中国科学技术大学
University of Science and Technology of China

# Software Architecture

SSE USTC Qing Ding
dingqing@ustc.edu.cn
http://staff.ustc.edu.cn/~dingqing

# OO Design Principles

# Outline

SOLID

- Overview of OO principles
- Open-Closed Principle
- Liskov Substitution Princple
- Single Responsibility Principle
- Law of Demeter
- Divide and conquer
- High Cohesion
- Low Coupling
- Enterprise Architecture Principles

# Overview of OO principles

# Software Design

- Design is the process of inventing new software entities to satisfy requirements
- Software entities might be different things depending on the level of granularity and programming paradigm
- Procedural programming: procedures, variables, ...
- OO programming: classes, objects, relations between them, ...

- 设计是发明新的软件实体以满足需求的过程
- 根据粒度级别和编程范式，软件实体可能是不同的东西
- 过程性编程:程序、变量、……
- 面向对象编程:类、对象和它们之间的关系

# Software Design

- Design is not an exact entity
- For example, you can calculate how fast is an algorithm
- But you can not measure how “good” is a design
- However, there are certain design principles and design rules that you should follow

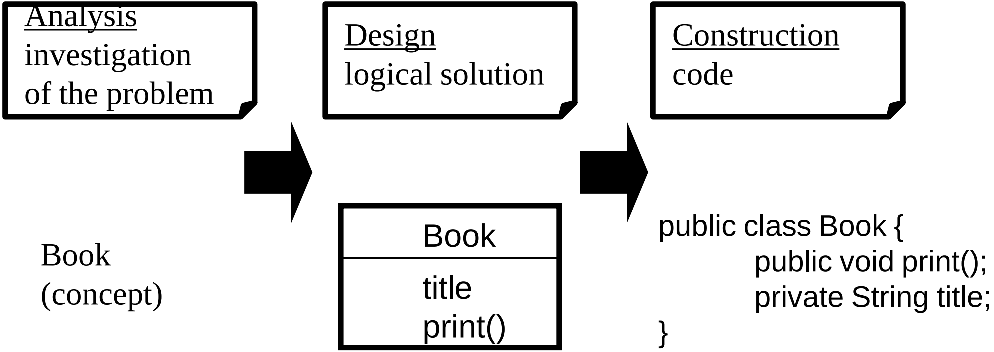
- 设计不是一个确切的实体
- 例如，你可以计算一个算法有多快
- 但你无法衡量一个设计有多“好”
- 然而，有一些你应该遵循的设计原则和设计规则

# Software Design

- These design principles come from the experience of numerous programmers
- Some of the experiences can be built directly into a programming language
- E.g. absence of direct multiple inheritance in Java
- It was a design decision to leave it out of the language because of problems it might cause

· 这些设计原则来自于许多程序员的经验
· 有些经验可以直接构建到编程语言中
· 例如，Java中没有直接的多重继承
· 把它排除在语言之外是一个设计决策，因为它可能会导致问题

# From Design to Implementation



Analysis
investigation
of the problem

Design
logical solution

Construction
code

Book
(concept)

Book

title
print()

```
public class Book {
        public void print();
        private String title;
}
```

Domain concept

Representation in analysis of concepts

Representation in an object-oriented programming language.

# Process of Design

- Design is all about decisions.
- Approaches: Top down – start with the architecture 🡐
- Bottom up – start with utilities
- Are a number of very serious design principles that lead to maintainable software that may persist for years
- Will look at satisfying functional requirements while accommodating portability, reuse potential, performance
- There are always TRADEOFFS! There is no free lunch!!!--
- Will look at several of the tradeoffs
- Will look at (and you will develop) a software architecture to support your high-level design

- 设计就是决定。
- 方法:自顶向下,从架构开始
  自底向上-从实用程序开始
- 是一些非常重要的设计原则，这些原则导致可维护的软件可以持续多年
- 将着眼于满足功能需求，同时适应可移植性、重用潜力和性能
- 总会有折衷的!天下没有免费的午餐!
- 将会看到几个折衷方案
- 将着眼于(你将开发)一个软件架构来支持你的高层设计

# Design as a series of decisions

设计是一系列的决定

- A designer is faced with a series of *design issues*
  - These are sub-problems of the overall design problem.
  - Are **always** several alternative solutions:  design *options*.
  - Designer makes a ***design decisions*** to resolve each issue.
    - This process involves choosing the **best** option from among the alternatives.
  - Recognize that there may be a number of solutions – in fact, there may be a number of good solutions for the problem to be solved.
  - We would like the ‘best’ one.

设计师面临着一系列的设计问题
-这些是整体设计问题的子问题。
-总是有几个可供选择的解决方案:设计选项。
-设计师做出设计决策来解决每个问题。
· 这个过程包括从众多备选方案中选择最佳方案。
-认识到可能有多种解决办法-事实上，对于待解决的问题可能有多种好的解决办法。
-我们想要最好的

# Making decisions

- To make each design decision, the software engineer uses:
  - Knowledge of
    - the requirements (use cases, UI prototype, supplementary specification document, class diagrams, interaction diagrams …)
    - the design as created ‘so far’
    - Available technologies (RMI, RPC, xml, jsp, servlets, html, jdbc, etc. etc.) given a development environment
    - software design principles and ‘best practices’
    - what has worked well in the past
  - Sometimes there is no single, best solution.
  - Sometimes they conflict – each presenting pros and cons

为了做出每个设计决策，软件工程师使用:

--知识

· 需求(用例、UI原型、补充规范文档、类图、交互...)

