哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

9 Construction for Reuse
面向复用的软件构造技术

Wang Zhongjie
rainy@hit.edu.cn

June 16, 2021

Objective of this lecture

- **Advantages and disadvantages of software reuse**
- **Construction for/with reuse**
- **Characteristics of generic reusable components**
- **Methods of developing portable application systems**

前几次课介绍了软件构造的核心理论（ADT）与技术（OOP），其核心是保证代码质量、提高代码安全性。

本次课面向一个重要的外部质量指标：可复用性——如何构造出可在不同应用中重复使用的软件模块/API？

首先探讨可复用的软件应该“长什么样”，然后学习“如何构造”

Outline

- **What is software reuse?**
- **How to measure “reusability”?**
- **Levels and morphology of reusable components**
  - Source code level reuse 源代码级别的复用
  - Module-level reuse: class/interface 模块级别的复用：类/抽象类/接口
  - Library-level: API/package 库级别的复用：API/包
  - System-level reuse: framework 系统级别的复用：框架

Outline

- **Designing reusable classes 设计可复用的类**
  - Inheritance and overriding 继承与重写
  - Overloading 重载
  - Parametric polymorphism and generic programming 参数多态与泛型编程
  - Behavioral subtyping and Liskov Substitution Principle (LSP) 行为子类型与Liskov替换原则
  - Composition and delegation 组合与委托

- **Designing system-level reusable libraries and frameworks 设计可复用库与框架**
  - API and Library
  - Framework
  - Java Collections Framework (an example)

从类、API、框架三个层面学习如何设计可复用软件实体的具体技术

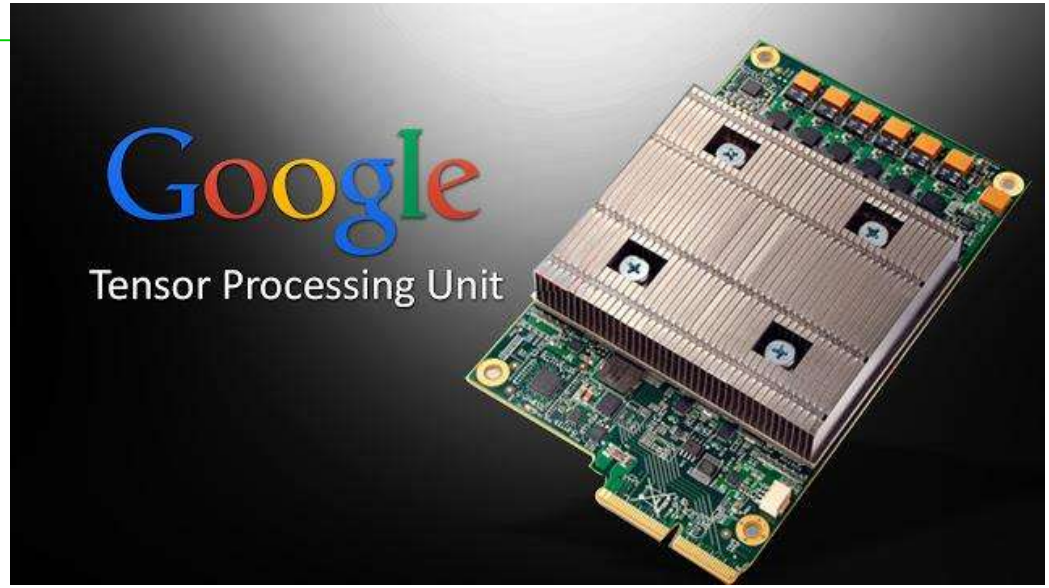
Reading

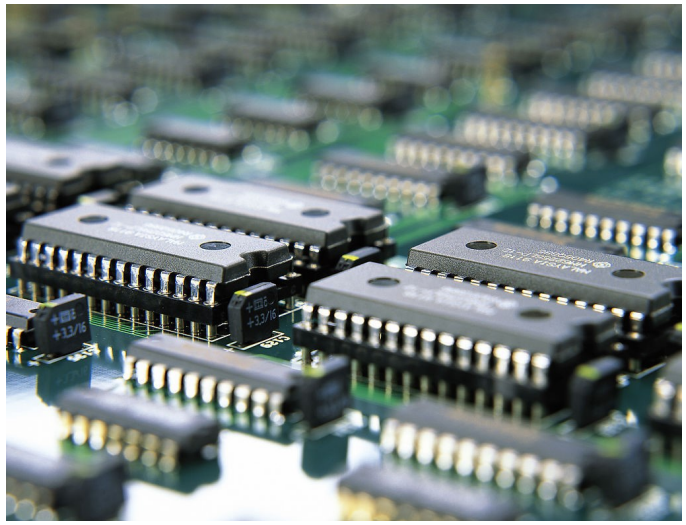
- **CMU 17-214：Oct 10、Oct 15、Oct 17、Oct 22**
- **Java编程思想：第14、15章**
- **Effective Java：第5章**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1 What is Software Reuse?

Hardware is reused inherently



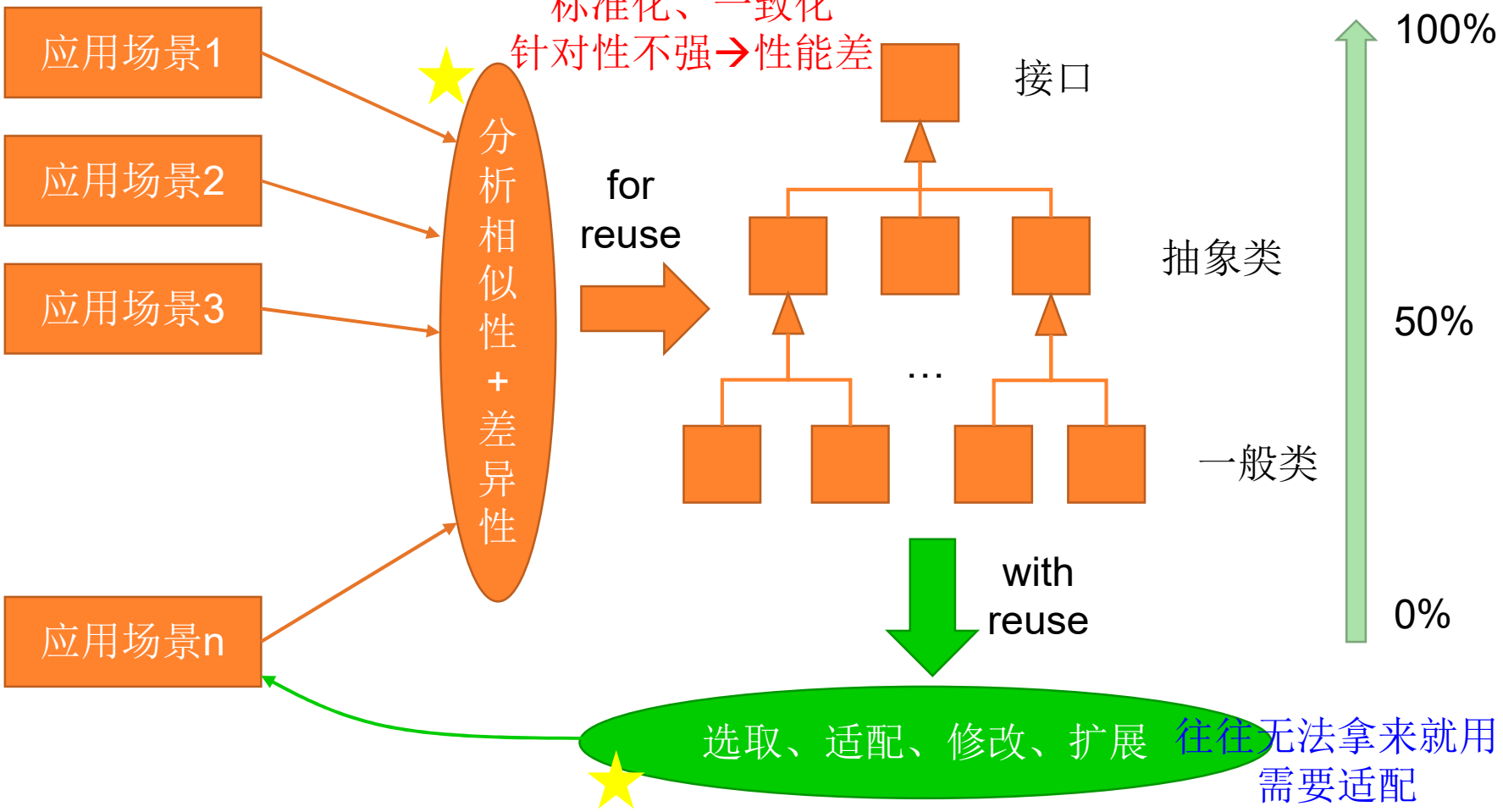


Software reuse

- **Software reuse** is the process of implementing or updating software systems using existing software components.
- **Two perspectives of software reuse**
  - Creation: creating reusable resources in a systematic way (programming for reuse 面向复用编程：开发出可复用的软件)
  - Use: reusing resources as building blocks for creating new systems (programming with reuse 基于复用编程：利用已有的可复用软件搭建应用系统)
- **Why reuse?**
  - “The drive to create reusable rather than transitory artifacts has aesthetic and intellectual as well as economic motivations and is part of man’s desire for immortality.
  - It distinguishes man from other creatures and civilized from primitive societies” (Wegner, 1989).

Programming for/with reuse

很大的适应性
降低成本和开发时间
充分的测试→高可靠
标准化、一致化
针对性不强→性能差



往往无法拿来就用
需要适配
可靠性建立在外部基础上

Recall Lab2

- 在**Lab2**中：你开发了一个基于泛型的抽象接口`Graph<L>`，定义了支持图结构的**ADT**
- 针对该**ADT**，用两种不同的**Rep**，开发了两个不同的实现`ConcreteVertexGraph<L>`和`ConcreteEdgeGraph<L>`

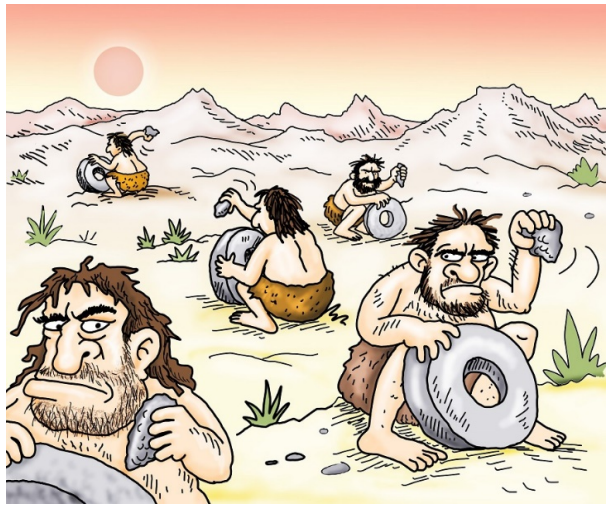
Programming for reuse
面向复用编程：开发出可复用的软件

- 进而，你利用该**ADT**及其两个实现，完成了两个应用的开发：
  - Poetic Walks
  - Friendship Social Network

Programming with reuse
基于复用编程：利用已有的可复用软件搭建应用系统

Why reuse?

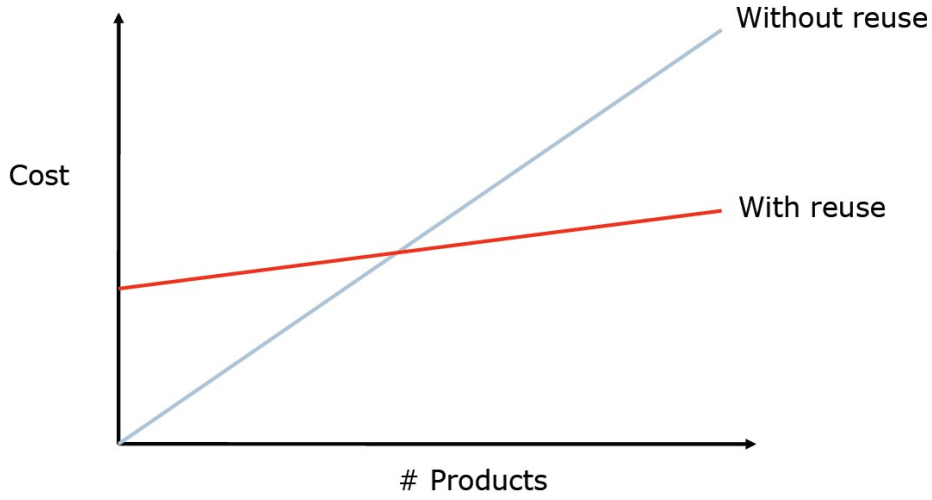
- **Reuse is cost-effective and with timeliness** 降低成本和开发时间
  - Increases software productivity by shortening software production cycle time (software developed faster and with fewer people)
  - Does not waste resources to needlessly "reinvent-the-wheel"
  - Reduces cost in maintenance (better quality, more reliable and efficient software can be produced)



- **Reuse produces reliable software** 经过充分测试，可靠、稳定
  - Reusing functionality that has been around for a while and is debugged is a foundation for building on stable subsystems
- **Reuse yields standardization** 标准化，在不同应用中保持一致
  - Reuse of GUI libraries produces common look-and-feel in applications.
  - Consistency with regular, coherent design.

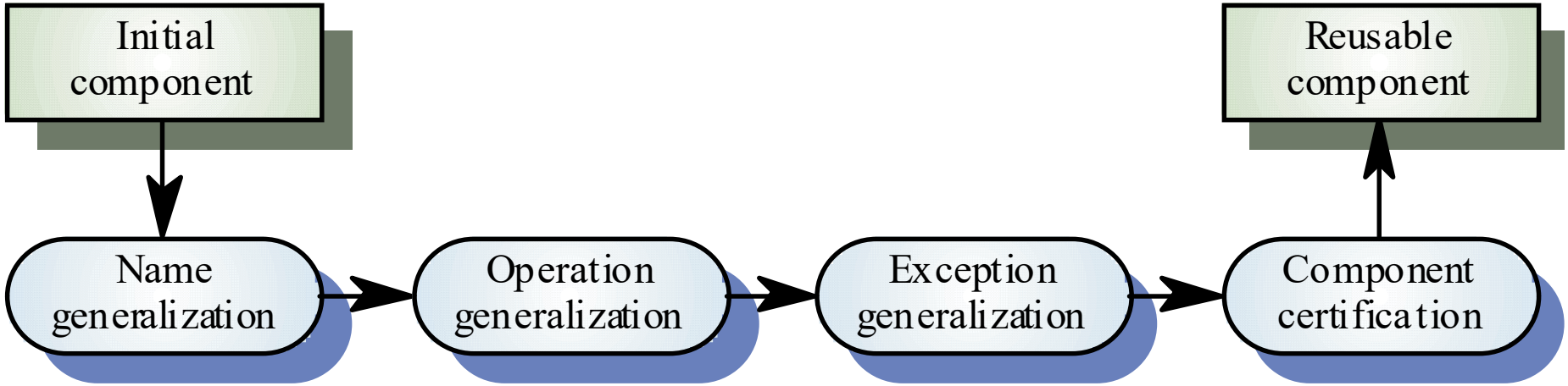
Reuse costs

- **Reusable components** should be designed and built in a *clearly* defined, *open* way, with *concise* interface specifications, *understandable* documentation, and an eye towards *future use*. （清晰的定义、开放模式、简洁的接口、易于理解的文档）做到这些，需要代价
- **Reuse is costly:** it involves spans organizational, technical, and process changes, as well as the cost of tools to support those changes, and the cost of training people on the new tools and changes. 不仅program for reuse代价高，program with reuse代价也高



