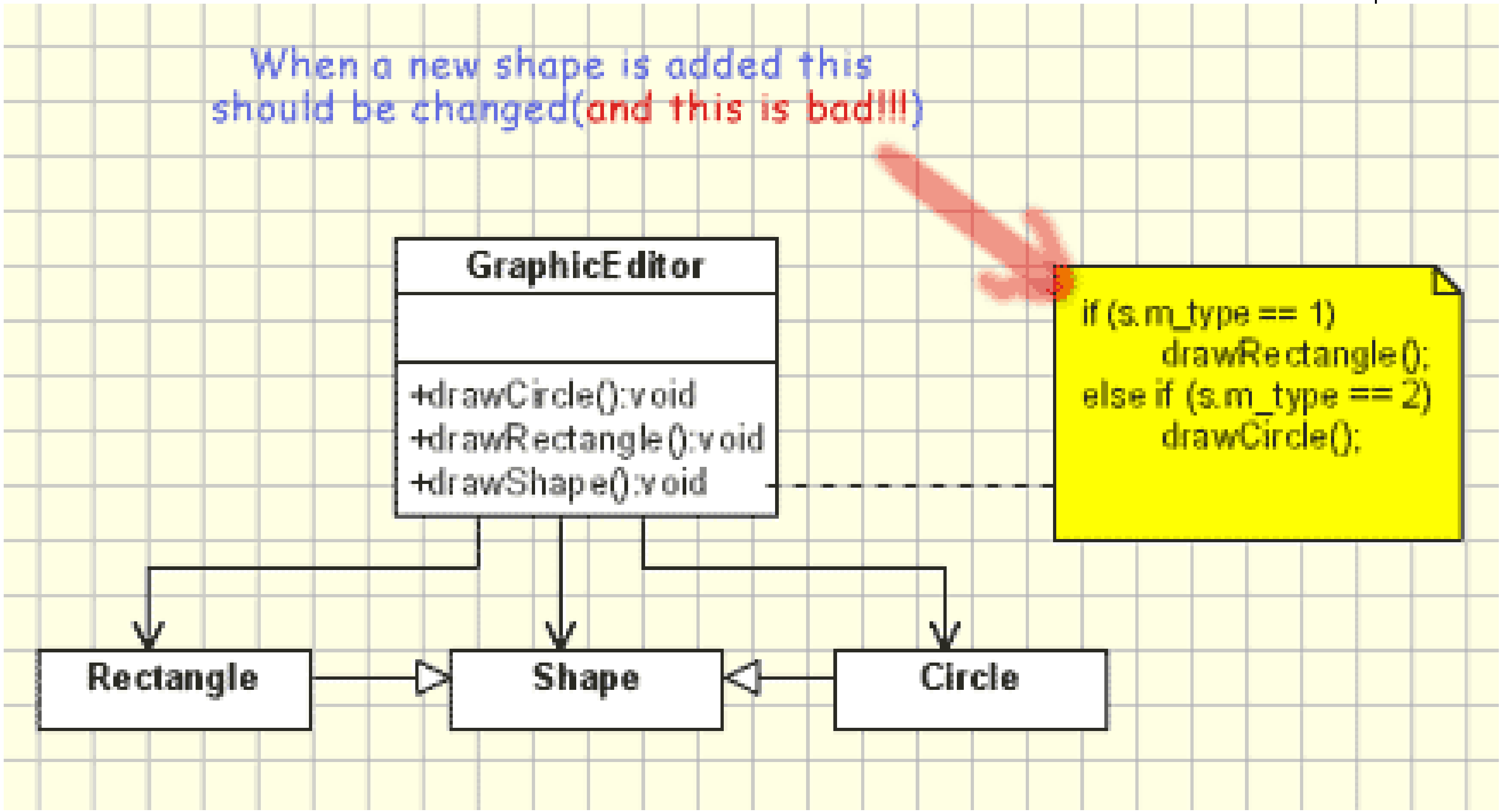
- The **Open Close Principle** states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code.