· 目前为止创建的设计

· 给定开发环境的可用技术(RMI、RPC、xml、jsp、servlet、html、jdbc等)

· 软件设计原则和 “ 最佳实践 ”

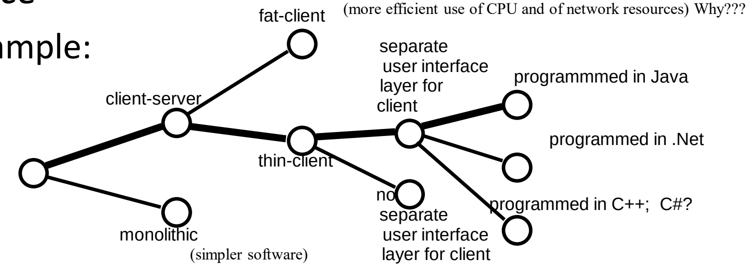
· 过去成功的事情

-有时没有唯一的、最好的解决方案。

-有时他们会发生冲突-每个都有优缺点

# 'Design space' – example Consider choice of thin client vs fat client options:

- The space of possible designs that could be achieved by choosing <u>different sets of alternatives</u> is often called the *design space* 通过选择不同的备选方案可以实现的可能设计空间通常被称为设计空间
  - – For example:



fat-client

(more efficient use of CPU and of network resources) Why???

client-server

separate user interface layer for client

programmmed in Java

programmed in .Net

thin-client

no separate user interface layer for client

programmed in C++; C#?

monolithic

(simpler software)

**Adv of fat client: bandwidth, networking services, reduced need for powerful server…**

**Adv of thin client: simpler client devices; maintaining services; central bus logic…**

**Disadvantages? Know! Cost, reliability, maintenance, security, bandwidth, network traffic; Never a single answer for all cases! ➔ Design!**

胖客户端Adv:带宽，网络服务，减少对强大服务器的需求…

瘦客户端Adv:更简单的客户端设备;维护服务;中央总线逻辑…

缺点呢?知道!成本、可靠性、维护、安全性、带宽、网络流量; 从来没有一个单一的答案针对所有的情况!➔设计

# Criticality of Certain Design Decisions

某些设计决策的临界性

- Some design decisions are critical;  others not so.

- Example:  architectural design decision to separate the user interface module from rest of system.

  – Yes:  easier to develop and maintain, internationalize, employ reuse.

  – No:   Likely not as efficient; (disadvantage.  **why**?)

  – **Recommend** iterating User Interface as a part of iteration planning!

  – As increments of value are produced, so too should the interface evolve.

一些设计决策是至关重要的;另一些则不是。
· 举例:架构设计决定将用户界面模块从系统的其他部分分离出来。
-是的:更容易开发和维护，国际化，重用。
-不:可能没有那么有效率;(不利。为什么?)
-推荐迭代用户界面作为迭代计划的一部分!
-随着价值的增加，界面也应该进化

# Modeling – Design Model

- Note that the design model (assumes a good domain model depicting important concepts in the business domain) really is a set of diagrams showing logical design.

- This will include software class diagrams, design level sequence diagrams (as opposed to analysis level sequence diagrams using conceptual objects…. and lots of abstraction) as well as package diagrams.  (Larman)

- The software architecture model will include a summary of design ideas and their motivations.  Why???

- 请注意，设计模型(假设有一个描述业务领域中重要概念的良好领域模型)实际上是一组显示逻辑设计的图。
- 这将包括软件类图、设计级序列图(相对于使用概念对象的分析级序列图...和许多抽象)以及包图。(Larman)
- 软件架构模型将包括设计思想及其动机的总结。为什么? ?

# Features of Top-down and Bottom-up design

自顶向下设计

-首先设计系统的高层结构。

-然后逐步细化到关于低级构想的详细决定。

-最后作出详细决定，例如:

 · 特定数据项的格式;

 · 将使用的单个算法。

-从软件架构和将使用的数据库类型开始(而不是 “ 哪个 ” 数据库)。

-最终得到具体的数据项和详细的算法。

- Top-down design
  - First design the very high level structure of the system.
  - Then gradually work down to detailed decisions about low-level constructs.
  - Finally arrive at detailed decisions such as:
    - the format of particular data items;
    - the individual algorithms that will be used.
  - **Start** with the **software architecture** and the **type** of **database** that will be used (**not** ‘which’ database).
  - Ultimately arrive at specific data items and detailed algorithms.

# Top-down and **bottom-up** design

- Bottom-up design

自底向上设计
-决定可重用的低级实用程序。
-然后决定如何将这些组合在一起来创建高级结构

  - Make decisions about **reusable** low-level utilities.
  - Then decide how these will be put together to create high-level constructs.

通常使用自顶向下和自底向上的混合方法:
-自上而下的设计几乎总是需要给系统一个好的结构(架构)。
-自下而上的设计通常是有用的，这样可以创建可重用的组件

- A mix of top-down and bottom-up approaches is normally used:
  - Top-down design is almost always needed to give the system a good **structure** (**architecture**).
  - Bottom-up design is normally useful so that **reusable** components can be created.

# Different aspects of Design – Very important!!

- *All kinds of ‘design’* ***This is where the decisions are made!!!!***
- ➔*Architecture design*:
  - The division into **subsystems and components**,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

•For a while, now, we will emphasize Architectural Design – after we discuss class design.

各种各样的“设计”，这是做决定的地方!!