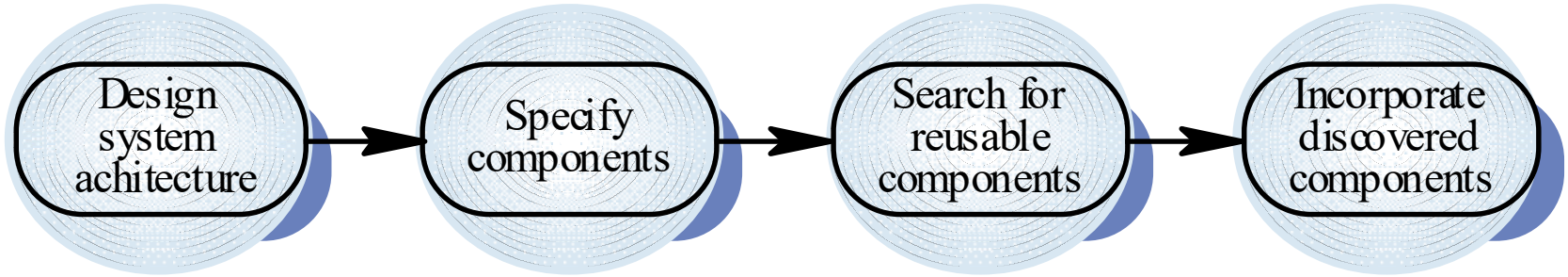
Development for reuse: 开发可复用的软件

- **The development cost of reusable components is higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost.** 开发成本高于一般软件的成本：要有足够高的适应性
- **Generic components may be less space-efficient and may have longer execution times than their specific equivalents.** 性能差些：针对更普适场景，缺少足够的针对性



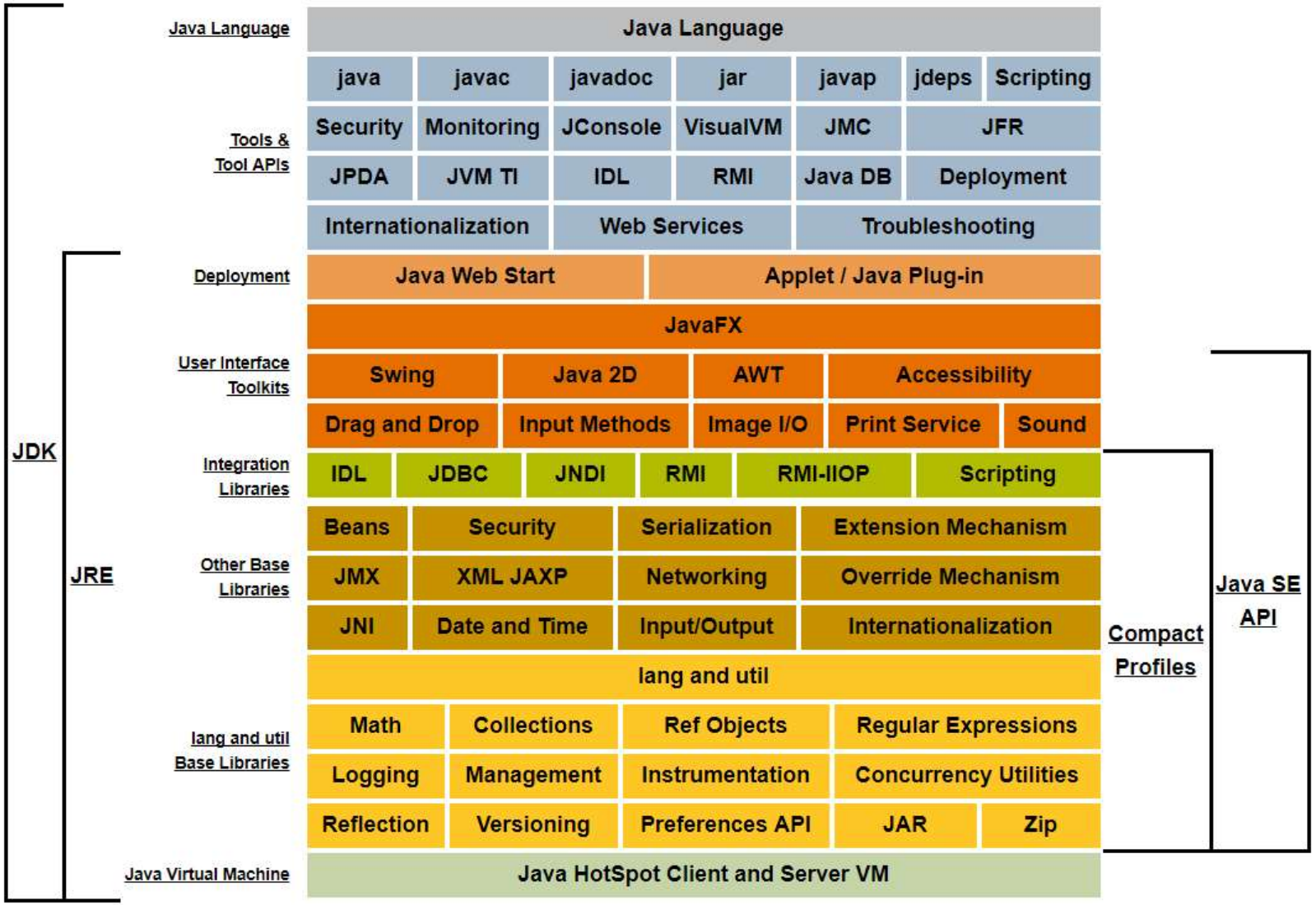
Development with reuse: 使用已有软件进行开发

- **Component management tools, such as repositories, for architectures, designs, documentation, and code must be developed and maintained.** 可复用软件库，对其进行有效的管理



- **A key issue: adaptation** 往往无法拿来就用，需要适配
  - Extra functionality may have to be added to a component. When this has been added, the new component may be made available for reuse.
  - Unneeded functionality may be removed from a component to improve its performance or reduce its space requirements
  - The implementation of some component operations may have to be modified.

Reusable Libraries and APIs in JDK



Rich third-party libraries and APIs in Java

Central Repository

http://central.maven.org/maven2/

URL	http://central.maven.org/maven2/
Jars	2,467,181 indexed jars

Published Jars by Year

2018	31,819
2017	709,421
2016	539,245
2015	362,410
2014	229,885
2013	170,787
2012	126,711
2011	106,285
2010	86,191
2009	49,537
2008	25,685
2007	12,478
2006	7,563
2005	9,067
2004	1

Home » log4j » log4j » 1.2.17

Apache Log4j » 1.2.17

Apache Log4j 1.2

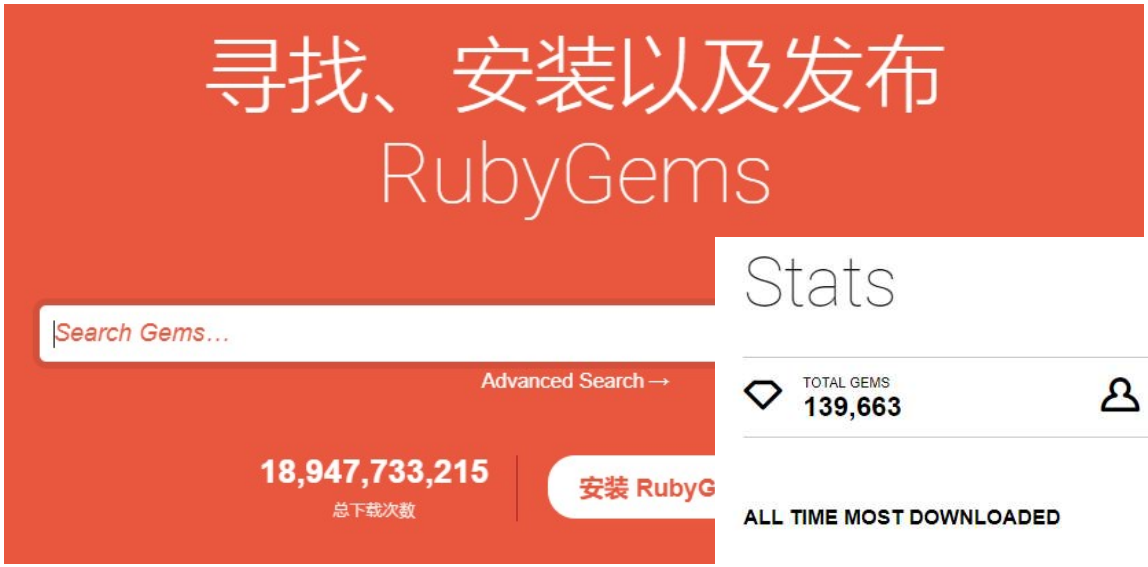
License	Apache 2.0
Categories	Logging Frameworks
Organization	Apache Software Foundation
HomePage	http://logging.apache.org/log4j/1.2/
Date	(May 26, 2012)
Files	pom (21 KB) bundle (478 KB) View All
Repositories	Central Apache Releases Redhat GA Sonatype Releases Spring Plugins
Used By	11,493 artifacts