- The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

Example

- An example which implements a graphic editor which handles the drawing of different shapes.
- It's obviously that it does not follow the Open Close Principle since the GraphicEditor class has to be modified for every new shape class that has to be added.

```
// Open-Close Principle - Bad example
 class GraphicEditor {

         public void drawShape(Shape s) {
                 if (s.m_type==1)
                         drawRectangle(s);
                 else if (s.m_type==2)
                         drawCircle(s);
         }
         public void drawCircle(Circle r) {....}
         public void drawRectangle(Rectangle r) {....}
 }

 class Shape {
         int m_type;
 }

 class Rectangle extends Shape {
         Rectangle() {
                 super.m_type=1;
         }
 }

 class Circle extends Shape {
         Circle() {
                 super.m_type=2;
         }
```



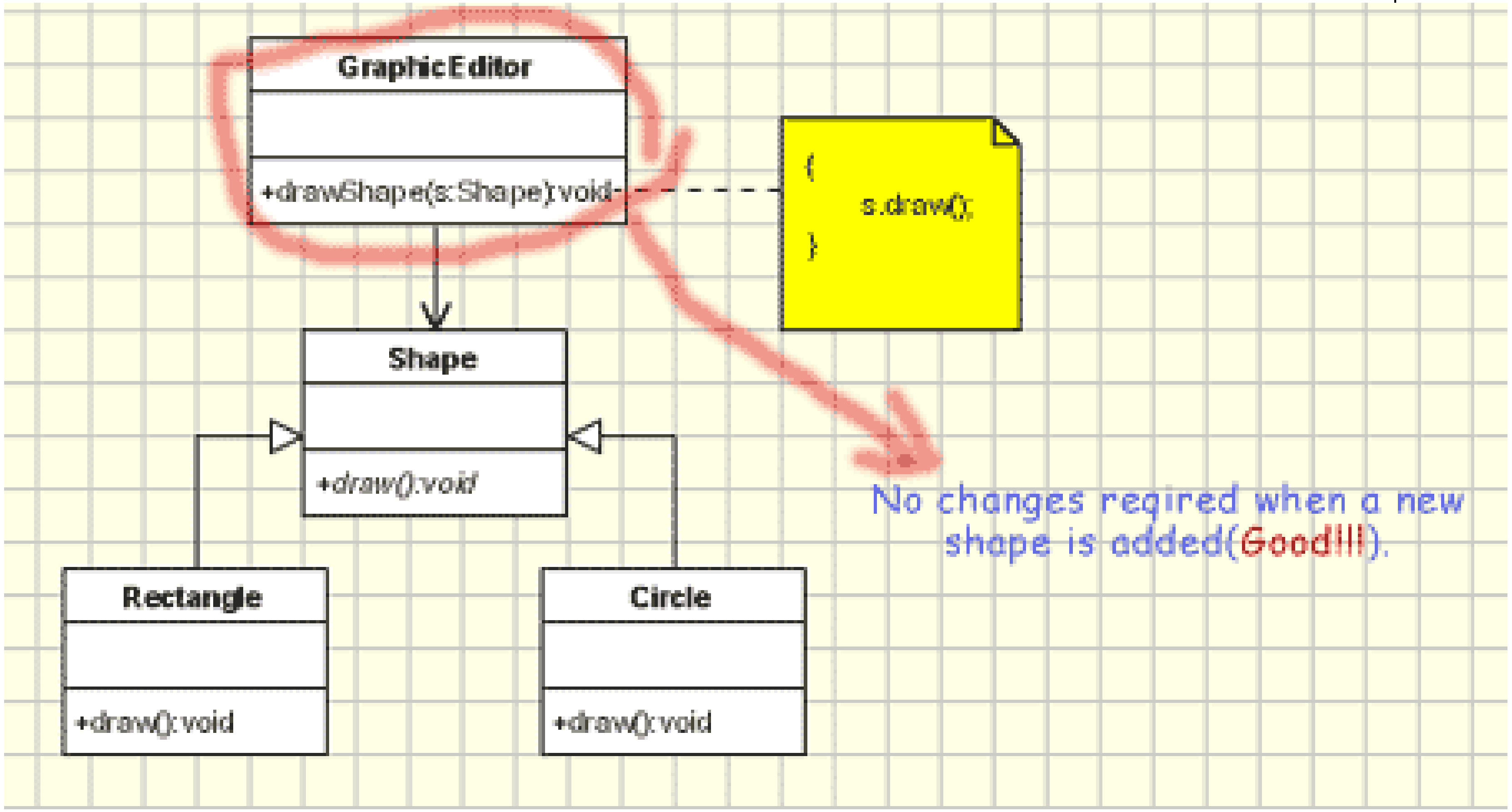
```
if (s.m_type == 1)
        drawRectangle();
else if (s.m_type == 2)
        drawCircle();
```