➔架构设计:

· 子系统和组件的划分,

-这些将如何连接。

-他们将如何互动。

-他们的接口。

-类设计:

·类的各种特性。

-用户界面设计

-算法设计:

·计算机制的设计。

-协议设计:

·通信协议的设计。

现在，在我们讨论过类设计之后，我们将会着重于架构设计

# Principles Leading to Good Design

- Overall *goals* of Good Design:
  - – Increase profit by **reducing cost** and **increasing revenue**
  - – Ensure **design accommodates requirements**
  - – **Speed up development** for use / competing in marketplace
  - – **Increase qualities** such as
    - Usability – learnability;  ease of use;  on-line help…
    - Efficiency
    - Reliability
    - Maintainability
    - Reusability to reduce cost and increase revenues

优秀设计的总体目标:
-通过降低成本和增加收入来增加利润
-确保设计符合要求
-加速开发使用/在市场竞争
-增加品质如
•可用性-易学性;易用性;联机帮助......
•效率
•可靠性
•可维护性
•可重用性，降低成本，增加收入

# OO design principles

- The design principles that we will discuss lay the foundation of solid OO practices
  - Open-Closed principle
  - Design by contract/Liskov substitution principle
  - Single responsibility principle
  - Law of Demeter
  - Divide and conquer
  - High Cohesion
  - Low Coupling

我们所讨论的设计原则奠定了OO实践的坚实的基础
•开关原则
•契约设计/利斯科夫替代原则
•单一责任原则
•得墨忒耳定律
•分而治之
•高内聚
•低耦合

# Open for extension, closed for modification

# Open-Closed Principle: Example

## Online publication system

An online publication system allows users to search in publications by the year of publication. Search functionality is supported by a search filter that filters publications by comparing their publication year with a user query.

在线出版系统允许用户按出版年份搜索出版物。搜索筛选器支持搜索功能，该筛选器通过比较发布年份和用户查询来筛选发布

# Open-Closed Principle: Example

```
public class Filter {
public List filterOnYearOfPublication(Collection
pubs,  int year) {
List<Publication> filtered =  new
ArrayList<Publication>();  for (Publication pub :
pubs) {
if (pub.getYearOfPublication() == year)
{  filtered.add(pub);
}
}
return filtered;
}
}
```

# Open-Closed Principle: Example

## Online publication system

Extend the search filter to also filter by the title.

扩展搜索过滤器，让它也按标题过滤。

# Open-Closed Principle: Example

```
public class Filter {
...
public List filterOnTitleOfPublication(Collection pubs,
String title) {
List<Publication> filtered =  new ArrayList<Publication>();
for (Publication pub : pubs) {
if (pub.getTitleOfPublication().equals(title))
{  filtered.add(pub);
}
}
return filtered;
}
}
```

# Open-Closed Principle

- This is the cornerstone principle of OOD
- All software changes during its life cycle
- Programmers face ever changing functional requirements
- But need to design software that is stable (i.e. does not change)
- **Software entities should be open for extension, but closed for modification** by B. Meyer

· 这是OOD的基石原则
· 所有软件在其生命周期中都会发生变化
· 程序员面对不断变化的功能需求
· 但需要设计稳定的软件(即不改变)
· 软件实体应该对扩展开放，但对修改关闭

# Open-Closed Principle

- 在OO条款
- 类应该对扩展开放，对修改关闭
- 换句话说，您应该能够在不修改类的情况下扩展它
- 开放扩展:类的行为可以扩展，以满足不断变化的需求
- 关闭修改:类本身不允许更改

- In OO terms
- **Classes should be open for extension, but closed for modification**
- In other words you should be able to **extend** a class **without modifying** it
- Open for extension: behaviour of classes can be extended to satisfy changing requirements
- Closed for modification: classes themselves are not allowed to change

# Open-Closed Principle

- At first this seems contradictory
- If you need a modified behaviour of a class just change that class
- This is however wrong
- The answer here is **abstraction**
- In OO it is possible to create abstractions with fixed design but  limitless behaviours

· 乍一看，这似乎有些矛盾
· 如果你需要修改一个类的行为，只需修改这个类
· 然而这是错误的
· 这里的答案是抽象
· 在OO中，可以创建具有固定设计但行为无限制的抽象

# Open-Closed Principle

- Abstractions through inheritance - changed behaviour through polymorphism 通过继承抽象——通过多态性改变行为

继承:抽象基类或接口

- Inheritance: abstract base classes or interfaces

您必须在设计早期决定系统的哪些部分将稍后扩展，哪些将保持不变

- You must decide early on in the design which parts of the system will  be expanded later and which will stay fixed

# Open-Closed Principle

- The design is extended by adding new code and classes by inheriting existing base classes
- Modifying the existing classes is not needed
- Completed and tested code is declared closed, and it is never modified

- 通过继承现有的基类来添加新的代码和类来扩展设计
- 不需要修改现有的类
- 已完成和测试的代码被声明为关闭，并且永远不会被修改

# Open-Closed Principle: Example

- We violated the OCP before
- To change the behaviour we changed the base class
- Both methods differ only in one statement: selection statement
- This is where we need to design for change
- We need to refactor

- 我们之前违反了OCP
- 为了改变行为，我们改变了基类
- 这两种方法只有一个语句不同:选择语句
- 这就是我们需要为改变而设计的地方
- 我们需要重构

# Open-Closed Principle: Example

```
public class Filter {
public Vector filterPublication(Collection
pubs,  Selector selector) {
List<Publication> filtered =  new
ArrayList<Publication>();  for (Publication
pub : pubs) {
if (selector.isPubSelected(pub))
{  filtered.add(pub);
}
}
return filtered;
}
}
```

# Open-Closed Principle: Example

```
public interface Selector {
  public boolean isPubSelected(Publication pub);
}
```

# Open-Closed Principle: Example

```
public class YearSelector implements Selector {  private final
   int year;

public YearSelector(int year) {  this.year = year;
}

public boolean isPubSelected(Publication pub) {  if
   (pub.getYearOfPublication() == year) {
return true;
}
return false;
}
}
```