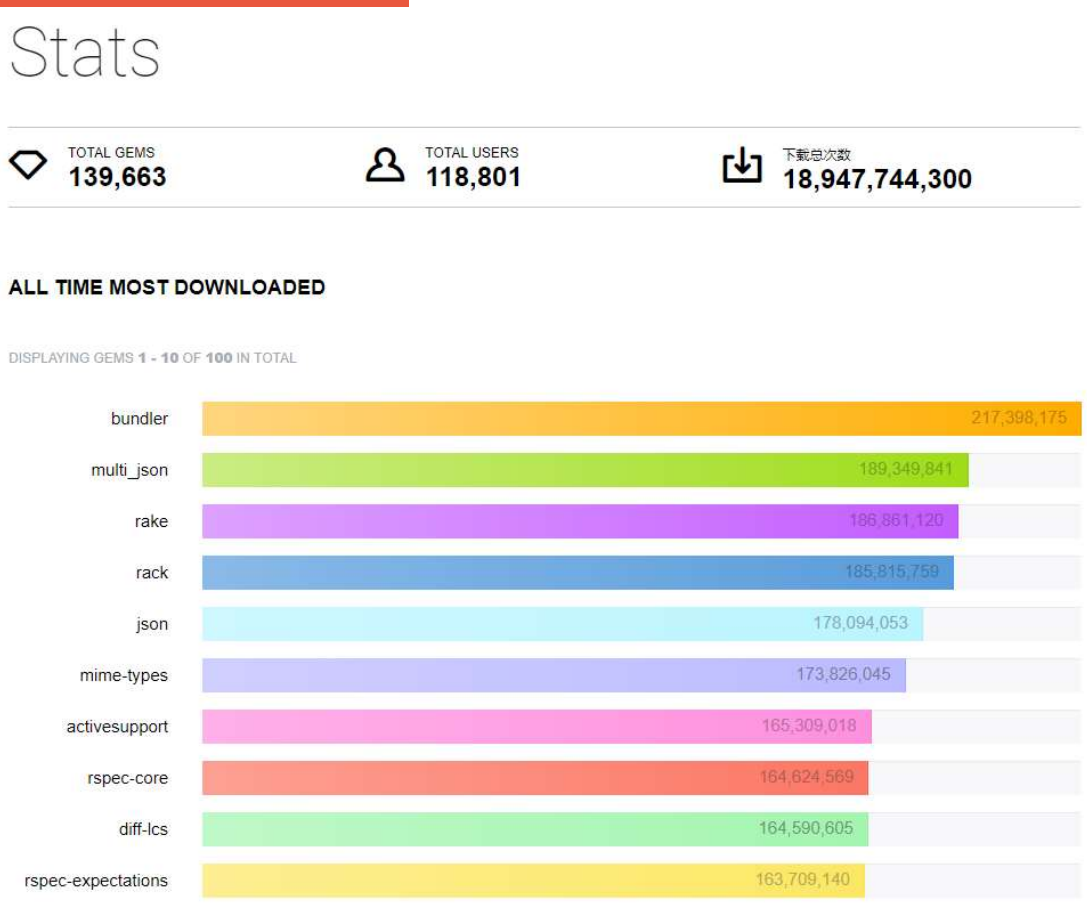
Maven | Gradle | SBT | Ivy | Grape | Leiningen | Buildr

```
<!— https://mvnrepository.com/artifact/log4j/log4j —>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

☑ Include comment with link to declaration

Rich RubyGems





哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2 How to measure “reusability”?

Measure resuability

- **How frequently can a software asset be reused in different application scenarios?** 复用的机会有多频繁？复用的场合有多少？
  - The more chance an asset is used, the higher reusability it has.
  - Write once, reuse multiple times.

- **How much are paid for reusing this asset?** 复用的代价有多大？
  - Cost to buy the asset and other mandatory libraries 搜索、获取
  - Cost for adapting and extending it 适配、扩展
  - Cost for instantiating it 实例化
  - Cost for changing other parts of the system that interact with it 与软件其他部分的互连的难度

Reusability

- **Reusability implies some explicit management of build, packaging, distribution, installation, configuration, deployment, maintenance and upgrade issues.**
- **A software asset with high reusability should:**
  - Brief (small size) and Simple (low complexity) 小、简单
  - Portable and Standard Compliance 与标准兼容
  - Adaptable and Flexible 灵活可变
  - Extensibility 可扩展
  - Generic and Parameterization 泛型、参数化
  - Modularity 模块化
  - Localization of volatile (changeable) design assumptions 变化的局部性
  - Stability under changing requirements 稳定
  - Rich documentation 丰富的文档和帮助

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

3 Levels and morphology 形态学of reusable components

复用模块的层次和形态学模式

Levels of Reuse

- **A reusable component may be code** 最主要的复用是在代码层面
  - Most prevalent: what most programmers relate with reuse

- **But benefits result from a broader and higher-level view of what can be reused.** 但软件构造过程中的任何实体都可能被复用
  - Requirements 需求
  - Design and specifications 设计/规约spec
  - Data 数据
  - Test cases 测试用例
  - Documentation 文档

What we concern in this lecture

- **Source code level: methods, statements, etc**
- **Module level: class and interface**
- **Library level: API**
  - Java Library, `.jar`
- **Architecture level: framework 框架**

Types of Code Reuse

- **White box reuse** 白盒复用：源代码可见，可修改和扩展
  - Reuse of code when code itself is available. Usually requires some kind of modification or adaptation 复制已有代码到正在开发的系统，进行修改
  - Pro: You can customize the module to fit the specific situation, this allows reuse in more situations 可定制化程度高
  - Con: You now own the customized result, so it adds to your code complexity. You requires intrinsic knowledge on component internals. 对其修改增加了软件的复杂度，且需要对其内部充分的了解
- **Black box reuse** 黑盒复用：源代码不可见，不能修改
  - Reuse in the form of combining existing code by providing some “glue”, but without having to change the code itself - usually because you do not have access to the code 只能通过API接口来使用，无法修改代码
  - Pro: Simplicity and Cleanliness 简单，清晰
  - Con: Many times it is just not possible 适应性差些

Formats for reusable component distribution

- **Forms:**
  - Source code
  - Package such as .jar, .gem, .dll,

以及最重要的：自己日积月累出来的可复用代码库

- **Sources of reusable software components:**
  - Internal (corporate) code libraries 组织的内部代码库（Guava）
  - Third party libraries 第三方提供的库（Apache）
  - Built-in language libraries 语言自身提供的库（JDK）
  - Code samples from tutorials, examples, books, etc. 代码示例
  - Local code guru or knowledgeable colleague 来自同事
  - Existing system code 已有系统内的代码
  - Open source products (be sure to follow any licensing agreements) 开源软件的代码

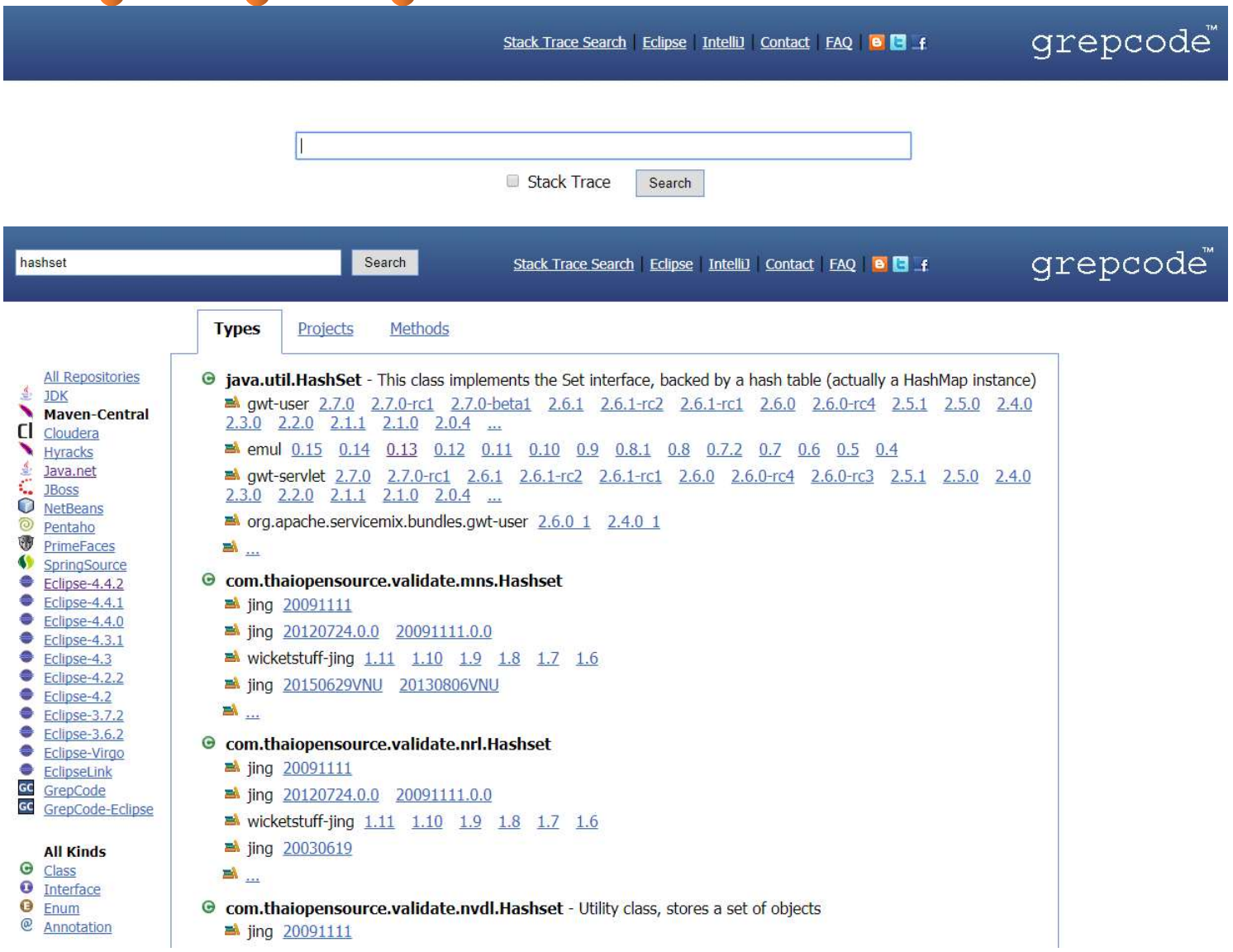
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(1) Source code reuse

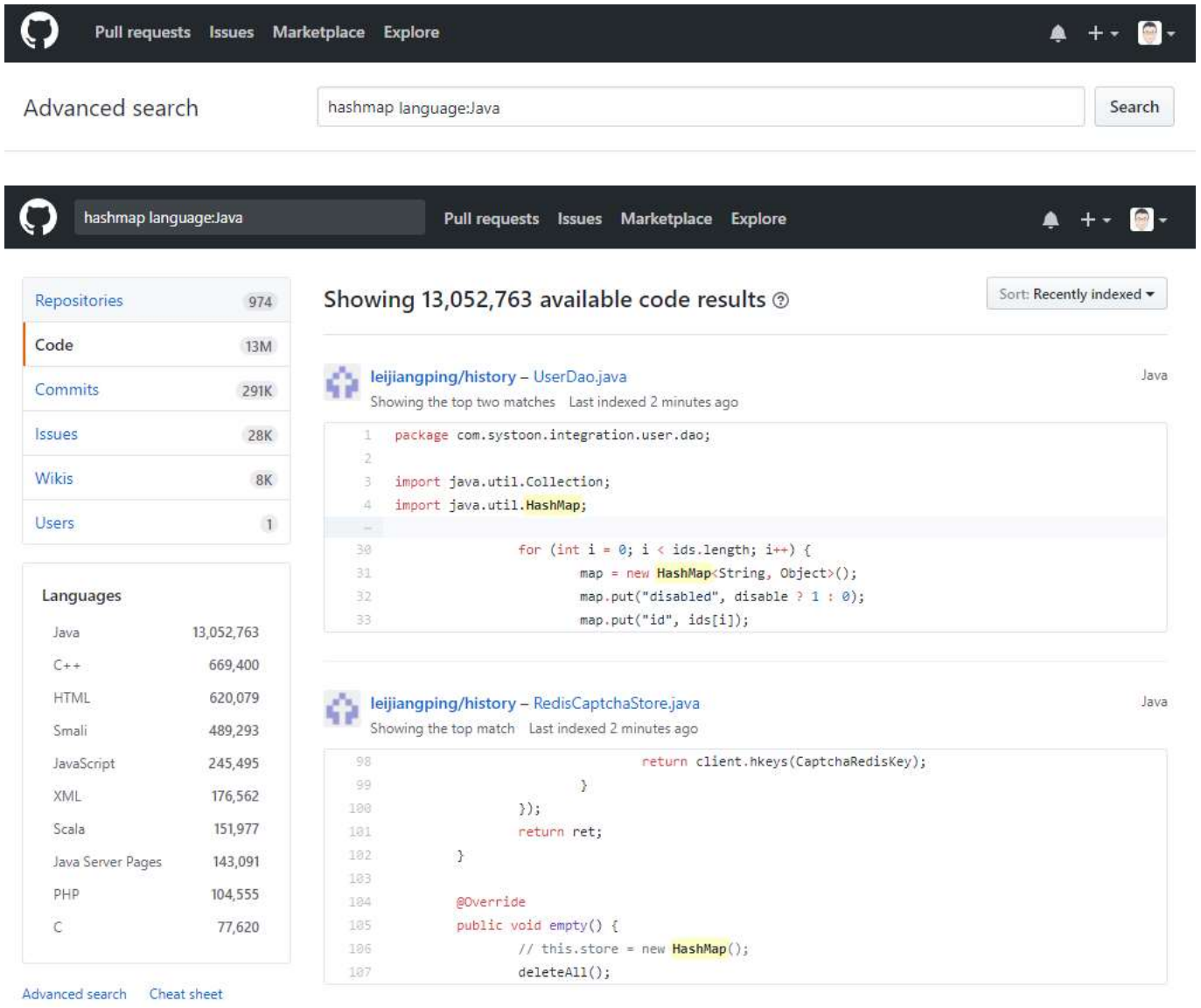
Reusing Code – Lowest Level

- **Copy/paste parts/all into your program**
- **Maintenance problem**
  - Need to correct code in multiple places
  - Too much code to work with (lots of versions) 在多种情况下需要修改代码
- **High risk of error during process**
- **May require knowledge about how the used software works**
- **Requires access to source code**
- 相关研究**1**：如何从互联网上快速找到需要的代码片段？
- 反向研究：如何从源代码中检测出克隆代码**(clone code)**？

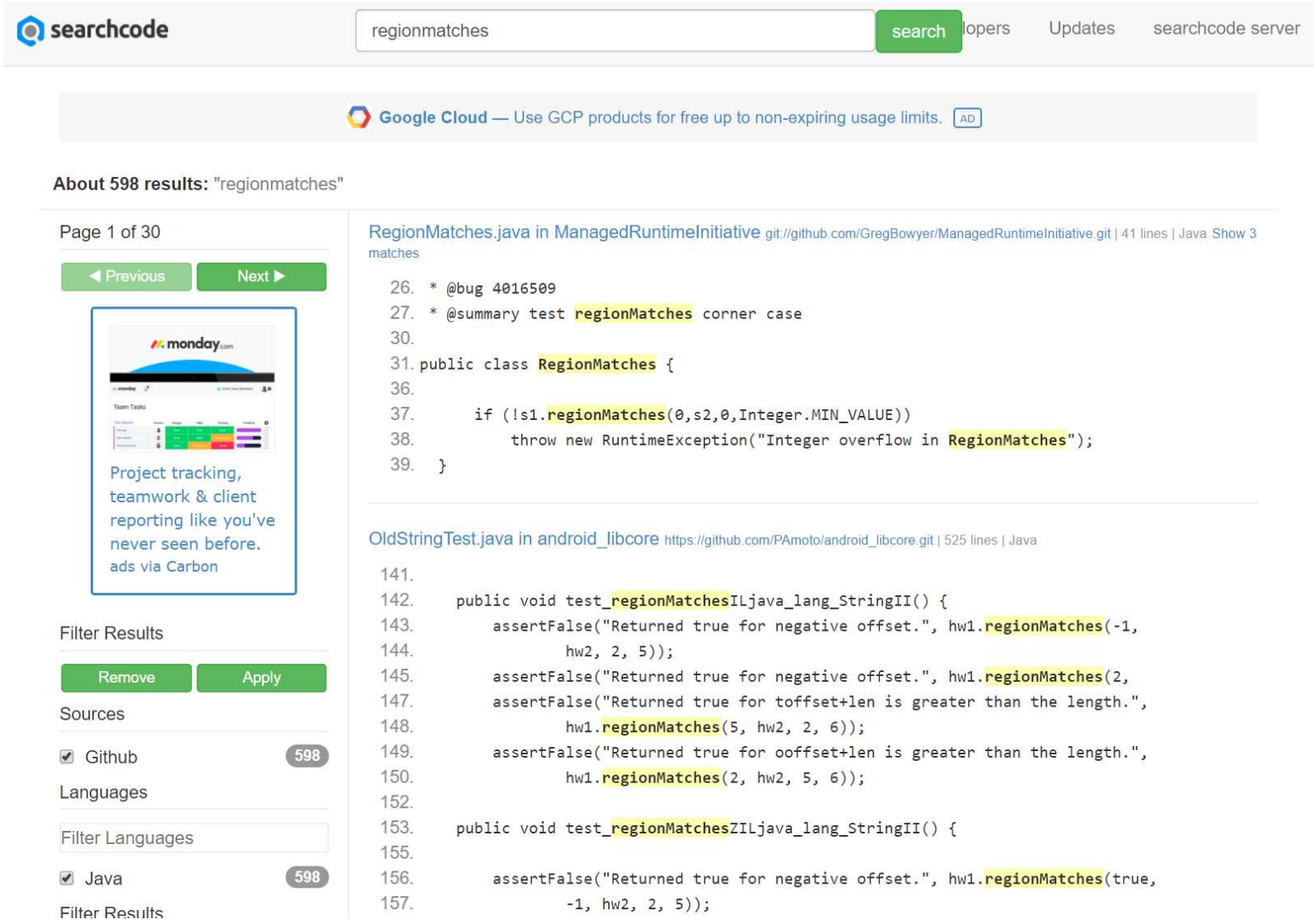
Example code search: grepcode.com



GitHub code search: github.com/search



Searchcode: searchcode.com



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(2) Module-level reuse: class/interface

Inheritance

Use

Composition/aggregation

Delegation/association

Reusing classes

- **A class is an atomic unit of code reuse**
  - Source code not necessary, class file or `jar/zip`
  - Just need to include in the `classpath`
  - Can use `javap` tool to get a class's public method headers

- **Documentation very important (Java API)** 文本说明很重要
- **Encapsulation** 封装 **helps reuse** 封装影响重用
- **Less code to manage**
- **Versioning, backwards-compatibility still problem**
- **Need to package related classes together** **-- Static Linking**

Approaches of reusing a class: inheritance继承

- **Java provides a way of code reuse named** `Inheritance`
  - Classes extend the properties/behavior of existing classes
  - In addition, they might `override` existing behavior