- There are several disadvantages:

- For each new shape added the unit testing of the GraphicEditor should be redone.
- When a new type of shape is added the time for adding it will be high since the developer who add it should understand the logic of the GraphicEditor.
- Adding a new shape might affect the existing functionality in an undesired way, even if the new shape works perfectly

- In the new design we use abstract draw() method in GraphicEditor for drawing objects, while moving the implementation in the concrete shape objects.
- Using the Open Close Principle the problems from the previous design are avoided, because GraphicEditor is not changed when a new shape class is added:

- No unit testing required.
- No need to understand the sourcecode from GraphicEditor.
- Since the drawing code is moved to the concrete shape classes, it's a reduced risk to affect old functionallity when new functionallity is added.


GraphicEditor
+drawShape(s:Shape):void
{
s.draw();
}
Shape
+draw():void
Rectangle
+draw():void
Circle
+draw():void
No changes reqired when a new shape is added(Good!!!).

```
// Open-Close Principle - Good example
 class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
 }

 class Shape {
        abstract void draw();
 }

 class Rectangle extends Shape  {
        public void draw() {
                // draw the rectangle
        }
 }
```

- Benefits
  - Flexibility
  - Reusability
  - maintainability

3.Liskov Substitution Principle

- *States that subtypes must be substitutable for their base types*

- Subclasses need to respect the behaviors defined by their superclasses
  - if they do, they can be used in any method that was expecting the superclass

- If a superclass method defines pre and post conditions
  - a subclass can only weaken the superclass pre-condition, and can only strengthen the superclass post-condition

- **What is Pre-Condition?**
- Pre-condition is a statement or set of statements that outline a condition that should be true when an action is called. The precondition statement indicates what must be true before the function is called.

- **Example:**
- To identify the square root of a number, the precondition is that the number should be greater than zero.

- **What is Post Condition?**
- Post Condition is a statement or set of statements describing the outcome of an action if true when the operation has completed its task.
- The Post Conditions statement indicates what will be true when the action finishes its task.

- **Example:**
- To identify the square root of a number, the precondition is that the number should be greater than zero. The POST Condition is that the square root of the number is displayed on the console.

- All the time we design a program module and we create some class hierarchies. Then we extend some classes creating some derived classes.

- We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

- Liskov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

- In short, this principle is just an extension of the Open Close Principle and it means that we must make sure that **new derived classes are extending the base classes without changing their behavior.**

4.Dependency-Inversion Principle

- High-level modules should not depend on low-level modules; Both should depend on abstractions.

- Abstractions should not depend on details. Details should depend on abstractions.

- Abstraction is the act of representing essential features without including the background details or explanations.

- The abstraction principle is used to reduce complexity and allow efficient design and implementation of complex software systems.

- When we design software applications we can consider the low level classes, the classes which implement basic and primary operations and high level classes, the classes which encapsulate complex logic.

- A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes.

- That is high level classes are defined in terms of others.
- This is not a flexible design.

- What happens if we need to replace a low level class?

- Let's take example of a **copy module** which reads characters from the keyboard and writes them to the printer device.
- The high level class containing the logic is the **Copy class**.
- The low level classes are **KeyboardReader** and **PrinterWriter.**

- In a bad design the high level class uses directly and depends heavily on the low level classes.

