哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 6: Maintainability-Oriented Software Construction Approaches

6.1 Metrics and Construction Principles for Maintainability

Ming Liu

April 8, 2019

Outline

- **Software Maintenance and Evolution**
- **Metrics of Maintainability**
- **Modular Design and Modularity Principles**
- **OO Design Principles: SOLID**
- **OO Design Principles: GRASP**
- **Summary**

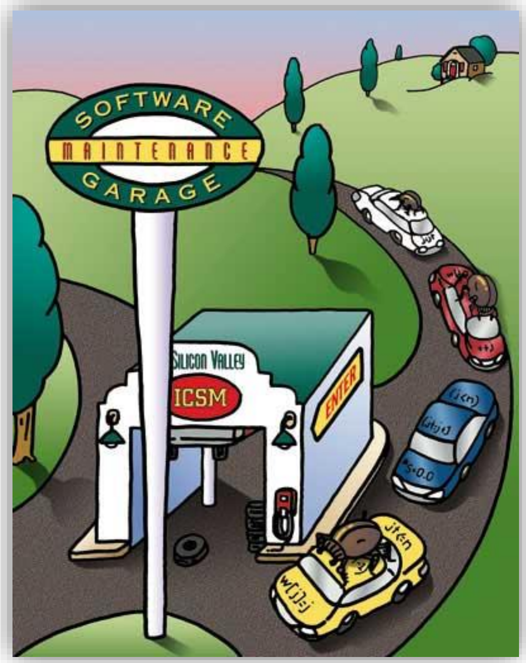
1 Software Maintenance and Evolution

What is Software Maintenance?

- **Software maintenance** in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes.
- In "ISO/IEC 14764:2006 Software Engineering — Software Life Cycle Processes — Maintenance"

Types of software maintenance

- **Corrective maintenance 25%**
  - Reactive modification of a software product performed after delivery to correct discovered problems;
- **Adaptive maintenance 21%**
  - Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment;
- **Perfective maintenance 50%**
  - Enhancement of a software product after delivery to improve performance or maintainability;
- **Preventive maintenance 4%**
  - Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

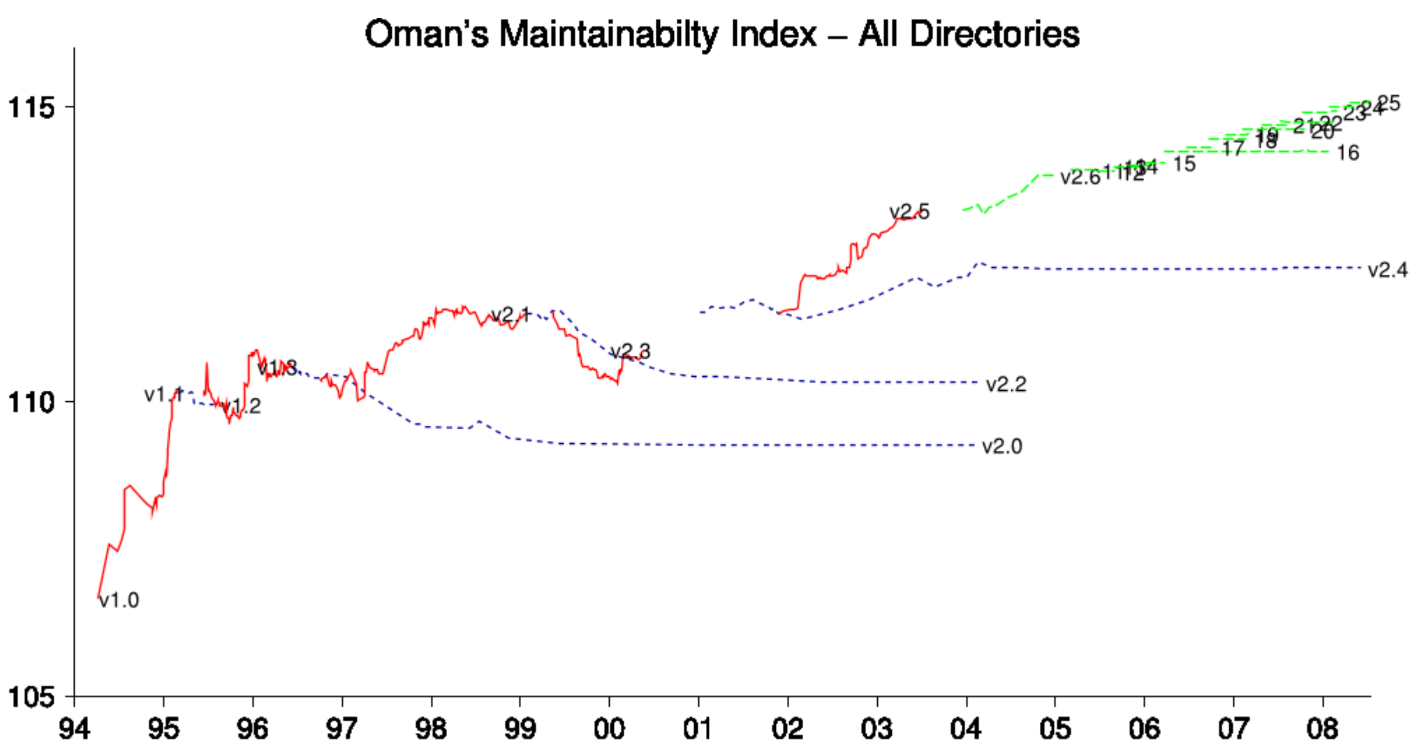


Software Evolution

- **Software evolution is a term used in software maintenance, referring to the process of developing software initially, then repeatedly updating it for various reasons.**
- **Over 90% of the costs of a typical system arise in the maintenance phase, and that any successful piece of software will inevitably be maintained.**

Objective of software maintenance and evolution

- **Objective of software maintenance and evolution: To improve the fitness / adaptability of software, and keep its vitality, i.e., "long-lasting software (low entropy software)"**
- **An example of Linux Kernel's evolution: its Maintainability Index.**



Examples of maintainability-oriented construction

- **Modular design and implementation**
  - Low coupling and high cohesion
- **OO design principles**
  - SOLID
  - GRASP
- **OO design patterns**
  - Factory method pattern, Builder pattern
  - Bridge pattern, Proxy pattern
  - Memento pattern, State pattern
- **State-based construction (Automata-based programming)**
- **Table-driven construction**
- **Grammar-based construction**

These are what to be studied in this Chapter

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2 Metrics of Maintainability

Many names of maintainability

Ready for Change
Ready for Extension

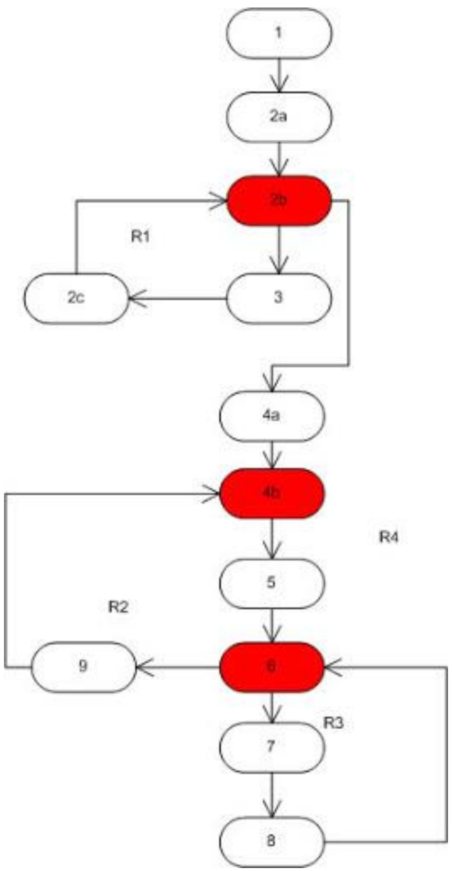
- **Maintainability** – "The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment".
- **Extensibility** – Software design/implementation takes future growth into consideration and is seen as a systemic measure of the ability to extend a system and the level of effort required to implement the extension.
- **Flexibility** – The ability of software to change easily in response to user requirements, external technical and social environments, etc.
- **Adaptability** – The ability of an interactive system (adaptive system) that can adapt its behavior to individual users based on information acquired about its user(s) and its environment.