- 继承是一种复用模式，子类可以复用父类的方法，也可以重写。
- **No need to put dummy methods 虚拟方法 that just forward or delegate work**
- **Captures the real world better 更易表达真实的世界**
- **Usually need to design inheritance hierarchy before implementation 通常需要在实现之前设计继承层次结构**
- **Cannot cancel out properties or methods, so must be careful not to overdo it 无法取消属性或方法，因此必须小心不要过度使用**

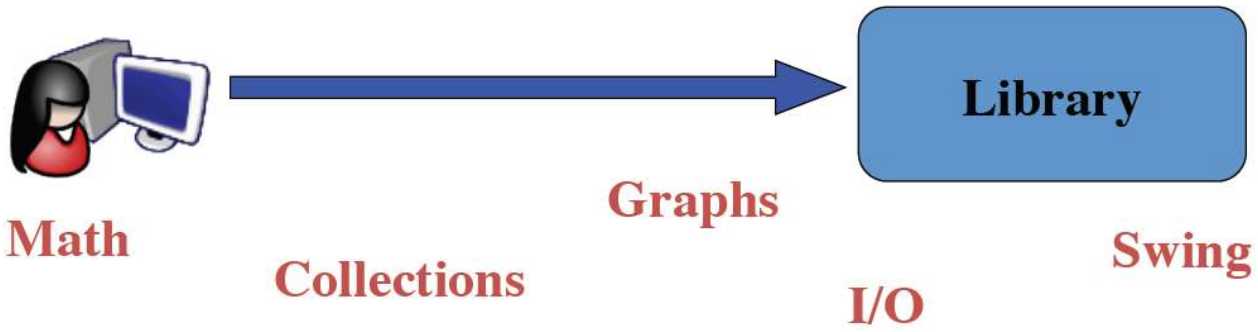
Approaches of reusing a class: delegation委托

- `Delegation` **is simply when one object relies on another object for some subset of its functionality** (one entity passing something to another entity) 委派只是一个对象依赖另一个对象来实现其功能的某个子集。
  - e.g. the `Sorter` is delegating functionality to some `Comparator`
- **Judicious delegation enables code reuse**
  - `Sorter` can be reused with arbitrary arbitrary 任意的 sort orders
  - `Comparators` can be reused with arbitrary client code that needs to compare integers
- **Explicit delegation**（明确授权）**:** passing the sending object to the receiving object
- **Implicit delegation**（隐性委托）**:** by the member lookup rules of the language
- **Delegation** can be described as a low level mechanism for sharing code and data between entities.

(3) Library-level reuse: API/Package

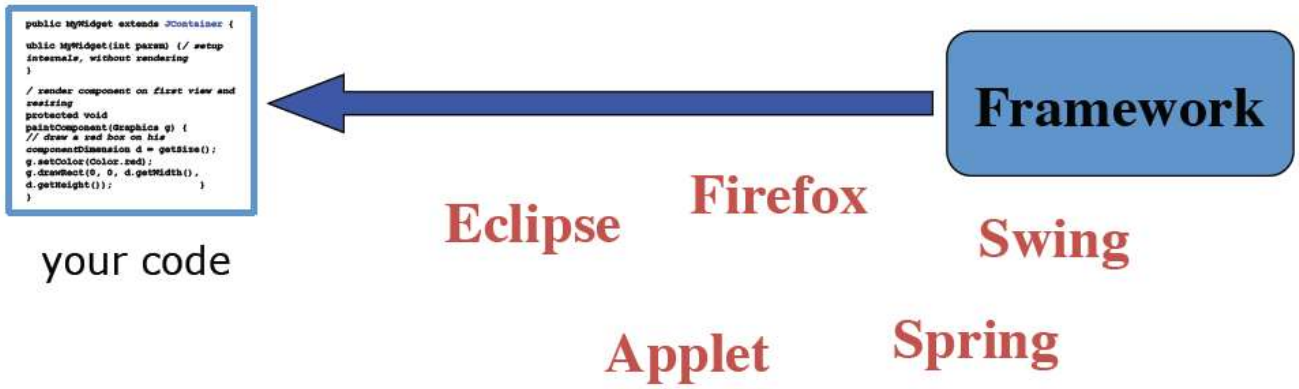
Libraries

- **Library: A set of classes and methods (APIs) that provide reusable functionality**

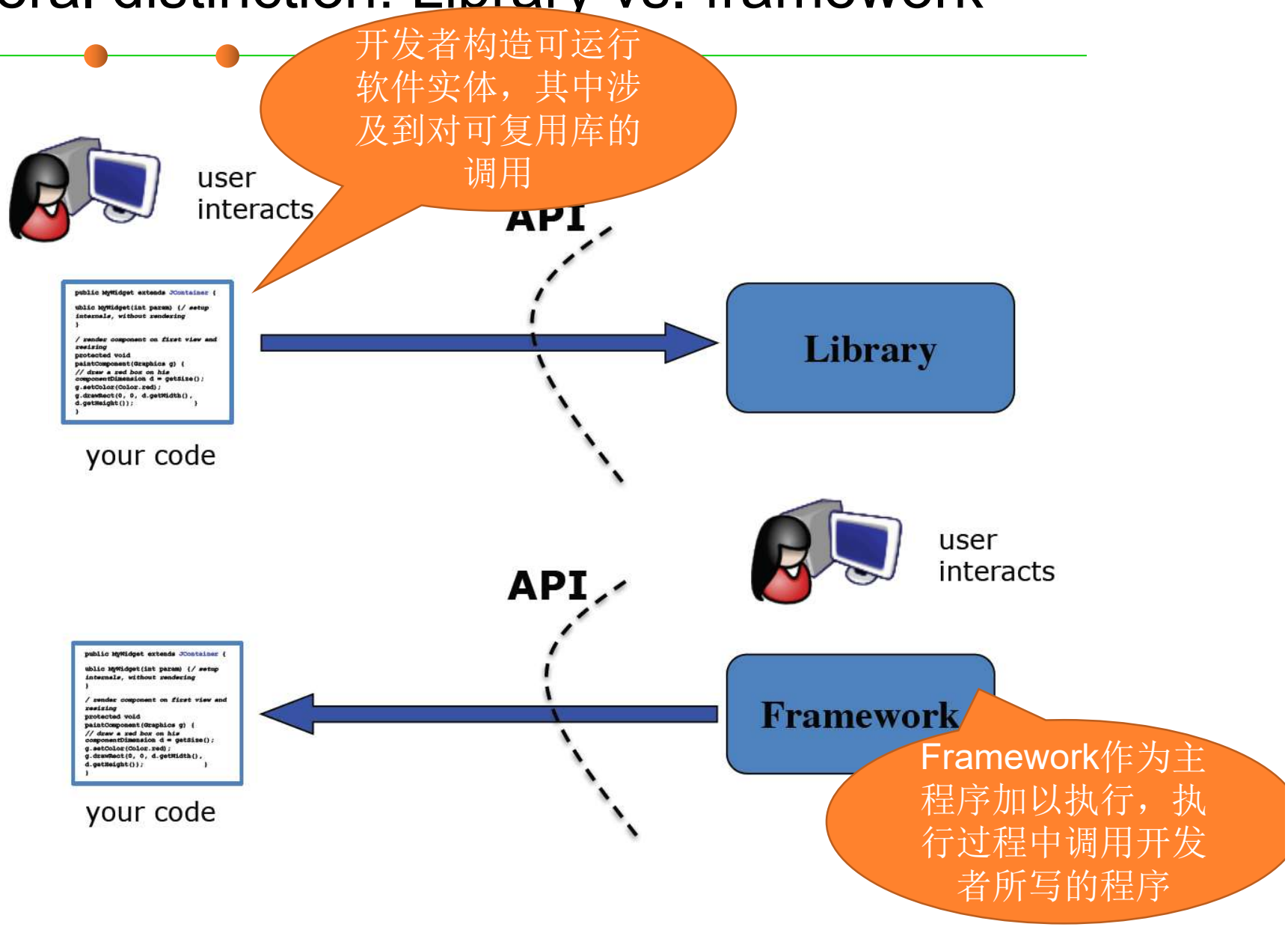


Framework

- **Framework: Reusable skeleton code that can be customized into an application**
- **Framework calls back into client code**
  - The Hollywood principle: “Don’t call us. We’ll call you.”



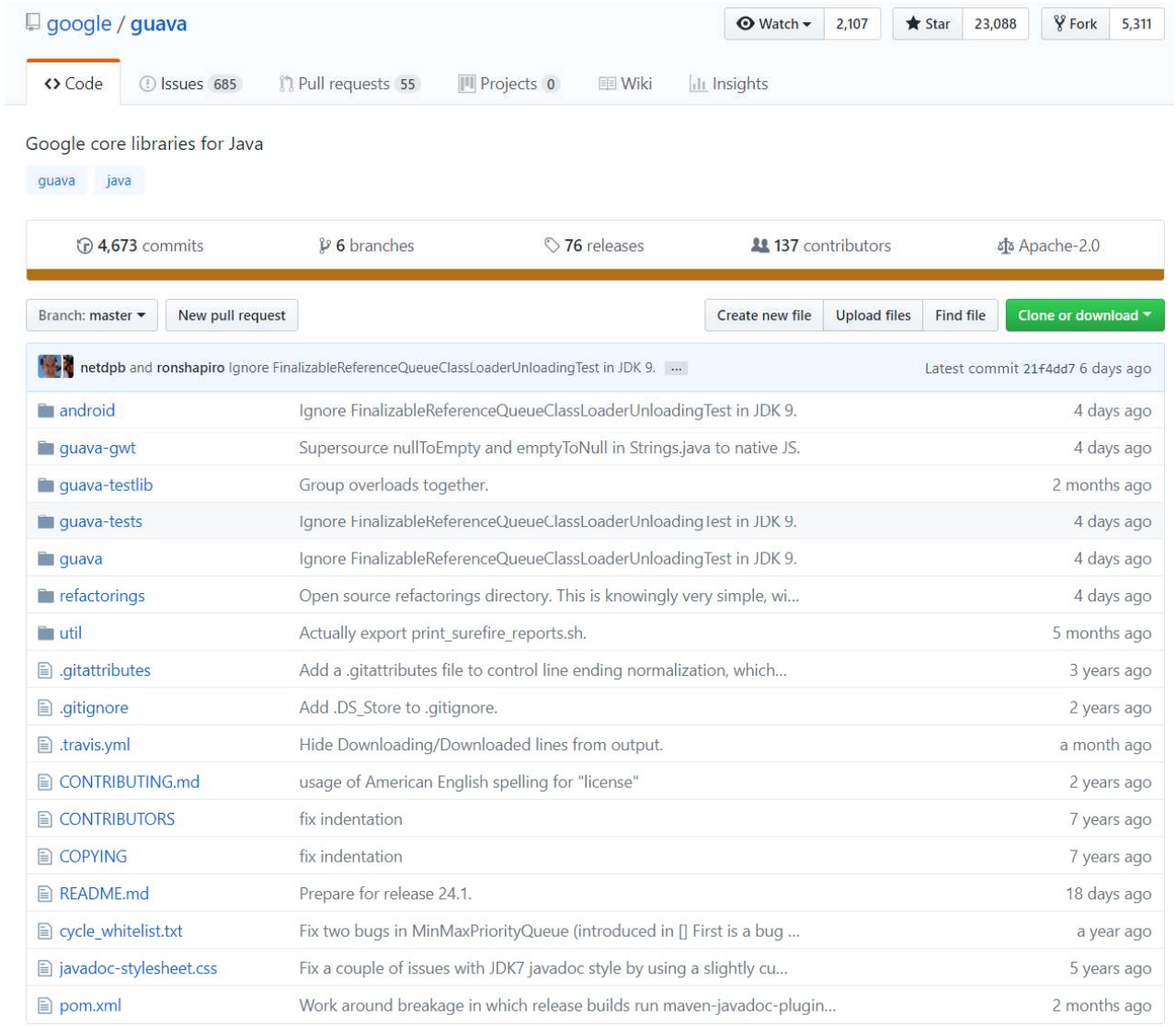
General distinction: Library vs. framework



Characteristics of a good API

- **Easy to learn**
- **Easy to use, even without documentation**
- **Hard to misuse**
- **Easy to read and maintain code that uses it**
- **Sufficiently powerful to satisfy requirements**
- **Easy to evolve**
- **Appropriate to audience**

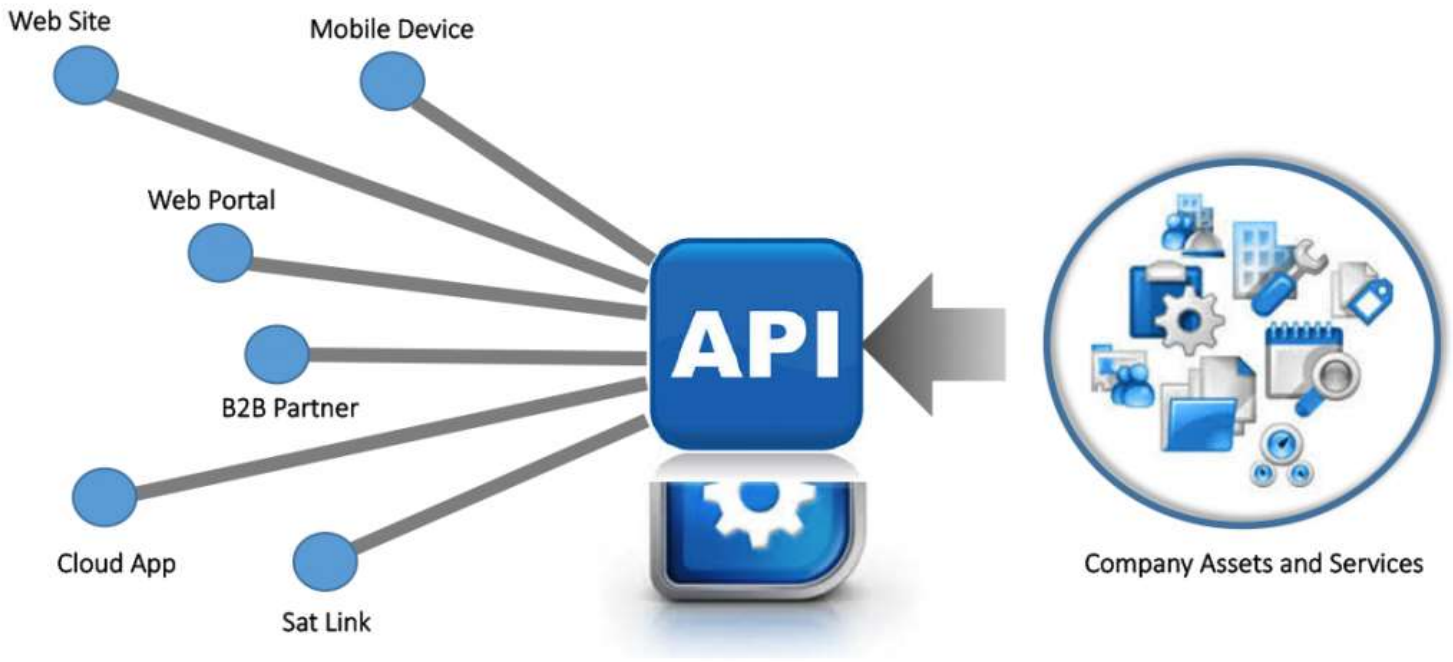
Guava: Google core libraries for Java



Apache Commons

- **Apache Commons is an Apache project focused on all aspects of reusable Java components.**
  - https://commons.apache.org
  - https://github.com/apache/commons-*

API on Web/Internet: Web Services/Restful APIs

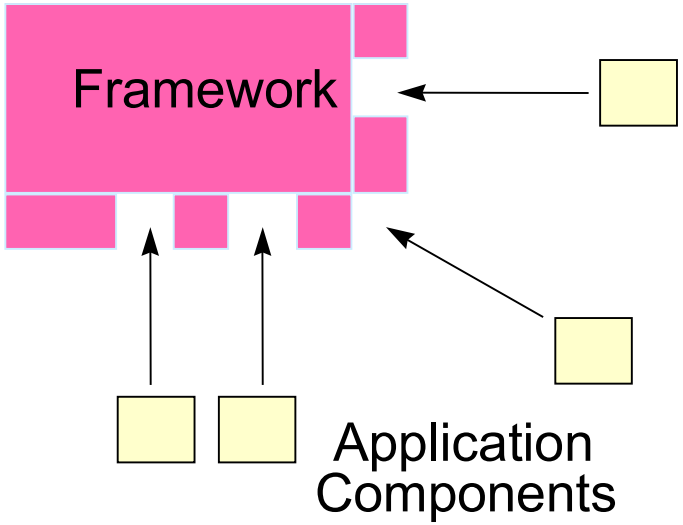


哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(4) System-level reuse: Framework

Application Frameworks

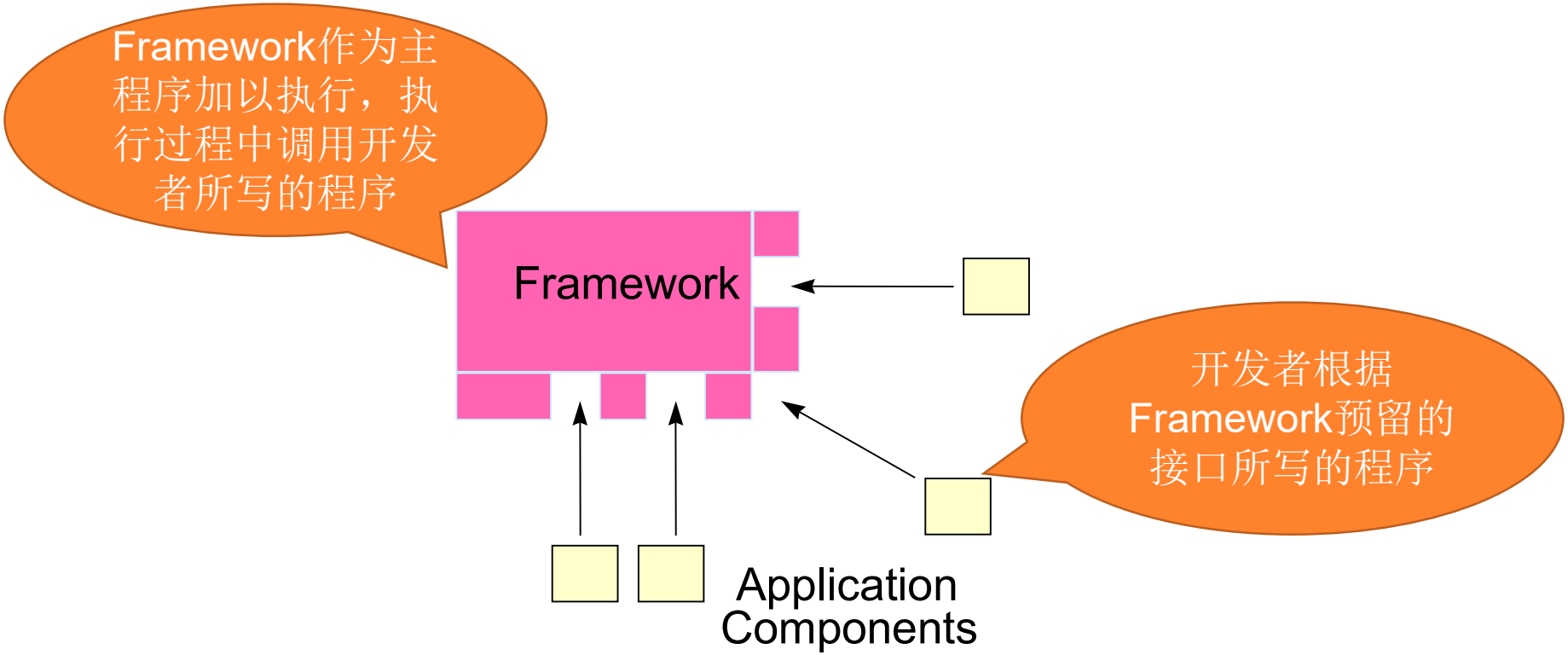
- **Frameworks are sub-system design containing a collection of abstract and concrete classes along with interfaces between each class** 框架：一组具体类、抽象类、及其之间的连接关系
  - 只有“骨架”，没有“血肉”
- **A framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software.**
  开发者根据**framework**的规约，填充自己的代码进去，形成完整系统



Application Frameworks

骨骼清奇，天赋异禀

- **Reusability leverages of the application domain knowledge and prior effort of experienced developers** 领域知识的复用
  - Data processing, GUI, etc
  - 将framework看作是更大规模的API复用，除了提供可复用的API，还将这些模块之间的关系都确定下来，形成了整体应用的领域复用



Framework Design

- **Frameworks differ from applications**
  - The level of abstraction is different as frameworks provide a solution for a family of related problems, rather than a single one.*抽象的层次是不同的，因为框架为一系列相关问题提供了解决方案，而不是单个问题。*
  - To accommodate the family of problems, the framework is incomplete, incorporating hot spots and hooks to allow customization*为了适应这一系列的问题，这个框架是不完整的，包含了热点和钩子以允许定制*

- **Frameworks can be classified by the techniques used to extend them.**
  - Whitebox frameworks 黑盒框架
  - Blackbox frameworks 白盒框架

White-box and Black-Box Frameworks

- **Whitebox frameworks** 白盒框架，通过代码层面的继承进行框架扩展
  - Extensibility achieved through inheritance and dynamic binding.
  - Existing functionality is extended by subclassing framework base classes and overriding predefined hook methods
  - Often design patterns such as the template method pattern are used to override the hook methods.
- **Blackbox frameworks** 黑盒框架，通过实现特定接口**/delegation**进行框架扩展
  - Extensibility achieved by defining interfaces for components that can be plugged into the framework.
  - Existing functionality is reused by defining components that conform to a particular interface
  - These components are integrated with the framework via delegation.

5 Designing reusable classes

Designing reusable classes in OOP

- **Encapsulation and information hiding**
- **Inheritance and overriding**
- **Polymorphism, subtyping and overloading**
- **Generic programming** 泛型编程

Already introduced in section 3.4 OOP

- **Behavioral subtyping and Liskov Substitution Principle (LSP)**
- **Delegation and Composition**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(1) Behavioral subtyping and Liskov Substitution Principle (LSP)

行为子类型和里氏替换原则

Behavioral subtyping 行为子类

- **Subtype polymorphism: Different kinds of objects can be treated uniformly by client code.** 子类型多态：客户端可用统一的方式处理不同类型的对象
  - If the type `Cat` is a subtype of `Animal`, then an expression of type `Cat` can be used wherever an expression of type `Animal` is used.