- In such a case if we want to change the design to direct the output to a new **FileWriter class** we have to make changes in the Copy class.

- In order to avoid such problems we can introduce an **abstraction layer** between high level classes and low level classes.

- Since the high level modules contain the complex logic they should not depend on the low level modules so the new abstraction layer should not be created based on low level modules.

- Low level modules are to be created based on the abstraction layer.

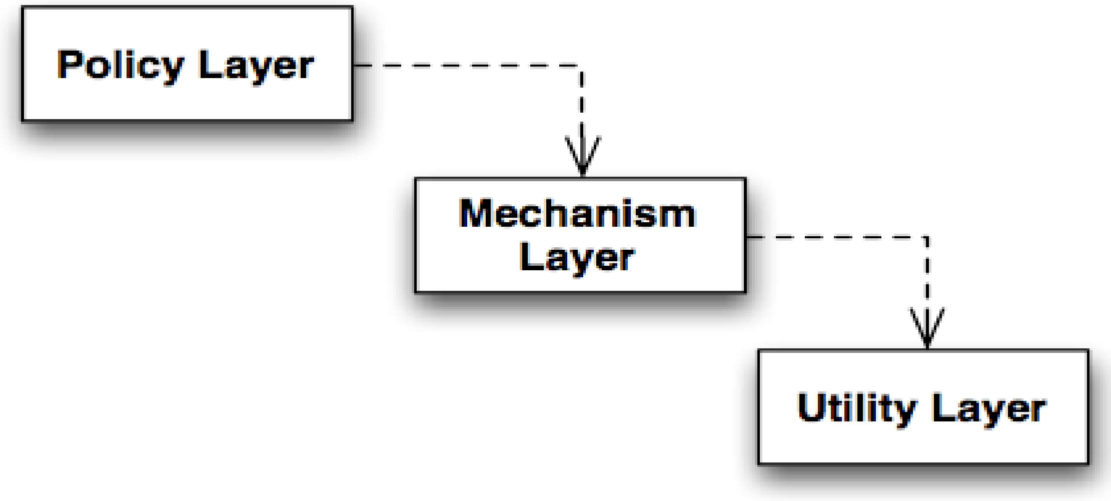
- According to this principle the way of designing a class structure is to start from high level modules to the low level modules:

High Level Classes --> Abstraction Layer --> Low Level Classes

Why "inversion"?