Many names of maintainability

- **Manageability**—How efficiently and easily a software system can be monitored and maintained to keep the system performing, secure, and running smoothly.
- **Supportability**—How effectively a software can be kept running after deployment, based on resources that include quality documentation, diagnostic information, and knowledgeable and available technical staff.

Some common-used maintainability metrics

- **Cyclomatic Complexity** - Measures the structural complexity of the code.
  - It is created by calculating the number of different code paths in the flow of the program.
  - A program that has complex control flow will require more tests to achieve good code coverage and will be less maintainable.
  - CC = E-N+2, CC=P+1, CC=number of areas
- **Lines of Code** - Indicates the approximate number of lines in the code.
  - A very high count might indicate that a type or method is trying to do too much work and should be split up.
  - It might also indicate that the type or method might be hard to maintain.



Some common-used maintainability metrics

- **Halstead Volume:** a composite metric based on the number of (distinct) operators and operands in source code.
- **Maintainability Index (MI)** - Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. It is calculated based on:
  - Halstead Volume (HV)
  - Cyclomatic Complexity (CC)
  - The average number of lines of code per module (LOC)
  - The percentage of comment lines per module (COM).

$$171 - 5.2 ln(\text{HV}) - 0.23\text{CC} - 16.2 ln(\text{LOC}) + 50.0 sin\sqrt{2.46 * \text{COM}}$$

Some common-used maintainability metrics

- **Depth of Inheritance** - Indicates the number of class definitions that extend to the root of the class hierarchy. The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or/and redefined.
- **Class Coupling** - Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration.
  - Good software design dictates that types and methods should have high cohesion and low coupling.
  - High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types.
- **Unit test coverage -** indicates what part of the code base is covered by automated unit tests. (to be studied in Chapter 7)

Many other maintainability metrics

Traditional metrics

- LOC
- Cyclomatic complexity
- Halstead complexity measures
- Maintainability Index
- Unit test coverage

Language specific coding violations (Fortran)

- Use of old FORTRAN 77 standard practices, when better, modern ones are available in e.g. Fortran 2008

Code smells

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- ...

Other maintainability metrics

- Defect density
- Active files

HARBIN INSTITUTE OF TECHNOLOGY

3 Modular Design and Modularity Principles

Modular programming

- **Modular programming is** a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
- **It usually refers to high-level decomposition of the code of an entire program into pieces, and has been widely adopted in both Structured Programming and OOP.**
  - Structured programming refers to the low-level code use of structured control flow;
  - Object-oriented programming refers to the data use of objects, a kind of data structure.

Modular programming

- **The goal of design is to partition the system into modules and assign responsibility among the components in a way that:**
  - High cohesion within modules
  - Loose coupling between modules
- **Modularity reduces the total complexity a programmer has to deal with at any one time assuming:**
  - Functions are assigned to modules in away that groups similar functions together (Separation of concerns)
  - There are small, simple, well-defined interfaces between modules (Information hiding)
- **The principles of cohesion and coupling are probably the most important design principles for evaluating the maintainability of a design.**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

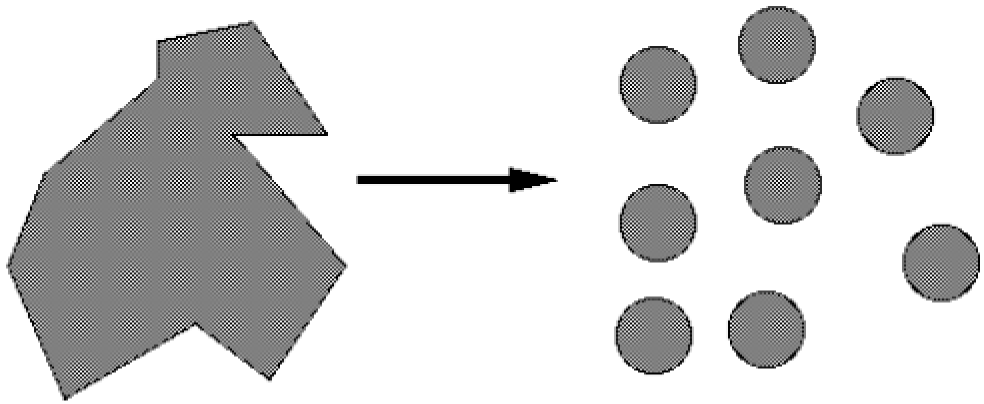
(1) Five Criteria for Evaluating Modularity

Five Criteria for Evaluating Modularity

- **Decomposability (可分解性)**
  - Are larger components decomposed into smaller components?
- **Composability (可组合性)**
  - Are larger components composed from smaller components?
- **Understandability (可理解性)**
  - Are components separately understandable?
- **Continuity (可持续性)**
  - Do small changes to the specification affect a localized and limited number of components?
- **Protection (出现异常之后的保护)**
  - Are the effects of run-time abnormalities confined to a small number of related components?

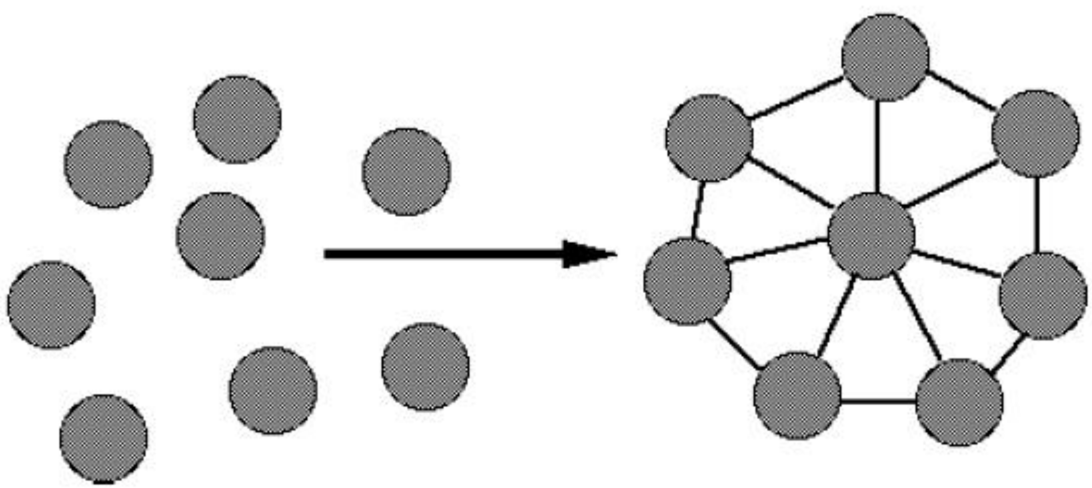
1. Decomposability

- **Decompose problem into smaller sub-problems that can be solved separately (将问题分解为各个可独立解决的子问题)**
  - Goal: keep dependencies explicit and minimal (目标：使模块之间的依赖关系显式化和最小化)
  - Example: top-down structural design