```
Animal a = new Animal();
Animal c1 = new Cat();
Cat c2 = new Cat();
```

在可以使用a的场景，都可以用c1和c2代替而不会有任何问题

```
a = c1;
a = c2;
```

- **Let `q(x)` be a property provable about objects `x` of type `T`, then `q(y)` should be provable for objects `y` of type `S` where `S` is a subtype of `T`.**

——Barbara Liskov

Barbara Liskov

Barbara Liskov (1939-)

MIT
http://www.pmg.csail.mit.edu/~liskov

美国第一位计算机科学方向的女博士
2008年图灵奖获得者

提出了第一个支持数据抽象的面向对象编程语言**CLU**，对现代主流语言如**C++/Java/Python/Ruby/C#**都有深远的影响。她所提炼出来的数据抽象思想，成为软件工程的重要精髓之一。

她提出的“**Liskov**替换原则”，是面向对象最重要的几大原则之一。

Liskov替换原则

- 第一种定义，也是最正宗的定义：**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 behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.**
- （如果对每一个类型为**S**的对象**o1**，都有类型为**T**的对象**o2**，使得以**T**定义的所有程序**P**在所有的对象**o1**都代换成**o2**时，程序**P**的行为没有发生变化，那么类型**S**是类型**T**的子类型。）
- 第二种定义：**Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.**
- （所有引用基类的地方必须能透明地使用其子类的对象。）

LSP（里氏替换原则）

- **1、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；**
- **2、子类中可以增加自己特有的方法。**
- **具体表现在：**

- 子类必须实现父类所有非私有的属性和方法，或子类的所有非私有属性和方法必须在父类中声明。即，子类可以有自己的“个性”，这也就是说，里氏代换原则可以正着用，不能反着用（在项目中，采用里氏替换原则时，尽量避免子类的“个性”，一旦子类有“个性”，这个子类和父类之间的关系就很难调和了）。根据里氏代换原则，为了保证系统的扩展性，在程序中通常使用父类来进行定义，如果一个方法只存在子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。
- 尽量把父类设计为抽象类或者接口。让子类继承父类或实现父接口，并实现在父类中声明的方法，运行时，子类实例替换父类实例，我们可以很方便地扩展系统的功能，同时无须修改原有子类的代码，增加新的功能可以通过增加一个新的子类来实现。

Behavioral subtyping

- **Compiler-enforced rules in Java (static type checking)**
  - Subtypes can add, but not remove methods 子类型可以增加方法，但不可删
  - Concrete class must implement all undefined methods 子类型需要实现抽象类型中的所有未实现方法
  - Overriding method must return same type or subtype 子类型中重写的方法必须有相同或子类型的返回值或者符合co-variance的参数
  - Overriding method must accept the same parameter types 子类型中重写的方法必须使用同样类型的参数或者符合contra-variance的参数
  - Overriding method may not throw additional exceptions 子类型中重写的方法不能抛出额外的异常

子类型规约更强

- **Also applies to specified behavior (methods):** 规约不变或更强
  - Same or stronger invariants 更强的不变量
  - Same or weaker preconditions 更弱的前置条件
  - Same or stronger postconditions 更强的后置条件

Liskov Substitution Principle (LSP)

Example 1 for Behavioral subtyping (LSP)

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method has the same pre- and post-conditions**

```
abstract class Vehicle {

  int speed, limit;
  //@ invariant speed < limit;

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  void brake();
}
```

不变性增强

强度不变

```
class Car extends Vehicle {
  int fuel;
  boolean engineOn;
  //@ invariant speed < limit;
  //@ invariant fuel >= 0;

  //@ requires fuel > 0 && !engineOn;
  //@ ensures engineOn;
  void start() { … }
  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  void brake() { … }
}
```

Example 2 for Behavioral subtyping (LSP)

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method `start` has weaker precondition**
- **Overridden method `brake` has stronger postcondition**

```
class Car extends Vehicle {
  int fuel;
  boolean engineOn;
  //@ invariant speed < limit;
  //@ invariant fuel >= 0;

  //@ requires fuel > 0 && !engineOn;
  //@ ensures engineOn;
  void start() { … }
  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  void brake() { … }
}
```

前置减弱

后置增强

```
class Hybrid extends Car {
  int charge;
  //@ invariant charge >= 0;

  //@ requires (charge > 0
  //@       || fuel > 0) && !engineOn;
  //@ ensures engineOn;
  void start() { … }
  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  //@ ensures charge > \old(charge)
  void brake() { … }
}
```

Behavioral subtyping (LSP)

- **How about these two classes? Is LSP satisfied?**

```
class Rectangle {
  int h, w;
  Rectangle(int h, int w) {
    this.h=h; this.w=w;
  }
  //methods
}
```

```
class Square extends Rectangle {
  Square(int w) {
    super(w, w);
  }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
  //@ invariant h>0 && w>0;
  int h, w;
  Rectangle(int h, int w) {
    this.h=h; this.w=w;
  }
  //methods
}
```

不变性增强

```
class Square extends Rectangle {
  //@ invariant h>0 && w>0;
  //@ invariant h==w;
  Square(int w) {
    super(w, w);
  }
}
```

Behavioral subtyping (LSP)

- **How about these two classes? Is LSP satisfied?**

```
class Rectangle {
  //@ invariant h>0 && w>0;
  int h, w;
  Rectangle(int h, int w) {
    this.h=h; this.w=w;
  }

  //@ requires factor > 0;
  void scale(int factor) {
    w=w*factor;
    h=h*factor;
  }
}
```

```
class Square extends Rectangle {
  //@ invariant h>0 && w>0;
  //@ invariant h==w;
  Square(int w) {
    super(w, w);
  }
}
```

Is this Square a behavioral subtype of Rectangle?

Behavioral subtyping (LSP)

Is this `Square` a behavioral subtype of `Rectangle`?

```
class Rectangle {
  //@ invariant h>0 && w>0;
  int h, w;
  Rectangle(int h, int w) {
    this.h=h; this.w=w;
  }
  //@ requires factor > 0;
  void scale(int factor) {
    w=w*factor;
    h=h*factor;
  }

  //@ requires neww > 0;
  void setWidth(int neww) {
    w=neww;
  }
}
```

```
class Square extends Rectangle {
  //@ invariant h>0 && w>0;
  //@ invariant h==w;
  Square(int w) {
    super(w, w);
  }
}
```

```
class GraphicProgram {
  void scaleW(Rectangle r, int factor) {
    r.setWidth(r.getWidth() * factor);
  }
}
```

Invalidates stronger invariant `(w==h)` in subclass

Behavioral subtyping (LSP)

```
class Rectangle {
  //@ invariant h>0 && w>0;
  int h, w;
  Rectangle(int h, int w) {
    this.h=h; this.w=w;
  }
  //@ requires factor > 0;
  void scale(int factor) {
    w=w*factor;
    h=h*factor;
  }
  //@ requires neww > 0;
  //@ ensures w=neww
  //        && h not changed
  void setWidth(int neww) {
    w=neww;
  }
}
```

```
class Square extends Rectangle {
  //@ invariant h>0 && w>0;
  //@ invariant h==w;
  Square(int w) {
    super(w, w);
  }

  //@ requires neww > 0;
  //@ ensures w=neww && h=neww

  @Override
  void setWidth(int neww) {
    w=neww;
    h=neww;
  }
}
```

Liskov Substitution Principle (LSP)

- **LSP is a particular definition of a subtyping relation, called (strong) behavioral subtyping 强行为子类型化**
- **In programming languages, LSP is relied on the following restrictions:**
  - Preconditions cannot be strengthened in a subtype. 前置条件不能强化
  - Postconditions cannot be weakened in a subtype. 后置条件不能弱化
  - Invariants of the supertype must be preserved in a subtype. 不变量要保持
  - Contravariance of method arguments in a subtype 子类型方法参数：逆变
  - Covariance of return types in a subtype. 子类型方法的返回值：协变
  - No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype. 异常类型：协变 (This is to be discussed in Section 7-2)

Covariance (协变)

父类型→子类型：越来越具体specific
返回值类型：不变或变得更具体
异常的类型：也是如此。

- **See this example:**

```
class T {
  Object a() { … }
}

class S extends T {
  @Override
  String a() { … }
}
```

- **More specific classes may have more specific return types**
- **This is called covariance of return types in the subtype.**

```
class T {
  void b( ) throws Throwable {…}
}

class S extends T {
  @Override
  void b( ) throws IOException {…}
}

class U extends S {
  @Override
  void b( ) {…}
}
```

- **Every exception declared for the subtype's method should be a subtype of some exception declared for the supertype's method.**

Contravariance (反协变、逆变)

- **What do you think of this code?**

```
class T {
  void c( String s ) { … }
}

class S extends T {
  @Override
  void c( Object s ) { … }
}
```

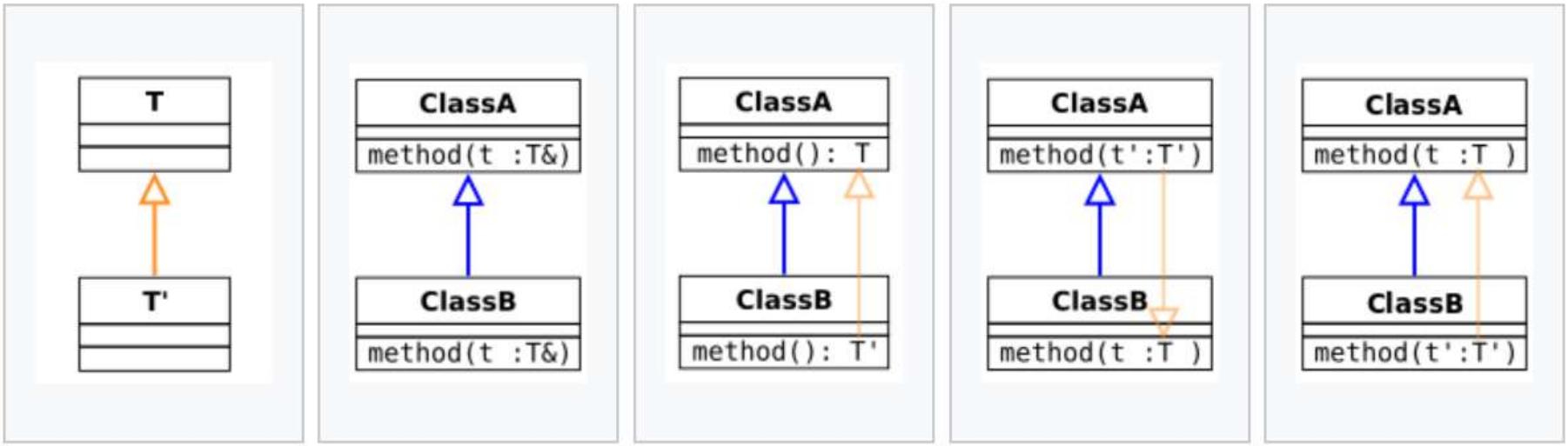
父类型→子类型：越来越具体specific
参数类型：要相反的变化，要不变或越来越抽象

- **Logically, it is called contravariance of method arguments in the subtype.**
- **This is actually not allowed in Java, as it would complicate the overloading rules.** 目前Java中遇到这种情况，当作overload看待 ☹

The method `c(Object)` of type `S` must override or implement a supertype method

Summary on subtyping and LSP

在Java中都当成overload



Subtyping of the argument/return type of the method.

Invariance. The signature of the overriding method is unchanged.

Covariant return type. The subtyping relation is in the same direction as the relation between ClassA and ClassB.

Contravariant argument type. The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.

Covariant argument type. Not type safe.

Co-variance and Contra-variance

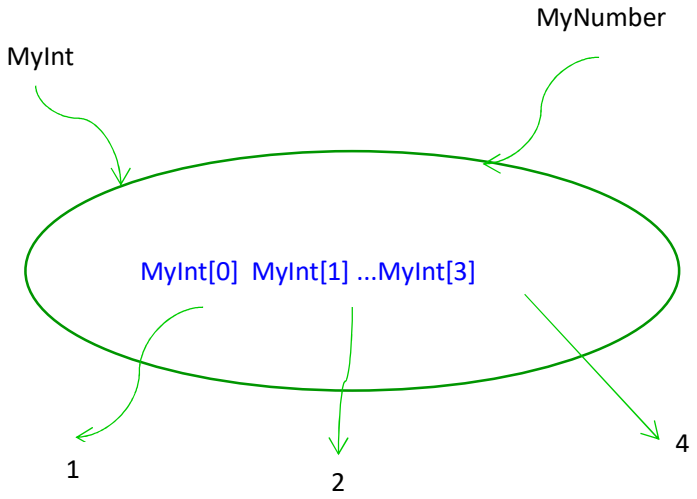
- **Arrays are covariant** 协变的**:** given the subtyping rules of Java, an array of type `T[]` may contain elements of type `T` or any subtype of `T`.

```
Number[] numbers = new Number[2];
numbers[0] = new Integer(10);
numbers[1] = new Double(3.14);

Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;

myNumber[0] = 3.14; //run-time error!
```

- **At run-time Java knows that this array was actually instantiated as an array of integers which simply happens to be accessed through a reference of type** `Number[]`**.**

myNumber是引用类型，而实际是Interger类型。

  - 区分：Type of an object **vs.** Type of a reference


MyNumber
MyInt
MyInt[0] MyInt[1] ...MyInt[3]
1
2
4

Consider LSP for generics 泛型中的LSP

```
Object obj = new Object();
Integer integer = new Integer(10);
obj = integer;
```

*满足 **LSP***

```
void someMethod(Number n)
{ ... }

someMethod(new Integer(10));
someMethod(new Double(10.1));
```

*满足 **LSP***

- **So, how about in generics?**
- **Generics are type invariant**
  - `ArrayList<String>` is a subtype of `List<String>`
  - `List<String>` is not a subtype of `List<Object>`
- **The type information for type parameters is discarded by the compiler after the compilation of code is done; therefore this type information is not available at run time.** 编译代码完成后，编译器丢弃类型参数的类型信息；因此，此类型信息在运行时不可用。
- **This process is called type erasure** 类型擦除
- **Generics are not covariant.** 泛型不是协变的

What is type erasure? 类型擦除

- **Type erasure:** Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.如果泛型类型中的类型参数是无限制的，将被对象类型参数所替换。因此，生成的字节码只包含普通类、接口和方法。

编译前

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

编译后

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data,
                Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() {
        return data; }
    // ...
}
```

An example

```
List<Integer> myInts = new ArrayList<Integer>();
myInts.add(1);
myInts.add(2);
List<Number> myNums = myInts;   //compiler error
myNums.add(3.14);
```

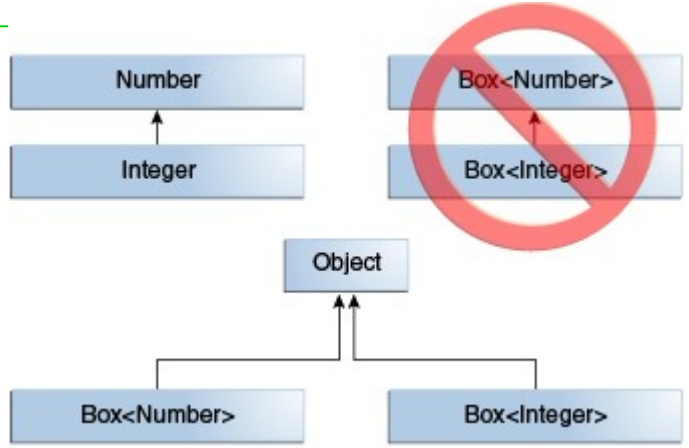
```
static long sum(List<Number> numbers) {
  long summation = 0;
  for(Number number : numbers) {
    summation += number.longValue();
  }
  return summation;
}
```

We cannot consider a list of integers to be subtype of a list of numbers.