# Open-Closed Principle: Example

```
public class TitleSelector implements Selector
  {  private final String title;

public TitleSelector(String title) {  this.title =
  title;
}

public boolean isPubSelected(Publication pub) {
if (pub.getTitleOfPublication().equals(title))
  {  return true;
}
return false;
}
}
```

# Open-Closed Principle

- Now filtering is open for extension, i.e. adding filters for author, pub  type
- Still filtering is closed for modification - no need to modify the Filter  class
- With the class being open for extension, it is highly reusable and can  be used to satisfy a ever changing list of requirements
- The class is extremely maintainable, as it never needs to change

· 现在过滤是开放的扩展，即添加过滤器的作者，发布类型
· 仍然过滤是关闭的修改-不需要修改过滤器类
· 由于类是开放扩展的，它是高度可重用的，可以用来满足不断变化的需求列表
· 类具有极强的可维护性，因为它从不需要更改

类似于goto语句，对程序破坏性大

- **instanceof** operator in Java checks whether an object is of a certain type

  Java中的instanceof操作符检查对象是否属于某种类型

- Typical use of this operator is in if-elseif-else or switch type statement

  该操作符的典型用法是在if-elseif-else或switch类型语句中

- Depending on the type of an object you initiate actions

  根据对象的类型，可以发起操作

# Disadvantages of instanceof

```
...
public void cookPizza(Pizza pizza) {
if (pizza instanceof ThinCrustPizza)
{   ((ThinCrustPizza)pizza).cookInWoodFireOve
n();
} else if (pizza instanceof PanPizza)
{   ((PanPizza)pizza).cookInGreasyPan();
}
}
...
```

Note: At least have an `else throw new IllegalArgumentException();`

# Disadvantages of instanceof

- This is a direct violation of OCP
- This class can not be extended without modifying it
- Suppose there is a new kind of pizza
- You need to extend the above class with a new elseif that handles the  new type

- 这直接违反了OCP
- 不修改类就不能扩展它
- 假设有一种新的披萨
- 您需要用一个处理新类型的新的elseif扩展上述类

# Disadvantages of instanceof

中国科学技术大学
University of Science and Technology of China

- You can always design your classes so that they do not need instanceof operator
- The key again is inheritance + polymorphism
- In certain cases you will need an extra design construct
- But it pays off because you get the benefits of the OCP

- 您总是可以设计您的类，使它们不需要instanceof操作符
- 关键还是继承+多态
- 在某些情况下，你将需要一个额外的设计构造
- 但它是值得的，因为你得到了OCP的好处

# Disadvantages of instanceof

```
abstract public Pizza {   public void cook()
  {
placeOnCookingSurface();
  placeInCookingDevice();
int cookTime = getCookTime();
  letItCook(cookTime);
  removeFromCookingDevice();
}
abstract public void
  placeInCookingDevice();
...
}
```

# Disadvantages of instanceof

```
...
public void cookPizza(Pizza pizza)
{   pizza.cook();
}
...
```

# Disadvantages of instanceof

- We applied an extra design construct here: Template Method
- `cook()` method defines a general cook algorithm
- It calls a number of template methods that are abstract
- Subclasses need to implement that behaviour by implementing template methods

- 我们在这里应用了一个额外的设计构造:Template方法
- cook()方法定义了一个通用的cook算法
- 它调用了许多抽象的模板方法
- 子类需要通过实现模板方法来实现这种行为

# Liskov Substitution Principle

# Design by contract

- Developed by B. Meyer
- Its a contract between a class and its clients (other classes that use it)
- Specifies rules that need to be satisfied by the class and by the clients
- Preconditions and postconditions for methods , invariants for the class

输入参数类型 返回参数类型

- 它是类和它的客户端(使用它的其他类)之间的契约
- 指定类和客户端需要满足的规则
- 方法的前置条件和后置条件，类的不变量

# Design by contract

- Example: Stack class
- Precondition for `push(x)` method: the stack is non-full
- Postcondition for `push(x)` method: x is on the top of the stack
- Invariants: stack size is non-negative, count is less than `max_size`, ...

# Design by contract

- In some languages you can formally specify such rules
- With predicate logic you can even check the correctness of such rules
- Certain approaches to generate code from such specifications
- In Java: **assert** keyword to check if rules are satisfied: Testing

- 在某些语言中，您可以正式地指定这些规则
- 使用谓词逻辑，你甚至可以检查这些规则的正确性
- 从这些规范中生成代码的某些方法
- 在Java中:assert关键字检查是否满足规则:测试

# Liskov Substitution Principle

Closely related with Design by Contract  Depends on preconditions and postconditions

契约设计与前提条件和后置条件密切相关

## Definition

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

如果对于每一个类型为S的对象o1都有一个类型为T的对象o2，这样，对于用T定义的所有程序P，当o1替换o2时，P的行为没有改变，那么S是T的子类型。

# Liskov Substitution Principle

- What does it mean?
- Anywhere you specify a base type, you should be able to use an  instance of a derived type.