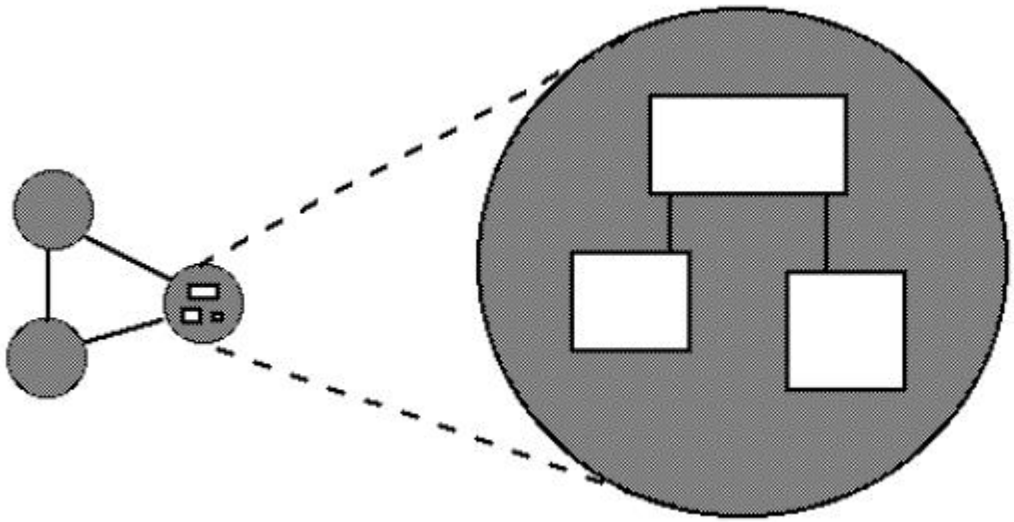
2. Composability

- **Freely combine modules to produce new systems (可容易的将模块组合起来形成新的系统)**
  - Goal: make modules reused in different environments (目标：使模块可在不同的环境下复用)
  - Example: Math libraries; UNIX command & pipes



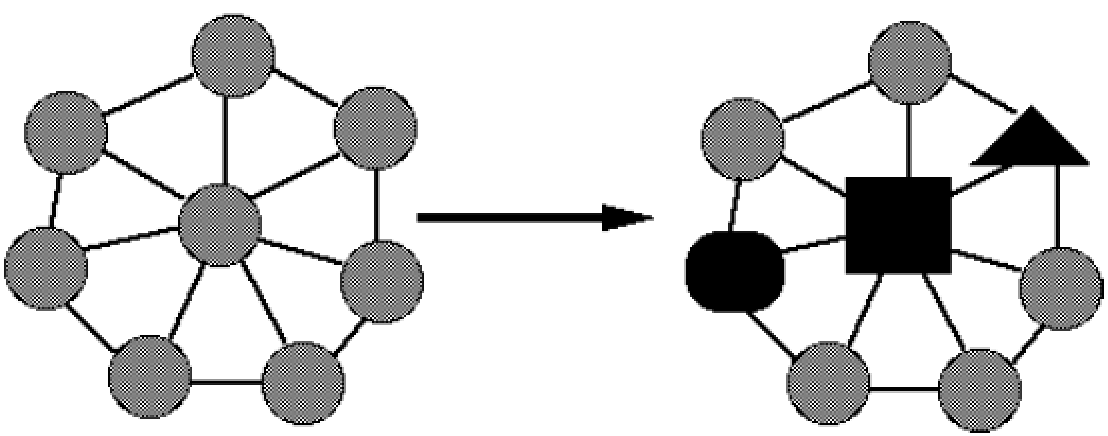
3. Understandability

- **Individual modules understandable by human reader (每个子模块都可被系统设计者容易的理解)**
  - Example: Unix shell such as Program1 | Program2 | Program3
  - Counter-example: Sequential Dependencies (A →B → C)



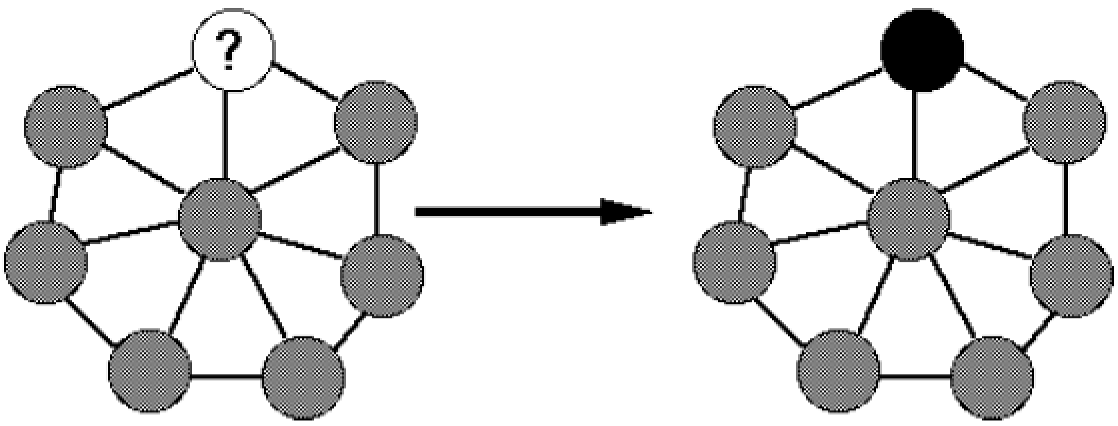
4. Continuity

- **Small change in specification results in:**
  - changes in only a few modules and does not affect the architecture (小的变化将只影响一小部分模块，而不会影响整个体系结构)
  - Example: Symbolic Constants (符号型变量)
    - const String PRODUCT_CODE = “PBS001291A”
  - Example: Principle of Uniform Access



5. Protection

- **Effects of an abnormal run-time condition is confined to a few modules (运行时的不正常将局限于小范围模块内)**
  - Example: Validating input at source



(2) Five Rules of Modularity Design

Five Rules of Modularity Design

- **Direct Mapping (直接映射)**
- **Few Interfaces (尽可能少的接口)**
- **Small Interfaces (尽可能小的接口)**
- **Explicit Interfaces (显式接口)**
- **Information Hiding (信息隐藏)**

1. Direct Mapping

- **Direct mapping: keep the structure of the solution compatible with the structure of the modeled problem domain (直接映射：模块的结构与现实世界中问题领域的结构保持一致)**

- **Impact on (对以下评价标准产生影响):**
  - Continuity (持续性)
    - easier to assess and limit the impact of change
  - Decomposability (可分解性)
    - decomposition in the problem domain model as a good starting point for the decomposition of the software

2. Few Interfaces

- **Every module should communicate with as few others as possible (模块应尽可能少的与其他模块通讯)**
  - 通讯路径的数目: n-1, n(n-1)/ 2
  - affects Continuity, Protection, Understandability and Composability (对以下评价标准产生影响：可持续性、保护性、可理解性、可组合性)