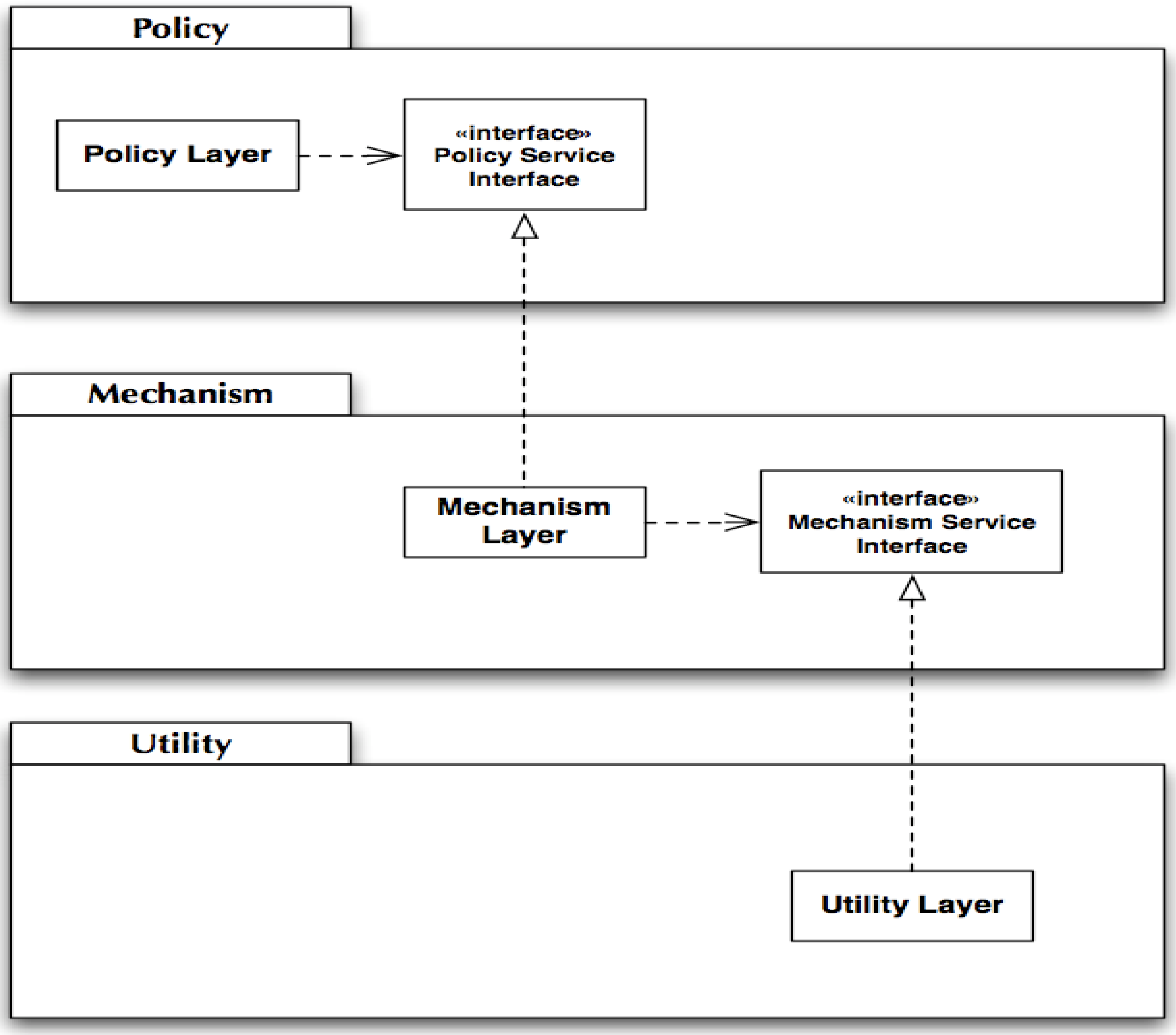
- DIP attempts to "invert" the dependencies that result from a structured analysis and design approach
  - High-Level modules tend to contain important policy decisions and business rules related to an application;
    - If they depend on low-level modules, changes in those modules can force the high-level modules to change!
    - High-level modules should not depend on low-level modules in any way
- when the high-level modules are independent of the low-level modules, the high level modules can be reused quite simply.

- Typical Layered System
  - high-level Policy layer uses a lower-level Mechanism layer, which in turn uses a detailed-level Utility layer
  - Policy layer transitively depends on the Utility layer



Solution

Layered System with DIP applied


Policy
Policy Layer
«interface» Policy Service Interface
Mechanism
Mechanism Layer
«interface» Mechanism Service Interface
Utility
Utility Layer

- Each higher-level class uses the next-lowest layer through the abstract interface.
- Thus, the upper layers do not depend on the lower layers.
- Instead, the lower layers depend on abstract service interfaces *declared in the upper layers*.

Example

- We have the **manager** class which is a high level class, and the low level class called **Worker**.
- We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers.
- We created a new class **SuperWorker** for this.

- Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker.

- Let's see the disadvantages:
- we have to change the Manager class (remember it is a complex one and this will involve time and effort to make the changes).
- Some of the current functionality from the manager class might be affected.
- The unit testing should be redone.