这是什么意思?
•在指定基类型的任何地方，都应该能够使用派生类型的实例。

# Liskov Substitution Principle

- But with polymorphism we can do it anyway, right?
- That is true but here we are concerned with the program correctness
- We are concerned with the question what happens when subclasses  reimplement (override) methods
- The overridden methods need to satisfy rules specified by the superclass

- 但有了多态，我们无论如何都可以做到，对吧?
- 这是对的，但这里我们关心的是程序的正确性
- 我们关心的问题是，当子类重新实现(重载)方法时会发生什么
- 被重载的方法需要满足父类指定的规则

长方形
Width
Height

正方形
Side

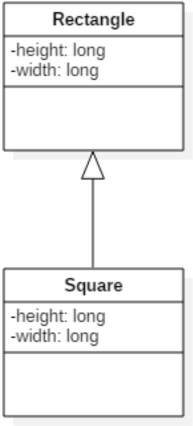
```
public class Rectangle {
  protected long width;
  protected long height;
  public void setWidth(long width) {
    this.width = width;
  }
  public long getWidth() {
    return this.width;
  }
  public void setHeight(long height) {
    this.height = height;
  }
  public long getHeight() {
    return this.height;
  }
}
```

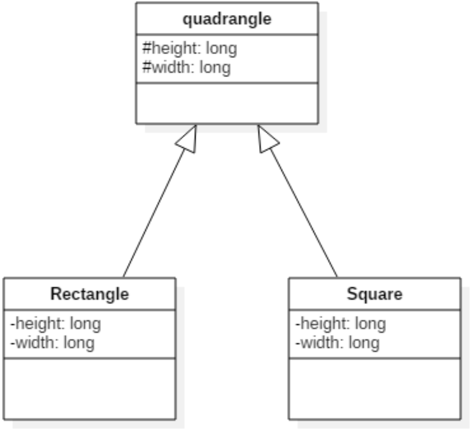
```
public class Square extends Rectangle {
  public void setWidth(long width) {
    this.height = width;
    this.width = width;
  }
  public long getWidth() {
    return width;
  }
  public void setHeight(long height) {
    this.height = height;
    this.width = height;
  }
  public long getHeight() {
    return height;
  }
}
```

```
public class SmartTest{   /**     * 长方形的长不断的
增加直到超过宽     * @param r     */
 public void resize(Rectangle r)   {
  while (r.getHeight() <= r.getWidth() ) {
      r.setHeight(r.getHeight() + 1);
  }
 }
}
```

调用SmartTest类中的resize方法时，长方形是可以的，但是正方形就会一直增大，直到溢出。按照我们的里氏替换原则，父类可以的地方，换成子类一定也可以，所以上边的这个例子是不符合里氏替换原则的。

- 问题由来
  - 有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。
- 解决方案
  - 当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。


Rectangle
-height: long
-width: long
Square
-height: long
-width: long


quadrangle
#height: long
#width: long
Rectangle
-height: long
-width: long
Square
-height: long
-width: long

```
/** * 定义一个四边形类，只有get方法没有set方法 *
public abstract class Quadrangle {
  protected abstract long getWidth();
  protected abstract long getHeight();
}
```

```
public class Rectangle extends Quadrangle {
 private long width;
 private long height;
 public void setWidth(long width) {
  this.width = width;
 }
 public long getWidth() {
  return this.width;
 }
 public void setHeight(long height) {
  this.height = height;
 }
 public long getHeight() {
  return this.height;
 }
}
```

```
public class Square extends Quadrangle {
  private long width;
  private long height;
  public void setWidth(long width) {
    this.height = width;
    this.width = width;
  }
  public long getWidth() {
    return width;
  }
  public void setHeight(long height) {
    this.height = height;
    this.width = height;
  }
  public long getHeight() {
     return height;
  }
```

# 里氏替换原则

- 子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：
  - 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
  - 子类中可以增加自己特有的方法。
  - 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
  - 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

```
public class A{
    public int func1(int a, int b){
        return a-b;
    }
}
```

```
public class B extends A{
  public int func1(int a, int b){
    return a+b;
  }
  public int func2(int a, int b){
    return func1(a,b)+100;
  }
}
```

```
public class Client{
  public static void main(String[] args){
    B b = new B();
    System.out.println("100-50="+b.func1(100, 50));
    System.out.println("100-80="+b.func1(100, 80));
    System.out.println("100+20+100="+b.func2(100, 20));
  }
}
```

100-50=150
100-80=180
100+20+100=220

原本运行正常的相减功能发生了错误。原因是类B在给方法起名时重写了父类的方法，造成所有运行相减功能的代码全部调用了类B重写后的方法，造成原本运行正常的功能出现了错误。

# Single Responsibility Principle

# Single Responsibility Principle

- An entity (an object, a class) should have only a **single responsibility**
- A responsibility is in fact a **reason to change**
- An entity should have only single reason to change

- 一个实体(一个对象，一个类)应该只有一个职责
- 责任实际上是改变的理由
- 一个实体应该只有一个改变的理由

# Single Responsibility Principle

```
public class Student {
  ...
  public void readStudent(int id) {
    // read student data from a database
    ...
  }
  public String printHTML() {
    // print student data in a nice HTML format
    ...
  }
}
```

# Single Responsibility Principle

- Student class has two responsibilities: reading from a database and printing HTML
- Two reasons for change: if a table is modified and if another HTML format is needed
- We need to separate responsibilities: reading and printing
- This is also very often called: **separation of concerns**

- 学生类有两个职责:从数据库读取和打印HTML
- 更改的两个原因:一个表被修改了，另一个是需要HTML格式
- 我们需要分开责任: read和print
- 这也经常被称为:关注点分离

# Single Responsibility Principle

```
public class Student {
  ...
  public void readStudent(int id) {
    // read student data from a database
    ...
  }
}
```

# Single Responsibility Principle

```
public class StudentPrinter {
  ...
  public String printStudent(Student student) {
    // print in a nice HTML format
    ...
  }
}
```

# Law of Demeter

最少知识原则

# Law of Demeter

- A method of an object should only invoke methods of:
- Itself
- Its parameters
- Objects it creates
- Its members
- Methods should not invoke methods of other objects’ members