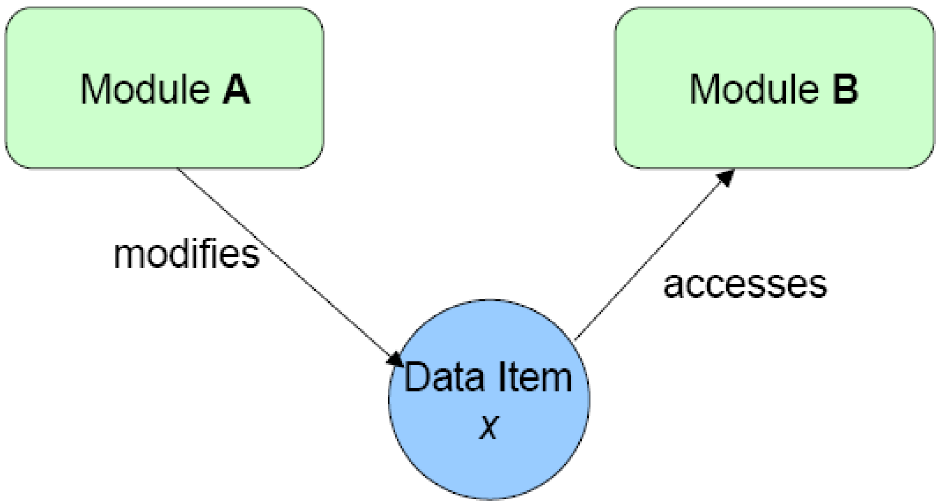
3. Small Interfaces

- **If two modules communicate, they should exchange as little information as possible (如果两个模块通讯，那么它们应交换尽可能少的信息)**
  - limited “bandwidth” of communication (限制模块之间通讯的“带宽”)
  - Continuity and Protection (对“可持续性”和“保护性”产生影响)

4. Explicit Interface

- **Whenever two modules A and B communicate, this must be obvious from the text of A or B or both (当A与B通讯时，应明显的发生在A与B的接口之间)**
  - Decomposability, Composability, Continuity, Understandability (受影响的评价标准：可分解性、可组合性、可持续性、 可理解性)

反例：



…再次强调Rule 2-3-4

- **(2) Few Interfaces:** *"Don't talk to many!"*

  尽可能少的接口："不要对太多人讲话…"

- **(3) Small Interfaces:** *"Don't talk a lot!"*

  尽可能小的接口："不要讲太多…"

- **(4) Explicit Interfaces:** *"Talk loud and in public! Don't whisper!"*

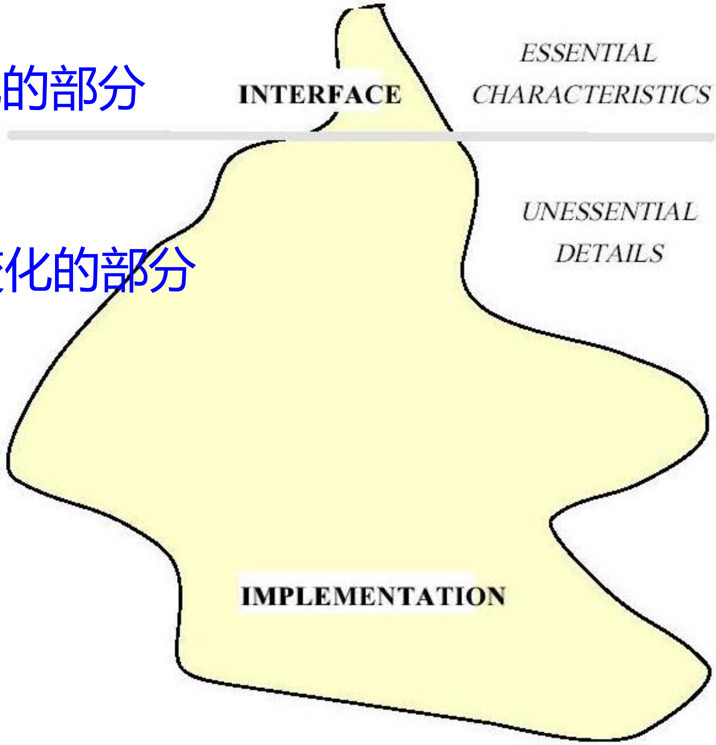
  显式接口："公开的大声讲话…不要私下嘀咕…"

5. Information Hiding

- **Motivation: design decisions that are subject to change should be hidden behind abstract interfaces (经常可能发生变化的设计决策应尽可能隐藏在抽象接口后面)**
  - Impact on Continuity
    (影响“可持续性”)

较少发生变化的部分

需要经常发生变化的部分



- **The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to client modules.**

(3) Coupling and Cohesion

Coupling

- **Coupling is the measure of dependency between modules. A dependency exists between two modules if a change in one could require a change in the other.**
- **The degree of coupling between modules is determined by:**
  - The number of interfaces between modules (quantity), and
  - Complexity of each interface (determined by the type of communication) (quality)

Coupling between CSS and JavaScript

- **A well-designed web app modularizes around:**
  - HTML files which specify data and semantics
  - CSS rules which specify the look and formatting of HTML data
  - JavaScript which defines behavior/interactivity of page
- **Assume you have the following HTML and CSS definitions.**

Example

- **HTML:**

```
<!doctype html>
<html>
<head>
  <script type="text/javascript" src="base.js"></script>
  <link rel="stylesheet" href="default.css">
</head>
<body>
  <button onclick="highlight2()">Highlight</button>
  <button onclick="normal2()">Normal</button>
  <h1 id="title" class="NormalClass">CSS <--> JavaScript Coupling</h1>
</body>
</html>
```

coupling-example.html

- **CSS:**

```
.NormalClass {
  color:inherit;
  font-style:normal;
}
```

default.css

- **Output:**

Option A

- **JavaScript code modifies the <u>style</u> attribute of HTML element.**

```
function highlight() {
    document.getElementById("title").style.color="red";
    document.getElementById("title").style.fontStyle="italic";
}

function normal() {
    document.getElementById("title").style.color="inherit";
    document.getElementById("title").style.fontStyle="normal";
}
```

base.js

Option B

- **JavaScript code modifies the class attribute of HTML element.**

```
function highlight() {
    document.getElementById("title").className = "HighlightClass";
}

function normal() {
    document.getElementById("title").className = "NormalClass";
}
```

base.js

```
.NormalClass {
  color:inherit;
  font-style:normal;
}

.HighlightClass {
  color:red;
  font-style:italic;
}
```

default.css

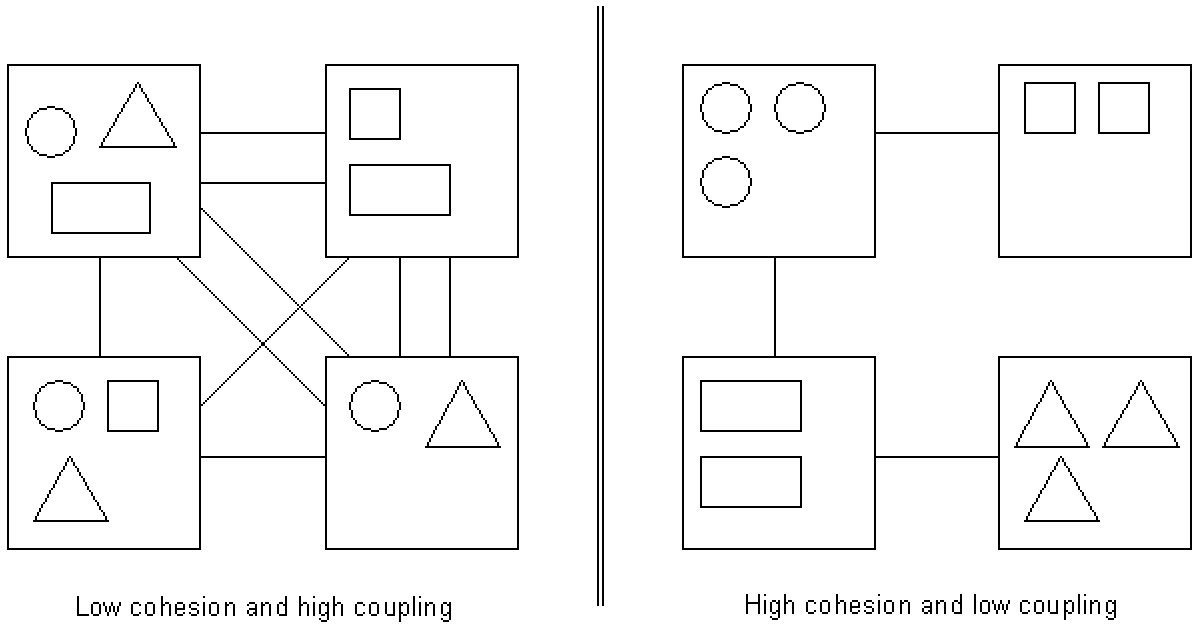
Cohesion

- **Cohesion is a measure of how strongly related the functions or responsibilities of a module are.**
- **A module has high cohesion if all of its elements are working towards the same goal.**