- All those problems could take a lot of time to be solved and they might induce new errors in the old functionality.
- The situation would be different if the application had been designed following the Dependency Inversion Principle.
- It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface.
- When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it.
- No additional changes in the existing classes.

```
// Dependency Inversion Principle - Bad example
class Worker {
        public void work()
                {
                        // ....working
                }
        }

class Manager {
        Worker worker;

        public void setWorker(Worker w)         {
                                                worker = w;
                                                }

        public void manage()
        {
        worker.work();
        }
    }

class SuperWorker {
                public void work() {
                //.... working much more
                                }
        }
```

```
• // Dependency Inversion Principle - Good example
  interface IWorker          {
                             public void work();
                             }

  class Worker implements IWorker
• {
        public void work()
        {
                // ....working
        }
}

class SuperWorker implements Iworker
{
        public void work()
        {
        //.... working much more
        }
}
```

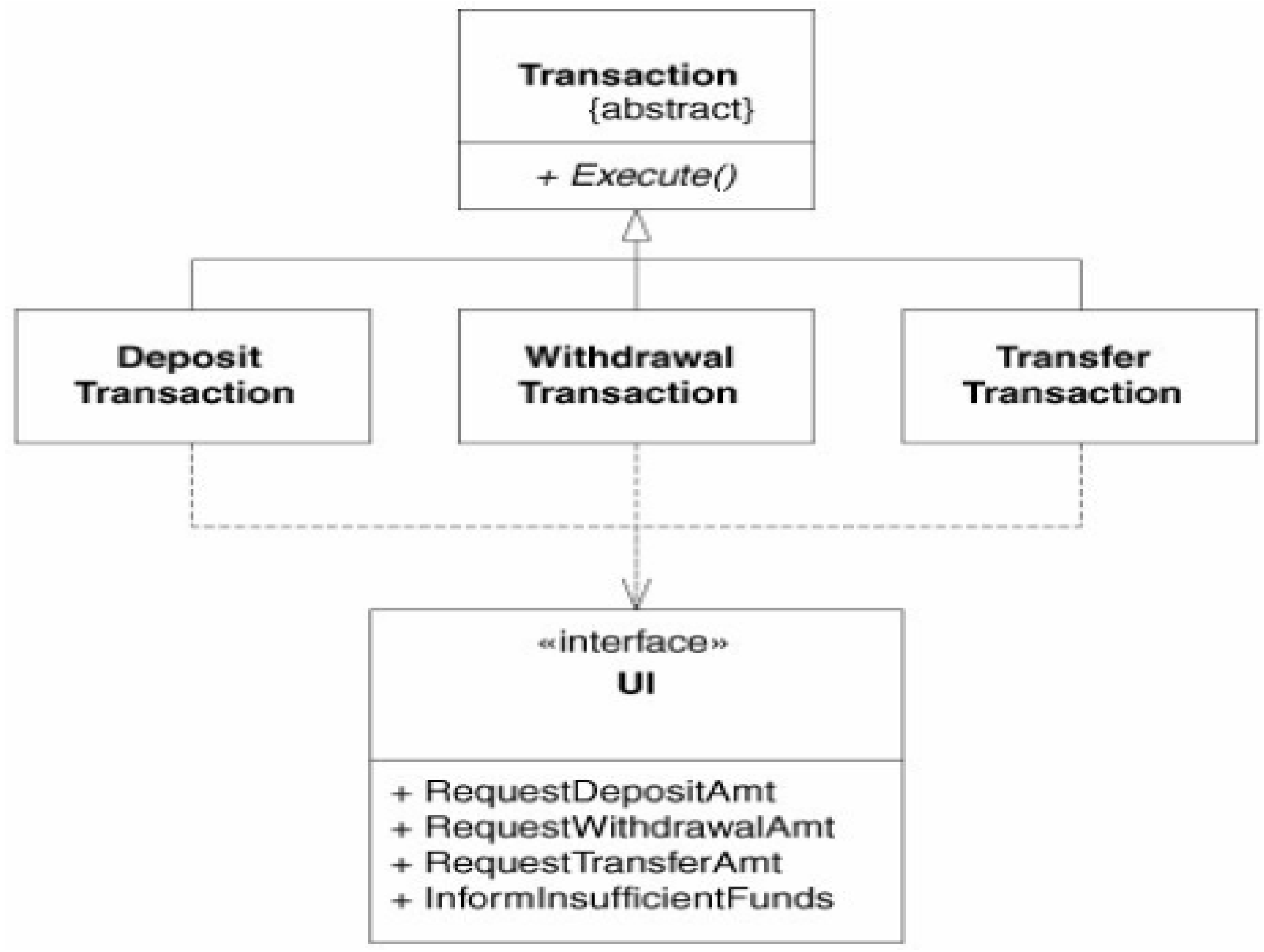
```
class Manager {
            IWorker worker;

            public void setWorker(IWorker w)
            {
            worker = w;
            }

            public void manage()
            {
                  worker.work();
            }
            }
```