That would be considered unsafe for the type system and compiler rejects it immediately.

```
List<Integer> myInts = asList(1,2,3,4,5);
List<Long> myLongs = asList(1L, 2L, 3L, 4L, 5L);
List<Double> myDoubles = asList(1.0, 2.0, 3.0, 4.0, 5.0);
sum(myInts);   //compiler error
sum(myLongs);  //compiler error
sum(myDoubles);//compiler error
```

Consider LSP for generics 泛型中的LSP



- `Box<Integer>` **is not a subtype of** `Box<Number>` **even though** `Integer` **is a subtype of** `Number`**.**
- **Given two concrete types** `A` **and** `B` **(for example,** `Number` **and** `Integer`**),** `MyClass<A>` **has no relationship to** `MyClass<B>`**, regardless of whether or not** `A` **and** `B` **are related. The common parent of** `MyClass<A>` **and** `MyClass<B>` **is** `Object`**.**
- **A与B有关系，Box（A）与Box（B）没有关系。**
- **For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see Wildcards.**
  - https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

Wildcards in Generics 泛型编程中的通配符

- **The unbounded wildcard type is specified using the wildcard character (`?`), for example, `List<?>`.**
  - This is called a list of unknown type.
- **There are two scenarios where an unbounded wildcard is a useful approach:**
  - If you are writing a method that can be implemented using functionality provided in the `Object` class.
  - When the code is using methods in the generic class that don't depend on the type parameter. For example, `List.size` or `List.clear`.
  - In fact, `Class<?>` is so often used because most of the methods in `Class<T>` do not depend on `T`.

Wildcards in Generics 泛型编程中的通配符

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

The goal of `printList` is to print a list of any type, but it fails to achieve that goal — it prints only a list of `Object` instances; it cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on, because they are not subtypes of `List<Object>`.

To write a generic `printList` method, use `List<?>`

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

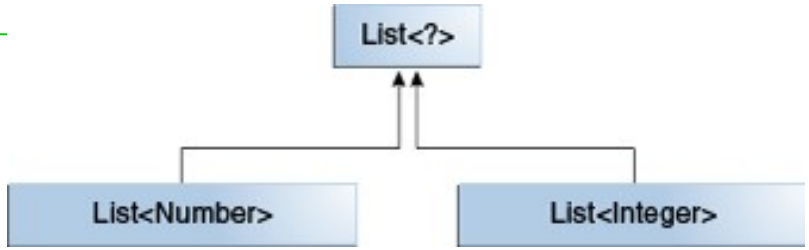
Wildcards in Generics

- **Lower Bounded Wildcards:** `<? super A>` 下限通配符
  - `List<Integer>` `List<? super Integer>`
  - The former matches a list of type `Integer` only, whereas the latter matches a list of any type that is a supertype of Integer such as `Integer`, `Number`, and `Object`.
- **Upper Bounded Wildcards:** `<? extends A>` 上线通配符
  - `List<? extends Number>`

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
```

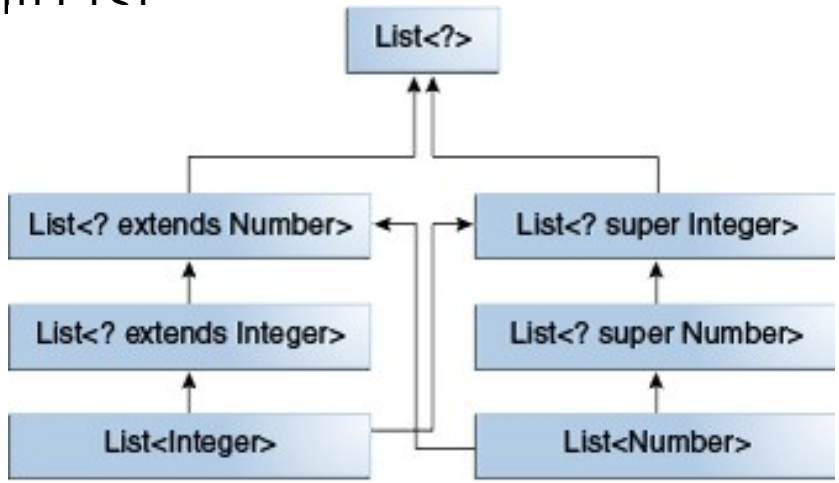
Consider LSP for generics with wildcards

- `List<Number>` is a subtype of `List<?>`



- `List<Number>` is a subtype of `List<? extends Object>`
- `List<Object>` is a subtype of `List<? super String>`

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;
```



Consider LSP for generics with wildcards

In the class `java.util.Collections`:

```
public static <T> void copy(
    List<? super T> dest,
    List<? extends T> src)
```

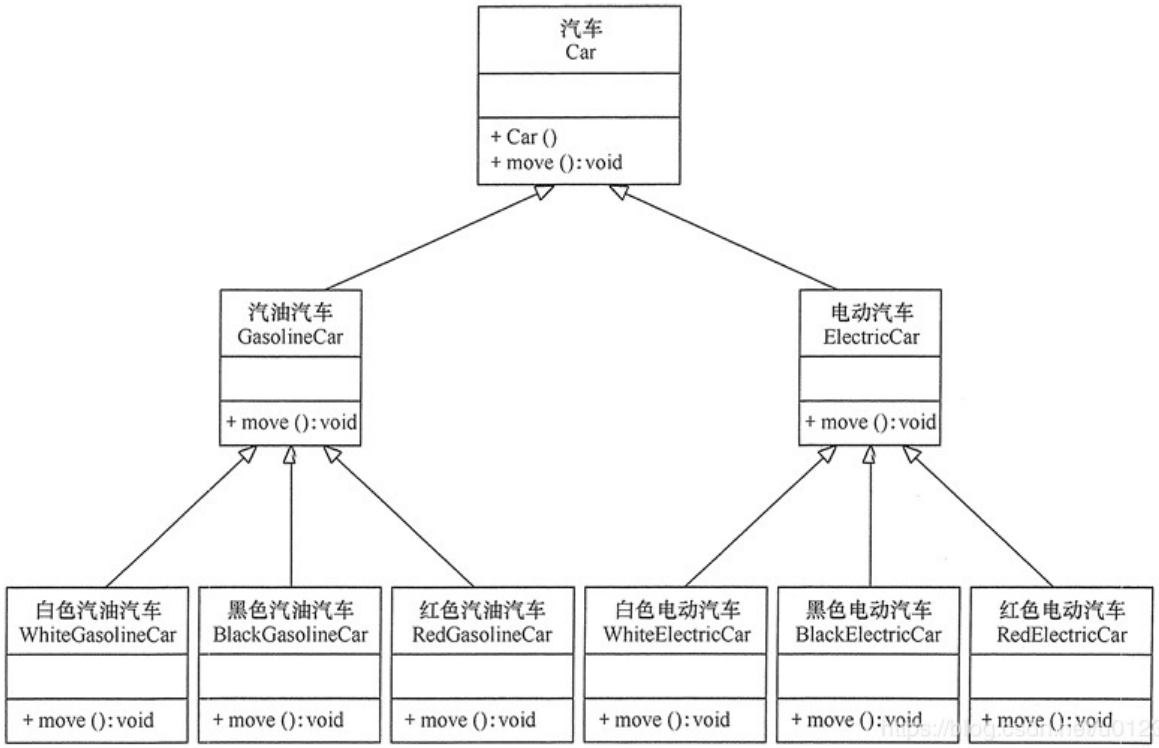
```
List<Number> source = new LinkedList<>();
source.add(Float.valueOf(3));
source.add(Integer.valueOf(2));
source.add(Double.valueOf(1.1));

List<Object> dest = new LinkedList<>();

Collections.copy(dest,source);
```

继承复用的优缺点

- 通常类的复用分为继承复用和合成复用两种，继承复用虽然有简单和易实现的优点。
- 继承复用破坏了类的封装性。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。
- 子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。
- 它限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化

继承复用有时会很复杂

- 汽车按“动力源”划分可分为汽油汽车、电动汽车等；按“颜色”划分可分为白色汽车、黑色汽车和红色汽车等。
- 如果同时考虑这两种分类，其组合就很多……



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(2) Delegation and Composition

授权和组成

A `Sorting` example

- **Version A:**

```
static void sort(int[] list, boolean ascending) {
   …
   boolean mustSwap;
   if (ascending) {
      mustSwap = list[i] < list[j];
   } else {
      mustSwap = list[i] > list[j];
   }
   …
}
```

- **Version B:**

```
interface Comparator {
  boolean compare(int i, int j);
}
final Comparator ASCENDING =  (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(list[i], list[j]);
  …
}
```

Interface Comparator<T>

- `int compare(T o1, T o2)`: **Compares its two arguments for order.**
  - A comparison function, which imposes a total ordering on some collection of objects.
  - Comparators can be passed to a sort method (such as `Collections.sort` or `Arrays.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering.

- 如果你的ADT需要比较大小，或者要放入`Collections`或`Arrays`进行排序，可实现`Comparator`接口并override `compare()`函数。

Interface Comparator<T>

```
public class Edge {
   Vertex s, t;
   double weight;
   ...
}
```

```
public class EdgeComparator
      implements Comparator<Edge>{

   @Override
   public int compare(Edge o1, Edge o2) {
      if(o1.getWeight() > o2.getWeight())
        return 1;
      else if (.. == ..) return 0;
      else return -1;
   }
}
```

```
public void sort(List<Edge> edges) {
  Comparator comparator = new EdgeComparator();
  Collections.sort(edges, comparator);
}
```

Interface Comparable<T>

- **This interface imposes a total ordering on the objects of each class that implements it.**
- **This ordering is referred to as the class's natural ordering, and the class's `compareTo` method is referred to as its natural comparison method.**
- **另一种方法：让你的ADT实现`Comparable`接口，然后override `compareTo()` 方法**
- **与使用`Comparator`的区别：不需要构建新的`Comparator`类，比较代码放在ADT内部。**
- **This is not *delegation* any longer.**

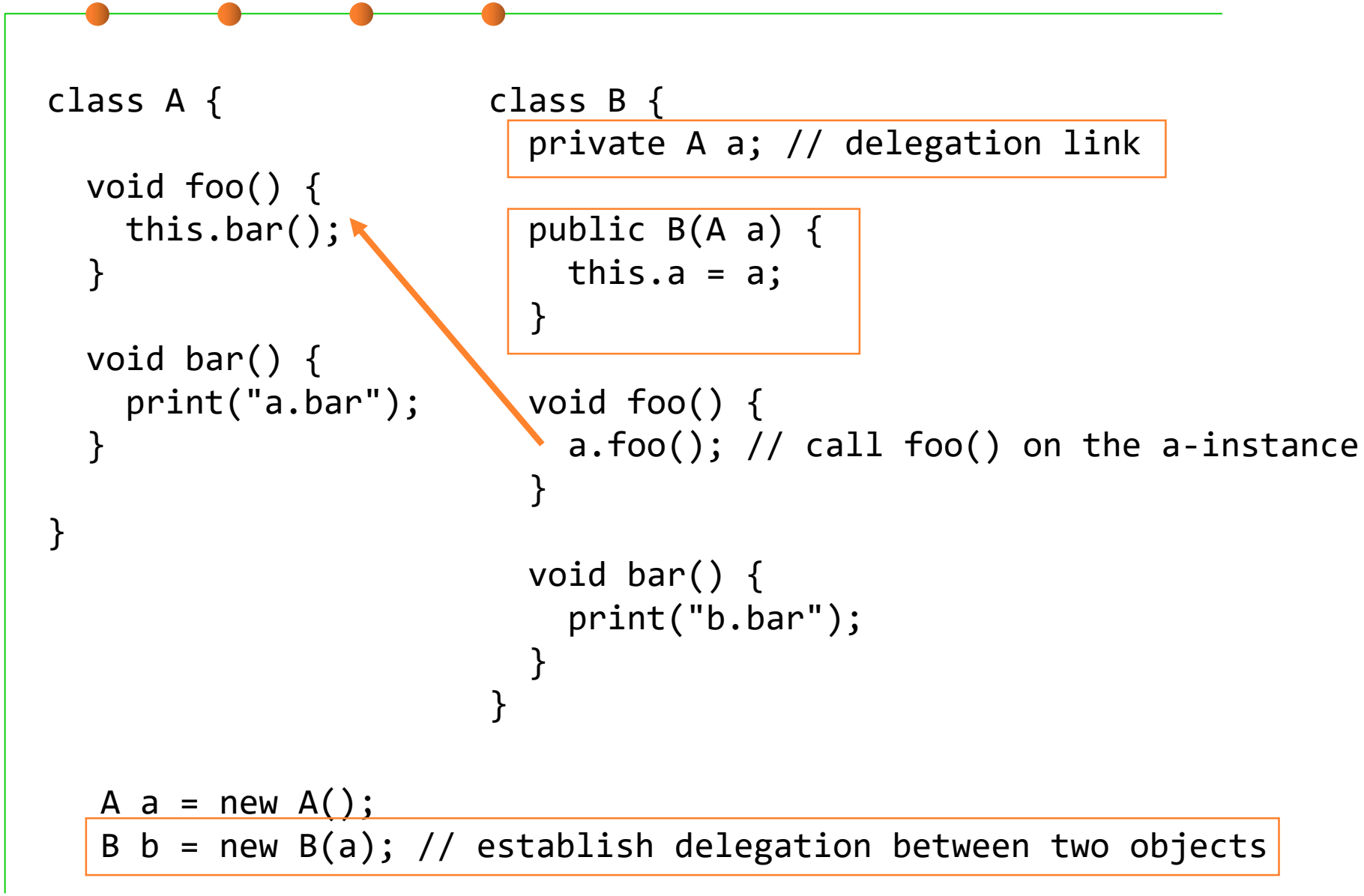
Interface Comparable<T>

```
public class Edge implements Comparable<Edge> {
    Vertex s, t;
    double weight;
    ...

    public int compareTo(Edge o) {
       if(this.getWeight() > o.getWeight())
         return 1;
       else if (.. == ..) return 0;
       else return -1;
    }

}
```

Delegation

- **Delegation is simply when one object relies on another object for some subset of its functionality** (one entity passing something to another entity) 委派/委托：一个对象请求另一个对象的功能
  - e.g. the `Sorter` is delegating functionality to some `Comparator`
- **Judicious delegation enables code reuse** 委派是复用的一种常见形式
  - `Sorter` can be reused with arbitrary sort orders
  - `Comparators` can be reused with arbitrary client code that needs to compare integers
- **Delegation** can be described as a low level mechanism for sharing code and data between entities.
  - Explicit delegation显性委托: passing the sending object to the receiving object
  - Implicit delegation隐性委托: by the member lookup rules of the language

A simple Delegation example



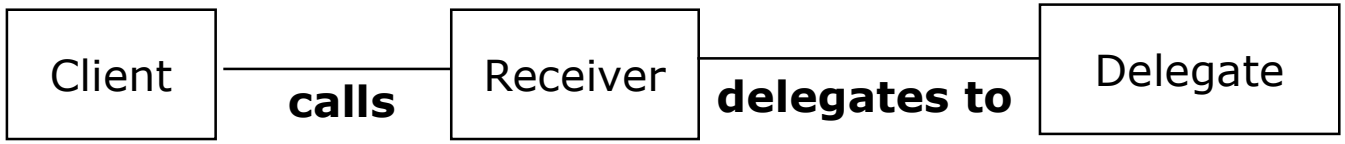
```
class A {

  void foo() {
    this.bar();
  }

  void bar() {
    print("a.bar");
  }

}
```

```
class B {
  private A a; // delegation link

  public B(A a) {
    this.a = a;
  }

  void foo() {
    a.foo(); // call foo() on the a-instance
  }

  void bar() {
    print("b.bar");
  }
}
```

```
A a = new A();
B b = new B(a); // establish delegation between two objects
```

Delegation

- **The delegation pattern** is a software design pattern for implementing delegation, though this term is also used loosely for consultation or forwarding. 委派模式：通过运行时动态绑定，实现对其他类中代码的动态复用
- **Delegation is dependent upon dynamic binding, as it requires that a given method call can invoke different segments of code at runtime.**
- **Process**
  - The `Receiver` object delegates operations to the `Delegate` object
  - The `Receiver` object makes sure, that the `Client` does not misuse the `Delegate` object.