Cohesion and Coupling

- **The best designs have high cohesion (also called strong cohesion) within a module and low coupling (also called weak coupling) between modules.**

화합, 결합



4 OO Design Principles: SOLID

SOLID

- **Single responsibility principle (SRP)**
- **Open/closed principle (OCP)**
- **Liskov substitution principle (LSP)**
- **Interface segregation principle (ISP)**
- **Dependency inversion principle (DIP)**

(1) Single responsibility principle (SRP)

Single Responsibility Principle

- **"There should never be more than one reason for a class to change", i.e., a class should concentrate on doing one thing and one thing only.**

Just because you can, doesn't mean you should

Single Responsibility Principle

- **Two resposibilities**

```
interface Modem {
 public void dial(String pno);
 public void hangup();

 public void send(char c);
 public char recv();
}
```

- **Connection Management + Data Communication**

Single Responsibility Principle

```
interface DataChannel {
 public void send(char c);
 public char recv();
}
```

```
interface Connection {
 public void dial(String phn);
 public char hangup();
}
```

(2) Open/closed principle (OCP)

Open Closed Principle

- **"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification“, i.e., change a class' behavior using inheritance and composition**

Open Closed Principle

```
// 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;
    }
 }
```

Open Closed Principle – a Few Problems….

- **Impossible to add a new** `Shape` **without modifying** `GraphEditor`
- **Important to understand** `GraphEditor` **to add a new** `Shape`
- **Tight coupling between** `GraphEditor` **and** `Shape`
- **Difficult to test a specific** `Shape` **without involving** `GraphEditor`
- `If-Else-/Case` **should be avoided**

Open Closed Principle - Improved

```
// 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
   }
}
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(3) Liskov substitution principle (LSP)

Liskov Substitution Principle

- **"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it", i.e., subclasses should behave nicely when used in place of their base class**

Liskov Substitution Principle

```
// Violation of LSP

 class Rectangle {
    int m_width;
    int m_height;

    public void setWidth(int width){
        m_width = width;
    }
    public void setHeight(int h){
        m_height = ht;
    }
    public int getWidth(){
        return m_width;
    }
    public int getHeight(){
        return m_height;
    }
    public int getArea(){
        return m_width * m_height;
    }
 }
```

```
class Square extends Rectangle {
    public void setWidth(int width){
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height){
        m_width = height;
        m_height = height;
    }
}
```

Liskov Substitution Principle

```
class LspTest{

   private static Rectangle getNewRectangle(){
       // it can be an object returned by some factory ...
       return new Square();
   }

   public static void main (String args[]){
       Rectangle r = LspTest.getNewRectangle();
       r.setWidth(5);
       r.setHeight(10);

//user knows that r it's a rectangle. It assumes that he's able to set
the width and height as for the base class

       System.out.println(r.getArea());
       // now he's surprised to see that the area is 100 instead of 50.
   }
}
```

(4) Interface segregation principle (ISP)

Interface Segregation Principle

- **"Clients should not be forced to depend upon interfaces that they do not use“, i.e., keep interfaces small.**

- **Don’t force classes so implement methods they can’t (Swing/Java)**
- **Don’t pollute interfaces with a lot of methods**
- **Avoid ’fat’ interfaces**

Interface Segregation Principle

```
//bad example (polluted interface)
interface Worker {
    void work();
    void eat();
}
```

```
ManWorker implements Worker {
    void work() {…};
    void eat() {30 min break;};
}
```

```
RobotWorker implements Worker {
    void work() {…};
    void eat() {//Not Appliciable
                 for a RobotWorker};
}
```

Interface Segregation Principle

- **Solution: split into two interfaces**

```
interface Workable {
   public void work();
}

interface Feedable{
   public void eat();
}
```

(5) Dependency inversion principle (DIP)

Dependency Inversion Principle

- **A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
B. Abstractions should not depend upon details. Details should depend upon abstractions", i.e., use lots of interfaces and abstractions.**

Dependency Inversion Principle

```
//DIP - bad example
public class EmployeeService {
        private EmployeeFinder emFinder //concrete class, not abstract.
         //Can access a SQL DB for instance
         public Employee findEmployee(…) {
                 emFinder.findEmployee(…)
         }
}
```

Dependency Inversion Principle

```
//DIP - fixed
public class EmployeeService {
        private IEmployeeFinder emFinder
        //depends on an abstraction, no an implementation