5.Interface Segregation Principle

- **Clients should not be forced to depend on interfaces/methods that they do not use**
  - Instead of one **fat interface** many small interfaces are preferred based on groups of methods, each one serving one sub-module.
- A class with a “fat” interface has groups of methods that each service different clients
  - This coupling is bad however, since a change to one group of methods may impact the clients of some other group

Example

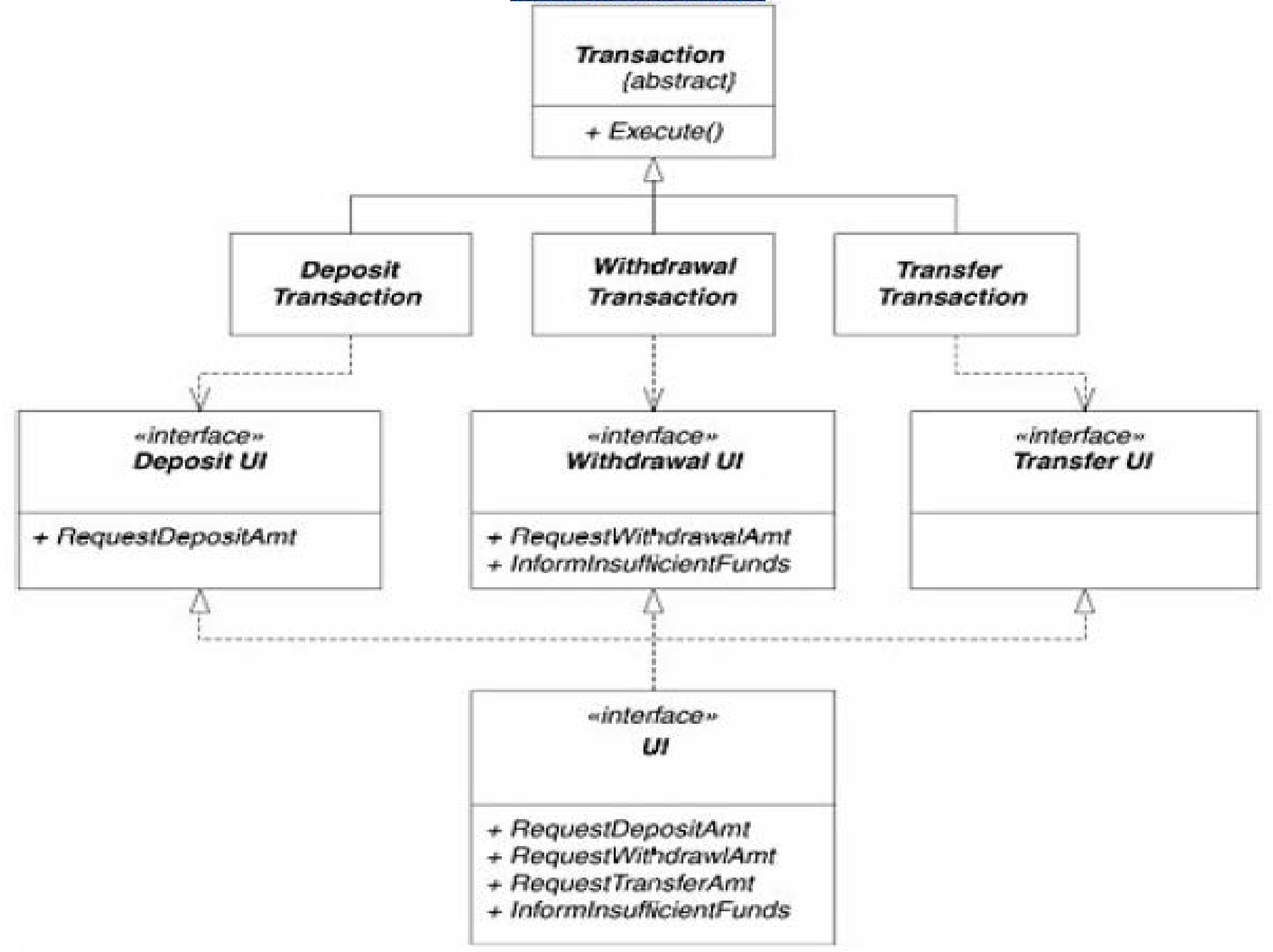
Figure 12-5. ATM transaction hierarchy



- Each of the transactions is using UI methods that no other class uses.
- This creates the possibility that changes to one of the derivatives of Transaction will force corresponding change to UI, thereby affecting all the other derivatives of transaction
- **Solution:- segregating the UI interface into individual interfaces**,such as DepositUI, WithdrawUI, and TransferUI.
- These separate interfaces can then be inherited into the final UI interface.

Figure 12-6. Segregated ATM `UI` interface

[View full size image]



- clients should have to depend only on methods that they call.
- This can be achieved by breaking the interface of the fat class into many client-specific interfaces.
- This breaks the dependence of the clients on methods that they don't invoke and allows the clients to be independent of one another.

Example

- This example violates the Interface Segregation Principle.
- We have a Manager class which represent the person which manages the workers.
- And we have 2 types of workers some average and some very efficient workers.
- Both types of workers works and they need a daily launch break to eat.

- But now some robots came in the company they work as well , but they don't eat so they don't need a launch break.
- One side the new Robot class need to implement the IWorker interface because robots works.
- On the other side, the don't have to implement it because they don't eat.
- This is why in this case the IWorker is considered a polluted interface.

- If we keep the present design, the new Robot class is forced to implement the eat method.

- According to the Interface Segregation Principle, a flexible design will not have polluted interfaces.