一个对象的方法应该只调用以下方法:
· 本身
· 它的参数
· 创建的对象
· 其成员
方法不应该调用其他对象成员的方法

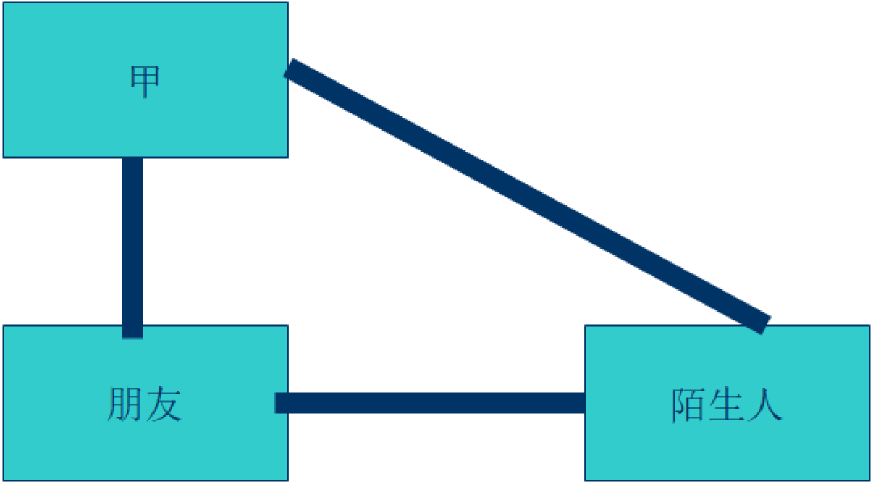
# 和陌生人说话

```
public class Jia{
    public void play(Friend friend){
        friend.play();
    }
    public void play(Stranger stranger) {
        stranger.play();
    }
}
```

甲和朋友认识，朋友和陌生人认识，而甲和陌生人不认识，这时甲可以直接和朋友说话，朋友可以直接和陌生人说话，而如果甲想和陌生人说话，就必须通过朋友

```
//朋友
 public class Friend {
     public void play(){
         System.out.println("朋友");
     }
 }
```

```
//陌生人
 public class Stranger {
     public void play(){
         System.out.println("陌生人");
     }
 }
```



甲

朋友

陌生人

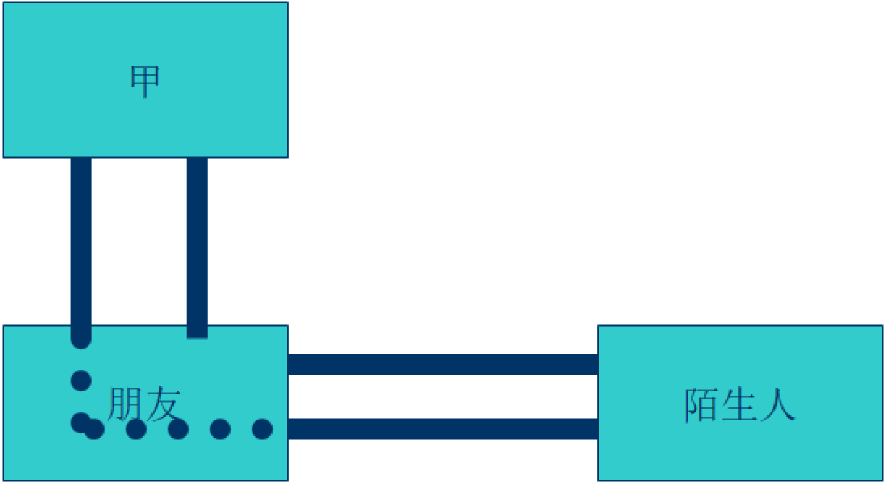
```
//甲
 public class Jia{
     public void play(Friend friend){
         friend.play();
         Stranger stranger = friend.getStranger();
         stranger.play();
     }
 }
```

- //朋友

```
public class Friend {
    public void play(){
        System.out.println(朋友");
    }
   public Stranger getStranger() {
         return new Stranger();
    }
}
```

```
//陌生人
 public class Stranger {
     public void play(){
         System.out.println("陌生人");
     }
 }
```

因为甲中包含的陌生人的引用，甲还是和陌生人直接关联上了，仍然不符合迪米特法则，我们要的是甲和陌生人没有任何直接关系



甲

朋友

陌生人

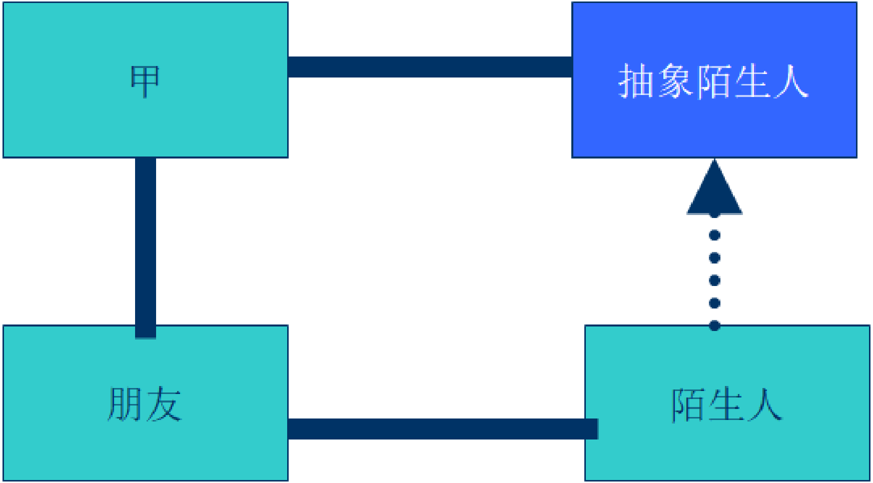
- //甲

```
public class Jia{
    private Friend friend;
    public Friend getFriend() {
        return friend;
    }
    public void setFriend(Friend friend) {
        this.friend = friend;
    }
    public void play(Friend friend){
        friend.play();
    }
    public void playWithStranger (Friend friend){
        friend.playWithStranger ();
    }
```

```
//朋友
 public class Friend {
     public void play(){
         System.out.println("朋友");
     }
     public void playWithStranger() {
         Stranger stranger = new Stranger();
         stranger.play();
     }
 }
```

```
//陌生人
public class Stranger {
    public void play(){
        System.out.println("陌生人");
    }
 }
```