        public Employee findEmployee(…) {
           emFinder.findEmployee(…)
        }
}
```

- **Now its possible to change the finder to be a XmEmployeeFinder, DBEmployeeFinder, FlatFileEmployeeFinder, MockEmployeeFinder….**

Why DIP?

- **Advantages:**
  - Formalize class contracts.
  - You an define the services of a routine in terms of pre- and post-conditions. This makes it very clear what to expect.

- **Try Design for Testing**
  - Create a test-friendly design
  - A test-friendly module is likely to exhibit other important design characteristics.
  - Example: you would avoid circular dependencies. Business logic will be better isolated from UI code if you have to test it separately from the UI

5 OO Design Principles: GRASP

What’s GRASP patterns

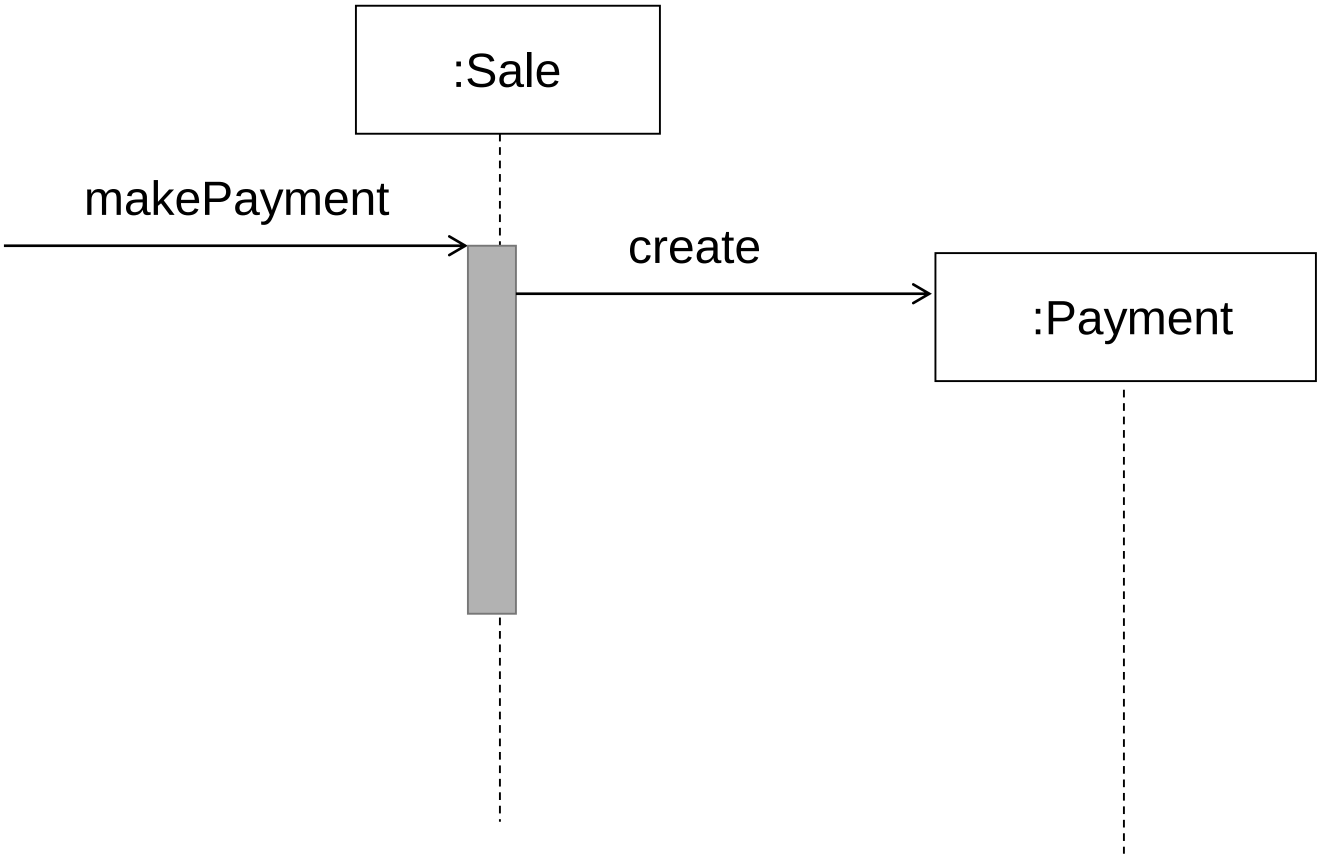
- **General Responsibility Assignment Software Patterns (principles)**, abbreviated **GRASP**, consist of guidelines for assigning responsibility to classes and objects in OOP.

- **The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.**
- **This approach to understanding and using design principles is based on patterns of assigning responsibilities to classes.**

What's responsibility

- **Responsibility of an object: related to the obligations of an object**
- **Knowing:**
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate
- **Doing:**
  - Doing something itself, such as creating an object or doing a calculation
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects.

Responsibilities and methods



Responsibilities are implemented using methods: `makePayment` implies `Sale` object has a responsibility to create a `Payment` object

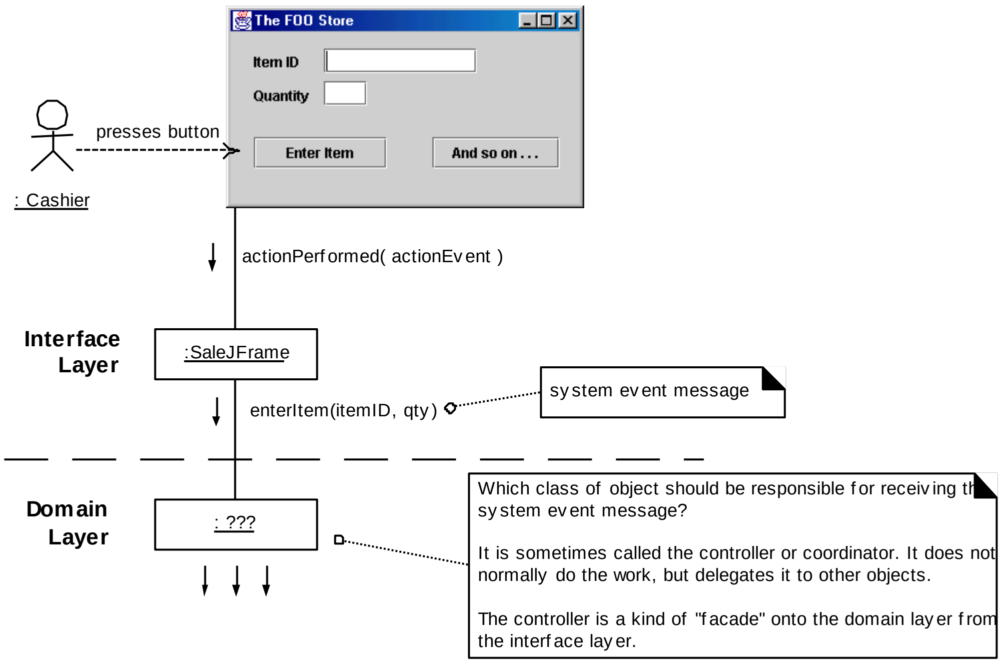
(1) Controller

- **Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:**
  - Represent the overall system, device or subsystem (Façade controller)