- In our case the IWorker interface should be split in 2 different interfaces.

- By splitting the IWorker interface in 2 different interfaces the new Robot class is no longer forced to implement the eat method.

- Also if we need another functionality for the robot like recharging we create another interface IRechargeble with a method recharge.

-
  ```
  / interface segregation principle - bad example
  interface IWorker {
  public void work();
  public void eat();
  }

  class Worker implements IWorker{
  public void work() {
  // ....working
  }
  public void eat() {
  // ...... eating in launch break
  }
  }

  class SuperWorker implements IWorker{
  public void work() {
  //.... working much more
  }

  public void eat() {
  //.... eating in launch break
  }
  }
  ```

-
  ```
  class Robot implements IWorker{
  public void work() {
  // ....working
  }
  public void eat() {
  // ...... eating in launch break
  }
  }
  ```

-
  ```
  class Manager {
  IWorker worker;

  public void setWorker(IWorker w) {
  worker=w;
  }

  public void manage() {
  worker.work();
  }
  }
  ```

-
  ```
  // interface segregation principle - good example
  interface IWorker extends Feedable, Workable {
  }

  interface IWorkable {
  public void work();
  }

  interface IFeedable{
  public void eat();
  }

  class Worker implements IWorkable, IFeedable{
  public void work() {
  // ....working
  }

  public void eat() {
  //.... eating in launch break
  }
  }

  class Robot implements IWorkable{
  public void work() {
  // ....working
  }
  }
  ```

```
class SuperWorker implements IWorkable, IFeedable{
public void work() {
//.... working much more
}

public void eat() {
//.... eating in launch break
}
}

class Manager {
Workable worker;

public void setWorker(Workable w) {
worker=w;
}

public void manage() {
worker.work();
}
}
```

Summary

- Agile design is a process, not an event
  - It’s the continuous application of principles, patterns, and practices to improve the structure and readability of software
- Agile Methods (of which XP is one) are a response to traditional software engineering practices
  - They deemphasize documents/processes and instead value people and communication