Using delegation to extend functionality

- **Consider** `java.util.List`

  我需要一个能**log**的`List`

```
public interface List<E> {
  public boolean add(E e);
  public E       remove(int index);
  public void    clear();
  …
}
```

- **Suppose we want a list that logs its operations to the console…**
  - The `LoggingList` *is composed of* a `List`, and delegates (the non-logging) functionality to that `List`.

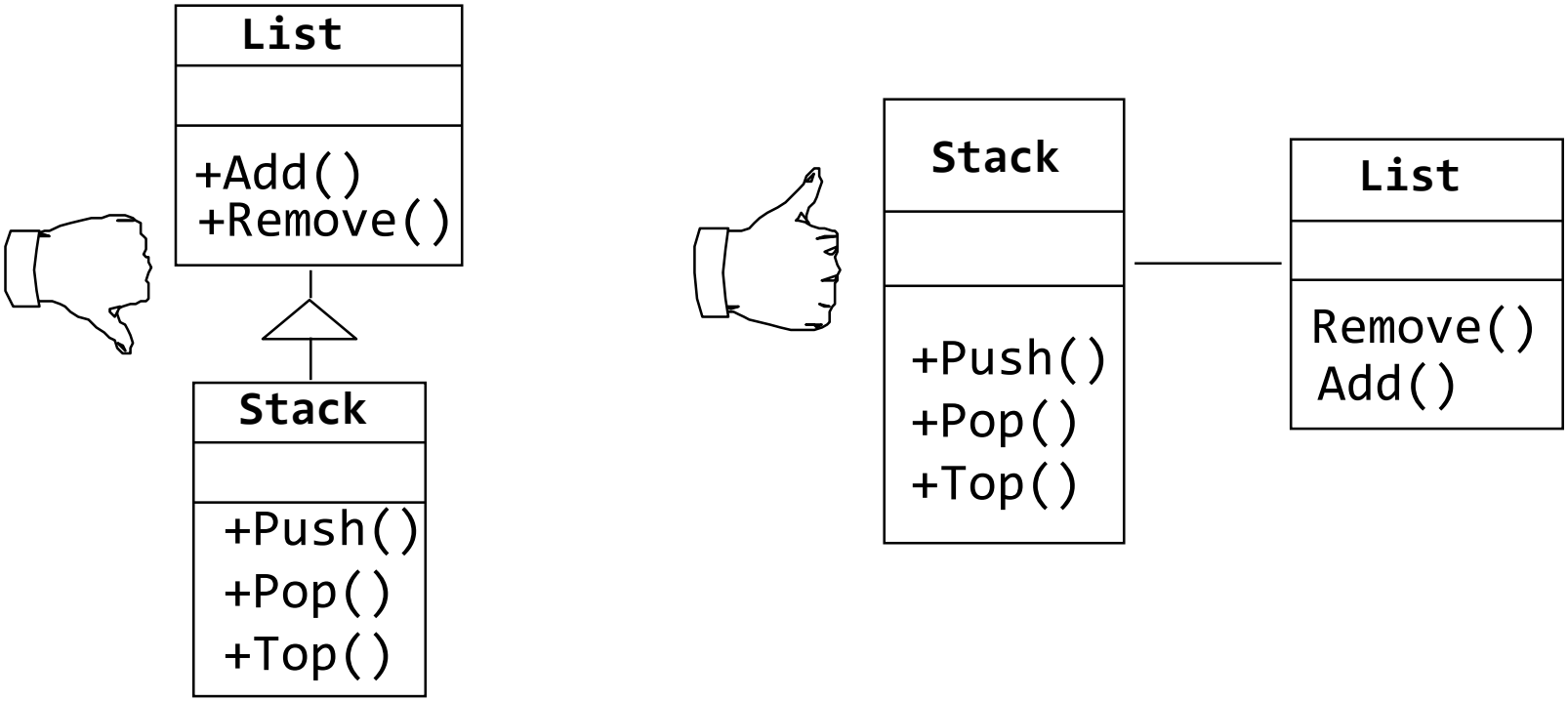
```
public class LoggingList<E> implements List<E> {
  private final List<E> list;
  public LoggingList<E>(List<E> list) { this.list = list; }
  public boolean add(E e) {
      System.out.println("Adding " + e);
      return list.add(e);
  }
  public E remove(int index) {
      System.out.println("Removing at " + index);
      return list.remove(index);
  }
```

这里实现动态绑定

这里进行功能委派

Delegation vs. Inheritance

- **Inheritance:** Extending a `Base` class by a new operation or overwriting an operation.
- **Delegation:** Catching an operation and sending it to another object.
- **Many design patterns use a combination of inheritance and delegation.** *设计经常采用继承和委托结合的方式*



Delegation vs. Inheritance

- **Lab2的P2中，要复用P1中开发出的Graph<L>完成社交网络相关功能**

```
public class FriendshipGraph {
  private Graph<Person> graph = Graph.empty();
  public void addVertex(Person p) {
     graph.add(p);
  }
  ...
  public void getDistance(Person a, Person b) {
  //Graph<L>不具备该功能，需要自己实现
  }
}
```

Delegation vs. Inheritance

- **Lab2的P2中，要复用P1中开发出的Graph<L>完成社交网络相关功能**

```
public class FriendshipGraph<Person>
                                extends ConcreteEdgeGraph<Person> {
  @Override
  public void addVertex(Person p) {
    ...
    super.add(p);
  }
  ...
  public void getDistance(Person a, Person b) {
  //Graph<L>不具备该功能，需要自己实现
  }
}
```

Replace Inheritance with Delegation

- **Problem:** You have a subclass that uses only a portion of the methods of its superclass (or it's not possible to inherit superclass data). 如果子类只需要复用父类中的一小部分方法
- **Solution:** Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance. 可以不需要使用继承，而是通过委派机制来实现

- **In essence, this refactoring** splits both classes and makes the superclass the helper of the subclass, not its parent.
  - Instead of inheriting all superclass methods, the subclass will have only the necessary methods for delegating to the methods of the superclass object. 一个类不需要继承另一个类的全部方法，通过委托机制调用部分方法
  - A class does not contain any unneeded methods inherited from the superclass. 从而避免大量无用的方法

Replace Inheritance with Delegation

```
class RealPrinter {
    void print() {
        System.out.println("Printing Data");
    }
}
class Printer extends RealPrinter {
    void print(){
        super.print();
    }
}

Printer printer = new Printer();
printer.print();
```

Inheritance

Delegation

```
class RealPrinter {
    void print() {
        System.out.println("The Delegate");
    }
}
class Printer {
    RealPrinter p = new RealPrinter();

    void print() {
        p.print();
    }
}
Printer printer = new Printer();
printer.print();
```

Composite over inheritance principle复合重用原则

- **Or called Composite Reuse Principle (CRP) 复合重用原则**
  - Classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement desired functionality) rather than inheritance from a base or parent class. 类通过“组合”实现多态和复用（通过引入其他类的实例来实现功能）而不是通过基类或父类。
  - It is better to compose what an object can do (`has_a or use_a`) than extend what it is (`is_a`).

- **Delegation can be seen as a reuse mechanism at the object level, while inheritance is a reuse mechanism at the class level.** “委托”发生在**object**层面，而“继承”发生在**class**层面

CRP example

- **An `Employee` class has a method for computing the employee's annual bonus:**

```
class Employee {
  Money computeBonus() {... // default computation}
  ...
}
```

- Different subclasses of Employee: `Manager`, `Programmer`, `Secretary`, etc. may want to override this method to reflect the fact that some types of employees get more generous bonuses than others: *不同级别的员工都需要继承该类。*

```
class Manager extends Employee {
  @Override
  Money computeBonus() {... // special computation}
  ...
}
```

CRP example

- There are several problems with this solution.
  - All `Manager` objects get the same bonus. What if we wanted to vary the bonus computation among managers? ——To introduce a special subclass of `Manager`?*如果想针对高层计算收入的方法有所区别*

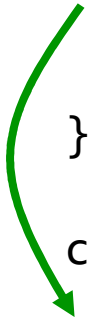
```
class SeniorManager extends Manager {
  @Override
  Money computeBonus() {... // more special computation}
  ...
}
```

  - What if we wanted to change the bonus computation for a particular employee? For example, what if we wanted to promote Smith from `Manager` to `SeniorManager`? *如果只针对某一个人或某几个人？*
  - What if we decided to give all managers the same bonus that programmers get? Should we copy and paste the computation algorithm from `Programmer` to `Manager`? *如果高管和程序员的计算方法一致呢？*

CRP Example

- **The problem is: the bonus calculator for each employer may be different and be changed frequently.**
  - The relation between Employee and bonus calculator should be in the object level instead of class level. 核心问题：每个Employee对象的奖金计算方法都不同，在object层面而非class层面。（逻辑上可以区分）
- **A CRP solution:**

```
class Manager {
  ManagerBonusCalculator mbc = new ManagerBonusCalculator();
  Money computeBonus() {
    return mbc.computeBonus();
  }
}

class ManagerBonusCalculator {
   Money computeBonus {... // special computation}
}
```



And so does for class Programmer and others

CRP example: more general design

```
class Employee {
   BonusCalculator bc;
   ...
}
```

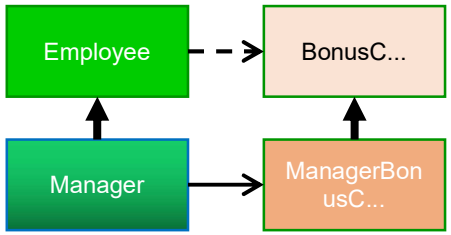
Delegation：委托

```
interface BonusCalculator {
   Money computeBonus();
}

class Manager extends Employee {
   Money computeBonus() {
      bc = new ManagerBonusCalculator();
      return bc.computeBonus();
   }
}
```

Delegation：object层面的委托

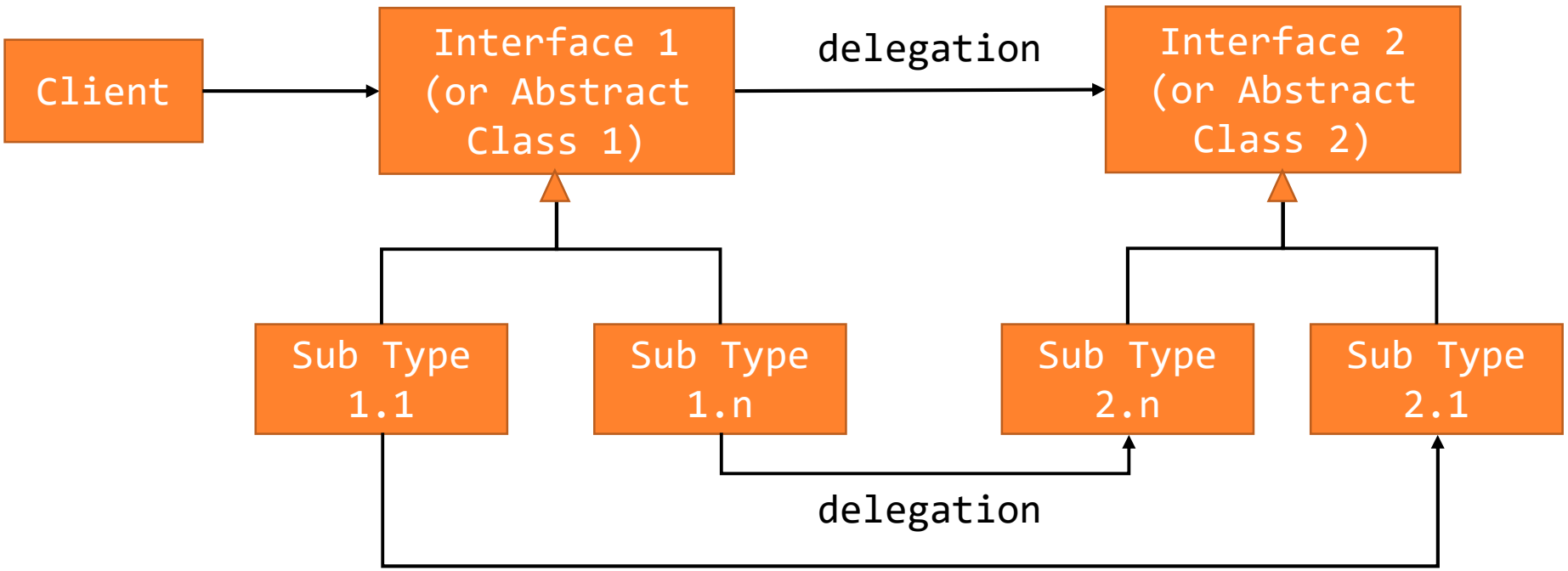
```
class ManagerBonusCalculator implements BonusCalculator {
   Money computeBonus {... // special computation}
}
```