  - Represent a use case scenario within which the system events occur. Often named <UseCaseName>Handler or <UseCaseName>Session
    - Use the same controller for all system events in the same use case scenario

- **Problem :**
  - Who should be responsible for handling an input system event ?

Example of Controller pattern

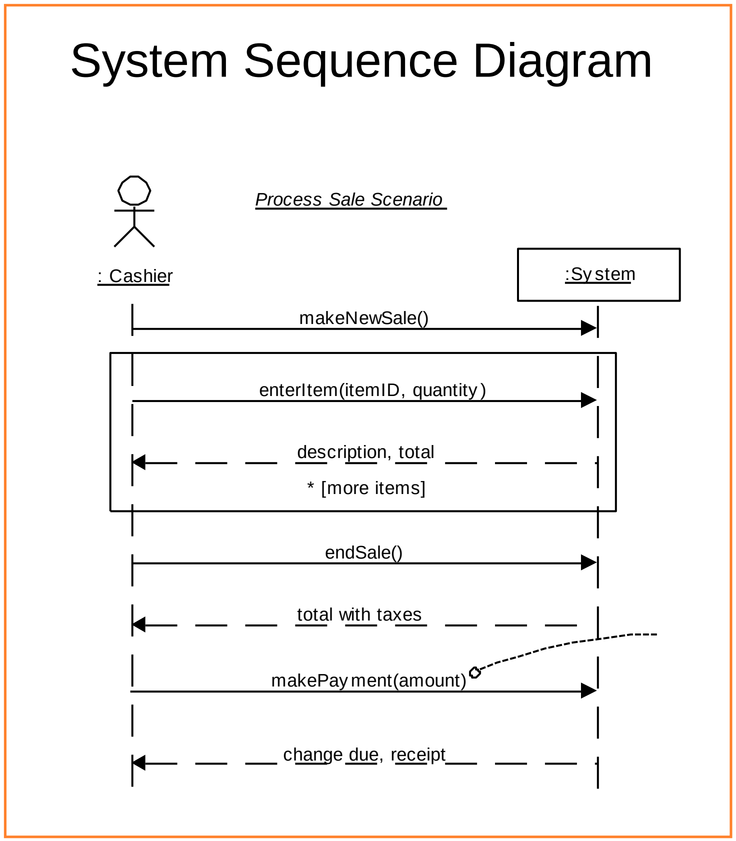
- **Who should be the controller for system events ?**

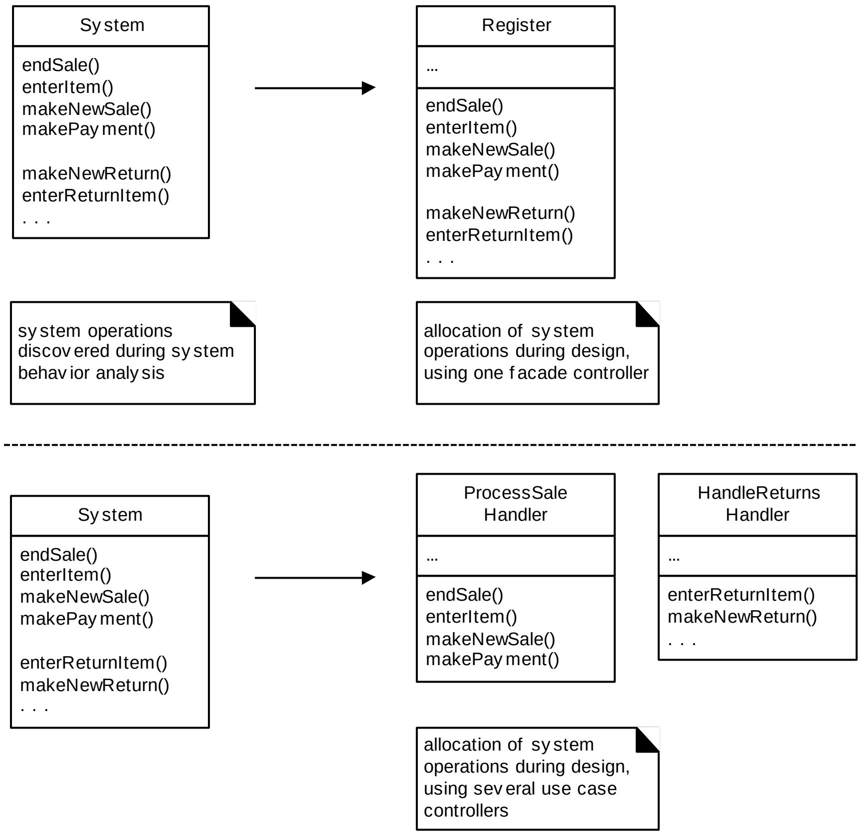


The Controller pattern suggest :

- A Facade controller
  - Register
  - POSSystem
- A receiver or handler of all system events of a use case scanario
  - ProcessSaleHandler
  - ProcessSaleSession

System events and Allocating system operations



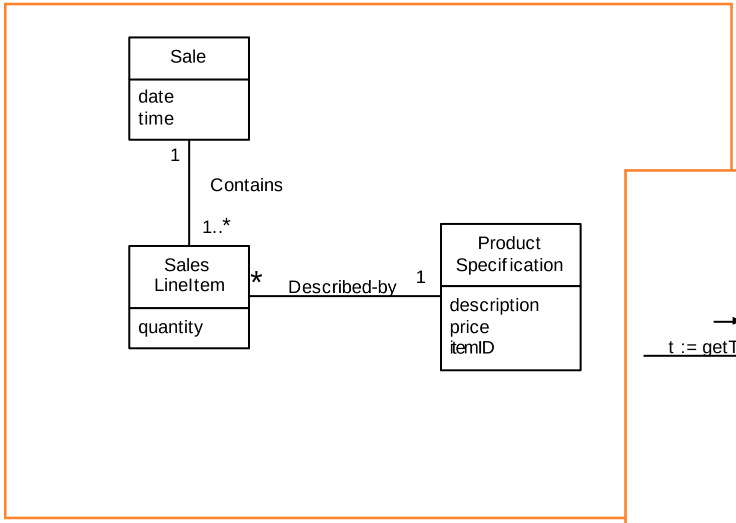


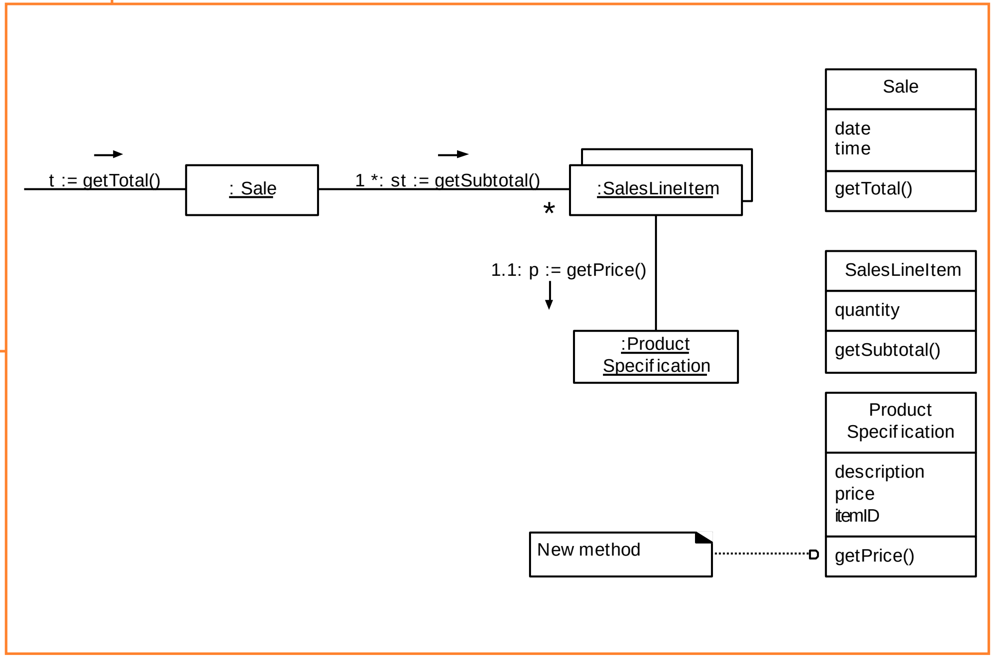
(2) Information Expert

- **Assign a responsibility to the information expert – the class that has the information necessary to fulfill the responsibility.**
- **What is a general principle of assigning responsibilities to objects?**
  - Who should be responsible for knowing/doing …?
  - Domain model (domain expert, domain analysis) to design model (software classes).
  - Any existing to any representative.

Example of Information Expert

- **Who should be responsible for knowing the grand total of a sale ?**





(3) Creator

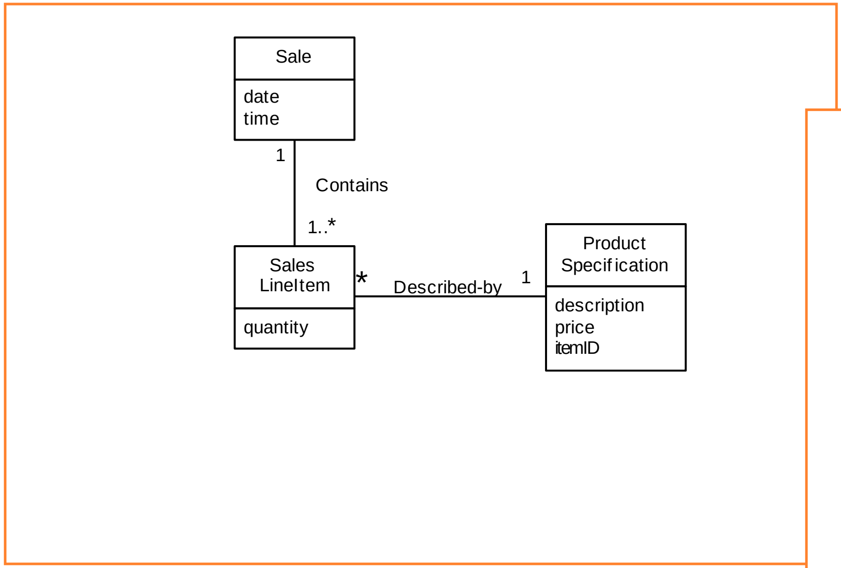
- **Class B has the responsibility to create an instance of class A if :**
  - B aggregates A objects
  - B contains A objects
  - B closely uses A objects
  - B has the initializing data.

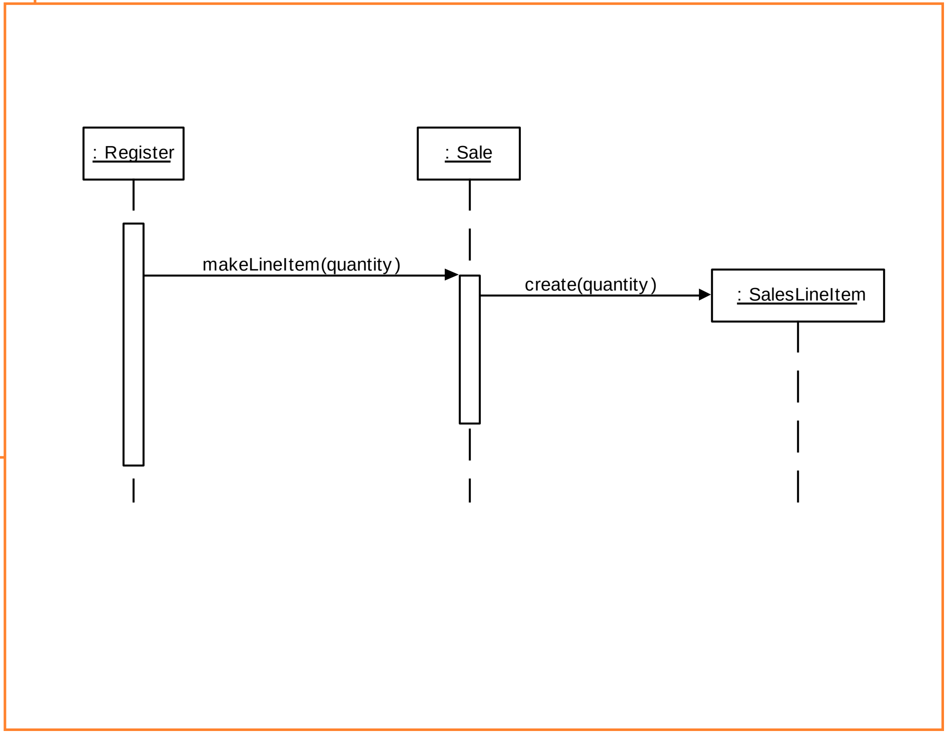
  - B is a creator of A objects.

- **Problem :**
  - Who should be responsible for creating a new instance of some class ?

Example of Creator

- **Who should be responsible for creating a SalesLineItem ?**



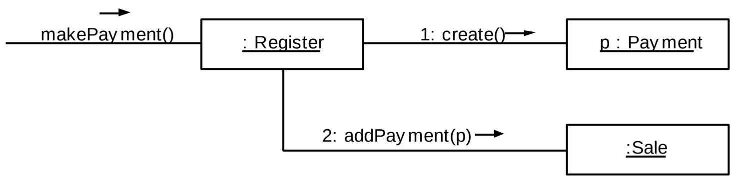


(4) Low Coupling