# IOC mixed



```
//甲
public class Jia{
    private Friend friend;
    private Stranger stranger;
    public Stranger getStranger() {
        return stranger;
    }
    public void setStranger(Stranger stranger) {
        this.stranger = stranger;
    }
    public Friend getFriend() {
        return friend;
    }
    public void setFriend(Friend friend) {
        this.friend = friend;
    }
    public void play() {
        System.out.println("someone play");
```

```
//朋友
public class Friend {
    public void play(){
        System.out.println("朋友");
    }
 }
```

```
//陌生人抽象类
public abstract class Stranger {
    public abstract void play();
}
```

```
//具体陌生人
 public class StrangerA extends Stranger {
    public void play() {
       System.out.println("陌生人");
    }
 }
```

迪米特法则应用实例
1.外观模式

2.中介者模式

# Law of Demeter

```
public class StudentPrinter {
  ...
  public String printStudent(Student student) {
    ...
    // violation
    String course = student.getCourse(i).getName();
  }
}
```

# Law of Demeter

```
public class Student {
...
public String getCourseName(int i)
{   return courses[i].getName();
}
}
```

# Law of Demeter

- Chaining of method calls is dangerous because you depend on all parts of that chain 方法调用的链接是危险的，因为您依赖于该链接的所有部分
- Provide a method in Student class that gets you names of the courses 在Student类中提供一个获取课程名称的方法
- That method delegates to the Course class 该方法委托给Course类
- If the Course class changes only Student class needs to be updated not StudentPrinter 如果课程类更改，只需要更新Student类，而不需要更新StudentPrinter
- Consequence for composition/delegation: only delegate to own members! 组成/委托的后果:只委托给自己的成员!

# Divide and conquer

# Divide and conquer

- Trying to deal with something big all at once is normally much more difficult than dealing with a series of smaller, managable, understandable things
- (hence the iterative approach to software development)
  - A software engineer/software developer can **specialize**.
    - Specialize in network, distribution, database, algorithms, searching / sorting techniques…
  - Individual **components smaller**, **easier to understand**.
  - **Parts** can be **replaced** or **changed** without having to replace or extensively change other parts.

# Ways of dividing a software system

- A distributed system is divided up into clients and servers
- A system is divided up into subsystems
- A subsystem can be divided up into one or more packages
- A package is divided up into classes
- A class is divided up into methods

# High <u>Cohesion</u>

# Increase (High) Cohesion where possible

- Divide and Conquer says split things up. Smaller parts, easier to grasp. "分而治之"的意思是把事情分开。零件越小，越容易掌握

- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps other things out!
  - Makes the system as a whole easier to understand / change
  - Type of cohesion:
    - Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

如果子系统或模块将相互关联的事物放在一起，并将其他事物拒之门外，那么它就具有高内聚性!

-使系统作为一个整体更容易理解/改变

-凝聚力类型:

•功能、层次、沟通、顺序、程序、时间、效用

# Functional Cohesion

- Achieved when all the code that computes a particular result is kept together - and everything else is kept out
  - i.e. when a module only performs a *single* computation, and returns a result, *without having side-effects*.
    - No changes to anything but the computation
    - Normally implemented via parameters
    - Can call other methods, if cohesion is preserved.
      - (Recall: Call by Value; Call by Reference… as examples)
      - Avoid things like common, global data, more
  - Benefits to the system:
    - Easier to **understand**
    - More **reusable**
    - Easier to **replace**
    - Example: pass an array of integers; sort the array and return sorted array. Can change algorithms if interface remains unchanged…

当计算特定结果的所有代码都放在一起时——其他的都被排除在外

-例如，当一个模块只执行一个单一的计算，并返回一个结果，没有副作用。

•除了计算没有任何改变

•通常通过参数实现

•如果保持内聚，可以调用其他方法。

-(召回:按值调用;通过引用调用...作为例子)

-避免使用常见的、全球性的数据等等

-对系统的好处:

•更容易理解

•更可重用

•更容易替换

·示例:传递一个整数数组;对数组进行排序并返回排序后的数组。如果界面保持不变，可以改变算法吗

# Layer cohesion

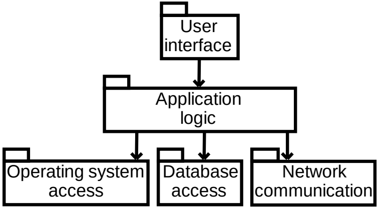
- All facilities for providing or accessing a set of related services are kept together, and everything else is kept out
  - – The layers should form a hierarchy
    - (Layers – presentation (interface); business (domain) logic; application logic; technical services…)
    - Higher layers can access services of lower layers,
    - Lower layers **do not** access higher layers
  - – Will talk about architectural layers a great deal very soon…

  - – The set of procedures through which a layer provides its services is the *application programming interface (API)*
  - – *Specification of API says how to use it.*
  - – ➔ You can replace a layer without having any impact on the other layers because you **know** that it does not access upper layers.