- **Assign a responsibility so that coupling remains low.**
- **Problem :**
  - How to support low dependency, low change impact, and increased reuse?
  - Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

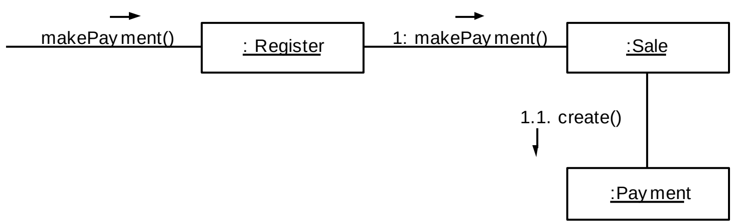
Example of Low Coupling

- **Who should be responsible for creating a Payment instance and associate with a Sale ?**



Bad Coupling

Here are two patterns – Low Coupling and Creator – that suggests different solutions.



Good Coupling

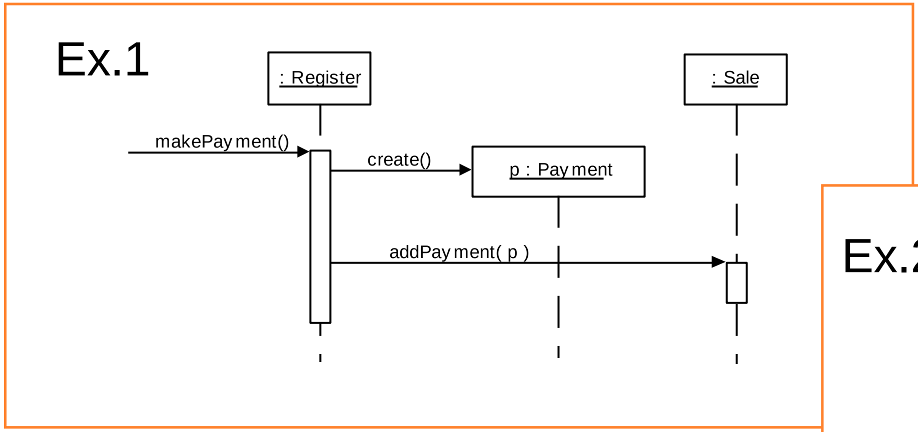
Low Coupling is an evaluative principle.

(5) High Cohesion

- **Assign a responsibility so that cohesion remains high.**
- **Problem :**
  - How to keep complexity manageable ?
  - Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

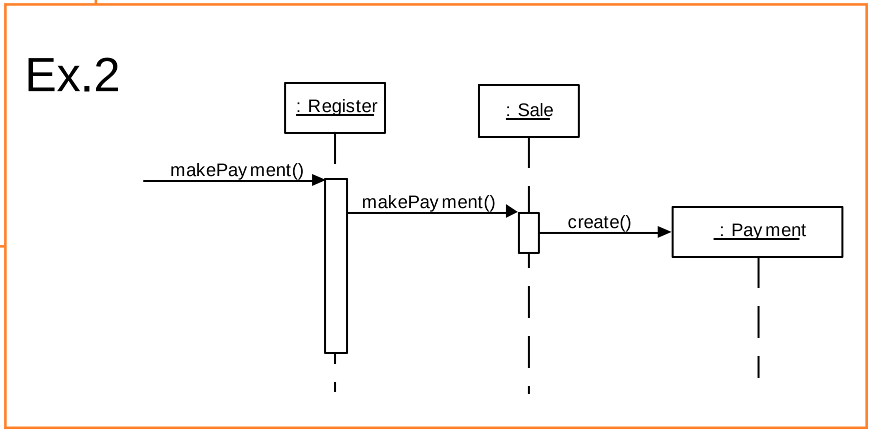
Example of High Cohesion

- **Who should be responsible for creating a Payment instance and associate it with a Sale ?**



High Cohesion pattern suggest ex. 2

Creator pattern suggest ex. 1



(6) Indirection

- **The indirection pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object.**
- **An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view-controller pattern.**

(7) Polymorphism

- **According to polymorphism principle, responsibility of defining the variation of behaviors based on type is assigned to the type for which this variation happens.**
- **This is achieved using polymorphic operations.**
- **The user of the type should use polymorphic operations instead of explicit branching based on type.**

(8) Protected variations

- **The protected variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.**

(9) Pure fabrication

- **A pure fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the information expert pattern does not).**
- **This kind of class is called a "service" in domain-driven design.**

The end

April 8, 2019