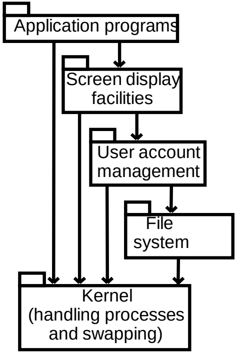
# Example of the use of layers

Ours will be similar to this  in some ways, but different in other  ways

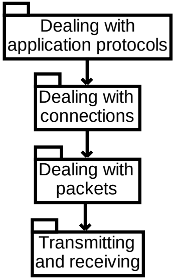
**Sometimes we have a business services and then a domain (more general) layer… Will have a middleware layer often!**



a) Typical layers in an application program


Application programs
Screen display facilities
User account management
File system
Kernel (handling processes and swapping)

b) Typical layers in an operating system


Dealing with application protocols
Dealing with connections
Dealing with packets
Transmitting and receiving

c) Simplified view of layers in a communication system

**Examples:  services for computations;  transmissions of messages;  storage of data;  managing security, interacting with users;  accessing the operating system;  interacting with hardware, and more**

# Communicational cohesion

- All the modules that access or “manipulate certain data” are kept together (e.g. in the same class) - and everything else is kept out
  - A class would have good communicational cohesion
    - if **all** the system’s facilities for storing and manipulating its data are contained in this class.
    - if the class does not do anything other than manage its data.
  - Main advantage: When you need to make **changes** to the data, you find all the code in one place
  - →Keep methods **where the data is**, if possible.
  - Talk about this extensively in Data Structures course!!

# Other measures of Cohesion

- Sequential Cohesion

- Procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out
  - You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

- Procedural Cohesion

- Keep together several procedures that are used one after another
  - Even if one does not necessarily provide input to the next. Weaker than sequential.

- Temporal Cohesion

- Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out
  - For example, placing together the code used during system start-up or initialization.
  - Weaker than procedural cohesion.

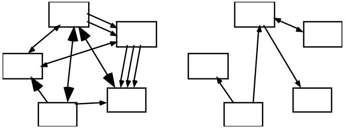
- Utility Cohesion

- When related utilities which cannot be logically placed in other cohesive units are kept together
  - A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
  - For example, the **`java.lang.Math`** class. ←

# Low Coupling

# Reduce (**Low**) **Coupling** where possible

- *Coupling* occurs when there are *interdependencies* between one module and another
  - When **interdependencies** exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it difficult to see at a glance how some component works.
  - Type of coupling: (in **decreasing** order of avoidance!)
    - **Content**, **Common**, **Control**, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



# Content Coupling: The Worst of the Bad!

- Occurs when one component *surreptitiously* modifies data (or instructions!) that is/are *internal* to another component
  - To reduce content coupling you should therefore *encapsulate* all instance variables
    - declare them **`private`**
    - and provide ***get*** and ***set*** methods
  - A worse form of content coupling occurs when you **directly modify** an instance variable from outside the object. 当从对象外部直接修改实例变量时，会出现更糟糕的内容耦合形式
  - Discuss: how easy it is to do this and how/why it has been done in the past! (especially in non-object-oriented systems)
    - Assembler;  'Alter' verb in Cobol

# Common Coupling – also Avoid where possible and practical!!!

公共耦合——也尽量避免使用!!

在使用全局变量时发生

- Occurs whenever you use a global variable
  - All the **components** using global variables become **coupled** to each other 所有使用全局变量的组件将相互耦合
  - Can be acceptable for creating global variables that represent system-wide default values 创建代表系统默认值的全局变量是否可以接受
  - Clearly, when a value is changed, it may be very difficult to trace the source of the change! 显然，当一个值发生更改时，跟踪更改的来源可能会非常困难

# Control Coupling – not as bad, but still is pretty strong coupling! Avoid if you are able to.

- Occurs when one procedure calls another using a 'flag' or 'command' that explicitly controls what the second procedure does (passing a switch….)
  - To make a change you have to change both the calling and called method; that is, to avoid the flags…
  - The use of polymorphic operations is normally the best way to avoid control coupling
  - One way to reduce the control coupling could be to have a *look-up table*
    - **commands** are then **mapped to a method** that should be called when that command is issued

# Example of control coupling

```
public routineX(String command)
{
    if (command.equals("drawCircle")
    {
        drawCircle();
    }
    else
    {
        drawRectangle();
    }
}
```

See?  Flag is passed (command) whose value is used to control flow!

Can be handled better through polymorphism…

# Other Forms of Coupling:

- Stamp Coupling
- Data Coupling
- Routine-call Coupling
- Type Use Coupling
- Inclusion or import Coupling
- External Coupling

# Enterprise Architecture Principles

# Enterprise Architecture Principles

## Reuse

- Existing components must be reused where applicable
- Instead of developing them from scratch
- In practice there is often a trade-off, as the existing component might not exactly match the requirements
- Optionally, extend the existing component

# Enterprise Architecture Principles

## Buy rather than build

- If a component exist (commercially or open-source) that match the requirements, it should rather be used
- Unless there are strategical (e.g. competition) or technical issues
- Motivation: higher quality of existing solutions, might provide extra functionality
- Often, it is cheaper to use existing solutions instead of developing them

# Enterprise Architecture Principles

## Single point of view

- If there are multiple sources of data (e.g. file-system and database system)
- Develop components to make them appear as a single source
- Motivation: will lead to more simple components (that use the source), logic to combine the sources is bundled in a single place