Inheritance：继承



Implement：接口实现

CRP example: more general design



Composite over inheritance principle 更普适的

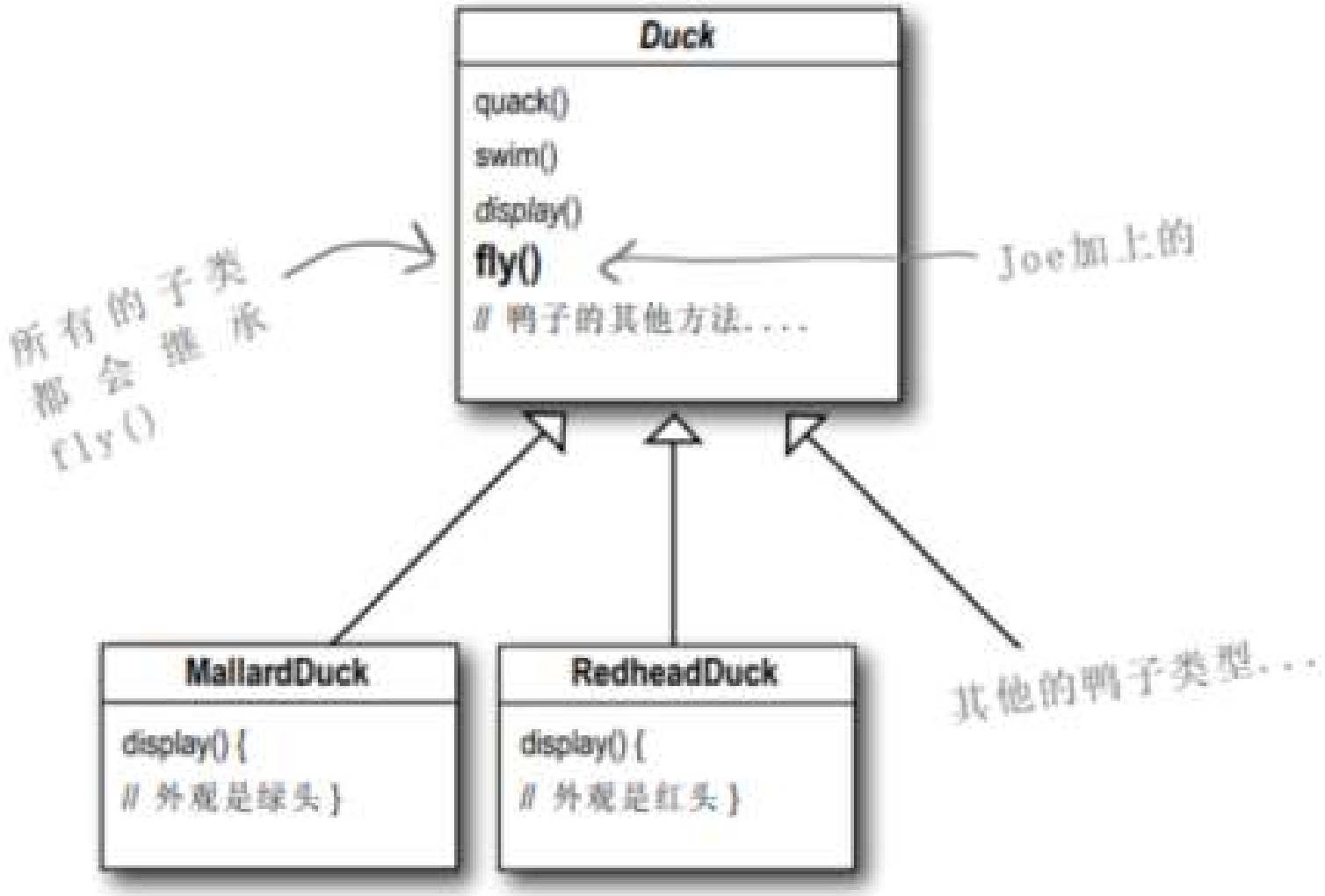
- 一个假象的场景：
  - 你要开发一套动物ADT，各种不同种类的动物，每类动物有自己的独特“行为”，某些行为可能在不同类型的动物之间复用。
  - 考虑到生物学和AI的进展，动物的“行为”可能发生会发生变化；
- 例如：
  - 行为：飞、叫、…
  - 动物：既会飞又会叫的鸭子、天鹅；不会飞但会叫的猫、狗；…
  - 有10余种“飞”的方式、有10余种“叫”的方式；而且持续增加

```
class duck {
   void fly() {…//飞法1}
   void quack() {…//叫法1}
}
```

```
class swan {
   void fly() {…//飞法2}
   void quack() {…//叫法2}
}
```

直接面向具体类型动物的编程：类
缺陷：存在大量的重复；不易变化

根据继承设计


Duck
quack()
swim()
display()
fly()
// 鸭子的其他方法....
所有的子类都会继承fly()
Joe加上的
MallardDuck
display() {
// 外观是绿头 }
RedheadDuck
display() {
// 外观是红头 }
其他的鸭子类型...

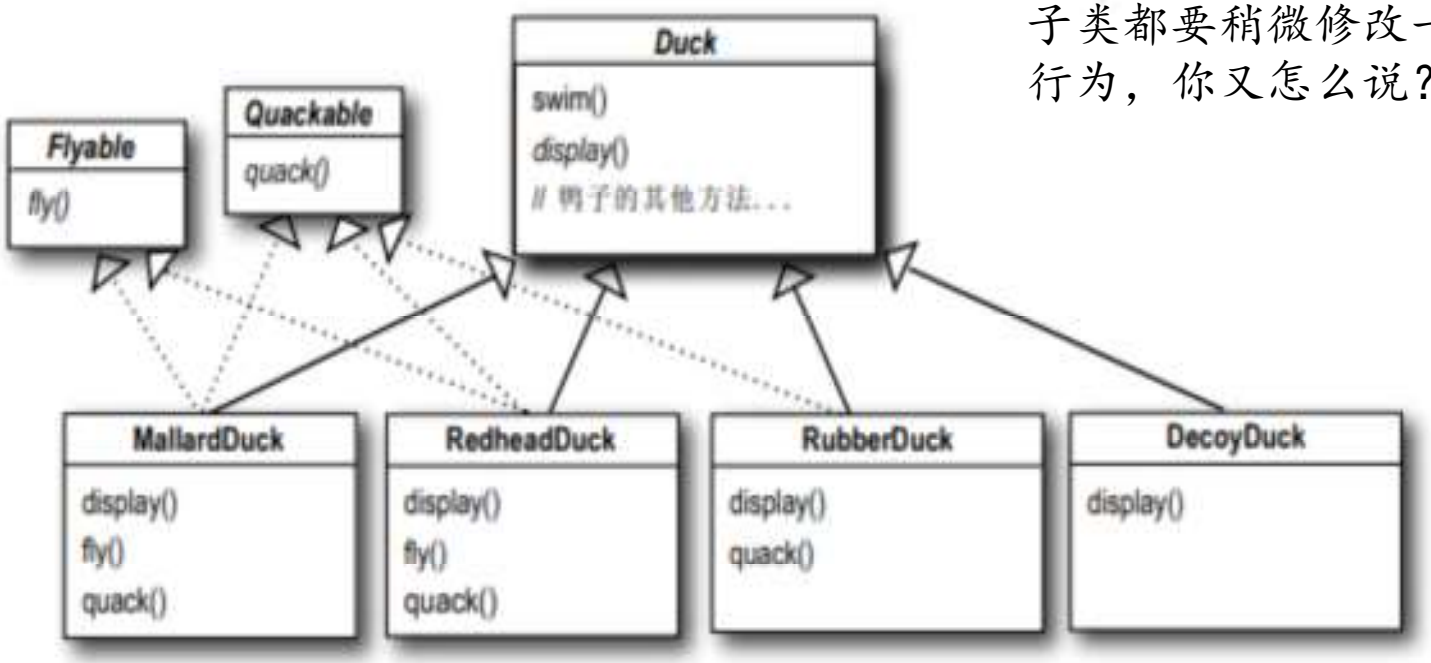
Composite over inheritance principle 更普适的

```
class flyableAnimal {
    //实现通用的飞法
    void fly() {…//飞法1}
    abstract void quack();
}

class duckLikeAnimal extends flyableAnimal {
    @Override //如果某类动物飞法不同，则override重写新“飞法”
    void fly() {…//飞法2}
    @Override //针对该动物的叫法，写具体实现；不会叫则保持实现体为空即可
    void quack() {…//叫法1}
}
```

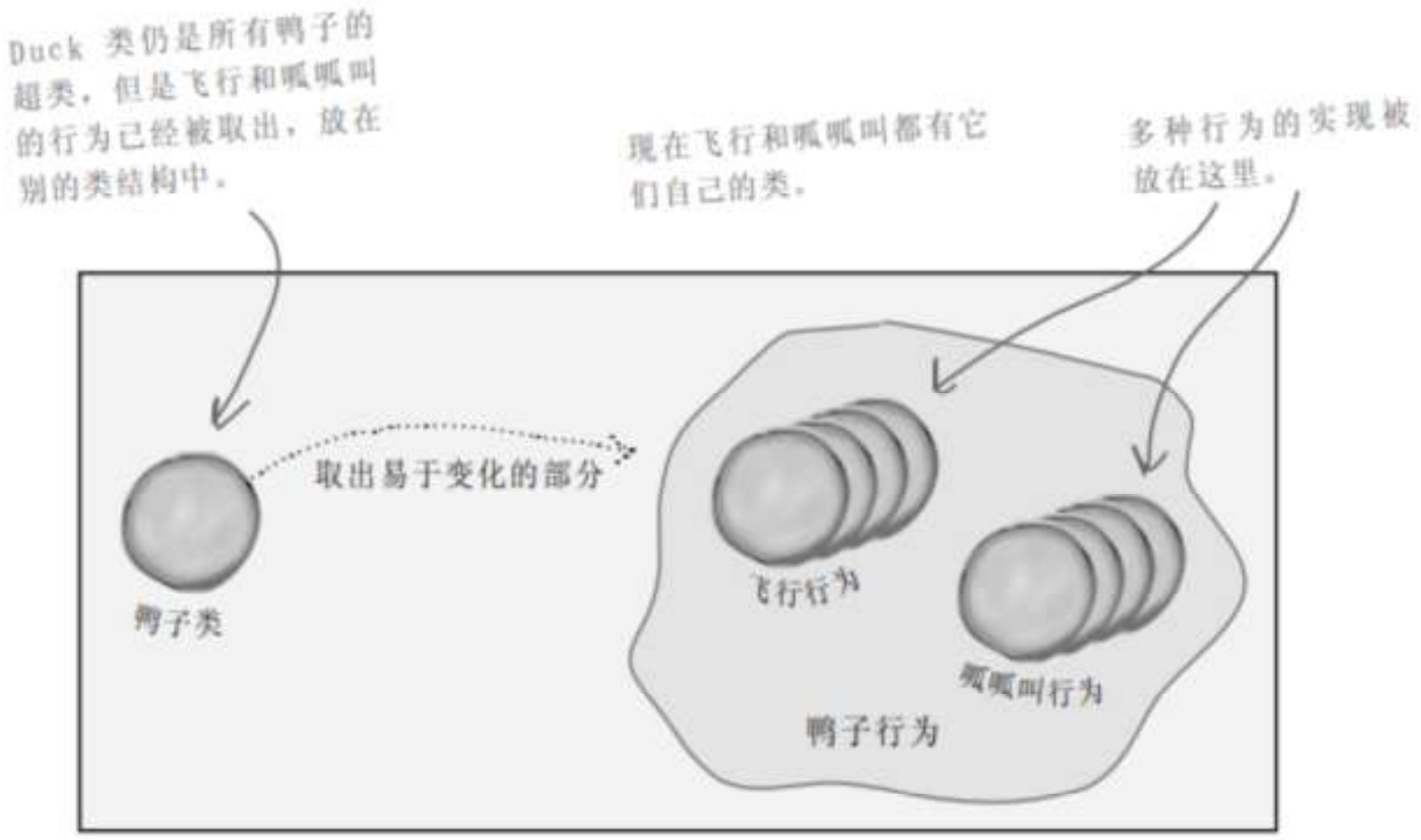
通过inheritance实现对某些通用行为的复用
缺点：需要针对“飞法”设计复杂的继承关系树；不能同时支持针对“叫法”的继承；动物行为发生变化时，继承树要随之变化。

采用继承方法

这真是一个超笨的主意，你没发现这么一来重复的代码会变多吗？
如果你认为覆盖几个方法就算是差劲，那么对于48个Duck的子类都要稍微修改一下飞行的行为，你又怎么说？！

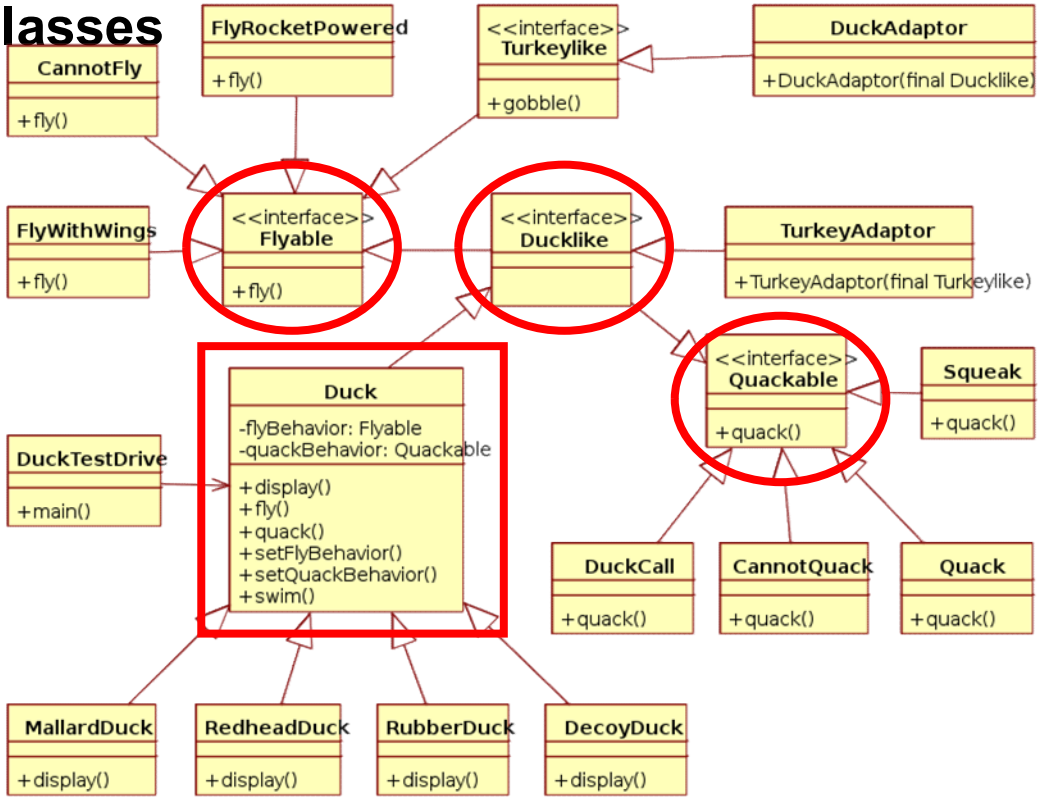


将变化从不变区分出来


Duck 类仍是所有鸭子的超类，但是飞行和呱呱叫的行为已经被取出，放在别的类结构中。
现在飞行和呱呱叫都有它们自己的类。
多种行为的实现被放在这里。
取出易于变化的部分
鸭子类
飞行行为
呱呱叫行为
鸭子行为

Composite over inheritance principle 更普适的

- **An implementation of composition over inheritance typically begins with the creation of various interfaces representing the behaviors that the system must exhibit.**
- **Classes implementing the identified interfaces are built and added to business domain classes as needed.**
- **Thus, system behaviors are realized without inheritance.**
- 使用接口定义系统必须对外展示的不同侧面的行为
- 接口之间通过**extends**实现行为的扩展（接口组合）
- 类**implements** 组合接口
- 从而规避了复杂的继承关系



Composite over inheritance principle

```
interface Flyable {
  public void fly();
}

interface Quackable {
  public void quack();
}
```

两个接口，定义抽象行为

```
class FlyWithWings implements Flyable {
  @Override
  public void fly() {
    System.out.println("fly with wings");
  }
}
class Quack implements Quackable {
  @Override
  public void quack() {
    System.out.println("quack like duck");
  }
}
```

接口的具体实现，可以有多种方式，在具体类中实现具体行为

Composite over inheritance principle

```
interface Ducklike extends Flyable, Quackable {}

public class Duck implements Ducklike {
   Flyable flyBehavior;
   Quackable quackBehavior;

   void setFlyBehavior(Flyable f) {
     this.flyBehavior = f;
   }
   void setQuackBehavior(Quackable q) {
     this.quackBehavior = q;
   }

   @Override
   public void fly() {
     this.flyBehavior.fly();
   }
   @Override
   public void quack() {
     this.quackBehavior.quack();
   }
}
```

接口的组合，定义了行为的组合

Delegation

从组合接口中派生具体类

设置Delegation对象实例

对接口编程

通过Delegation实现具体行为

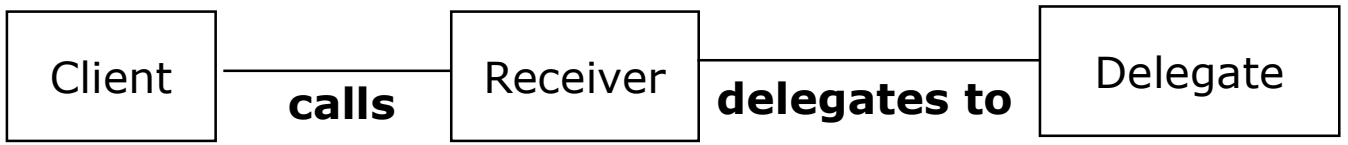
Client code:

```
Flyable f = new FlyWithWings();
Quackable q = new Quack();

Duck d = new Duck();
d.setFlyBehavior(f);
d.setQuackBehavior(q);
d.fly();
d.quack();
```

Types of delegation

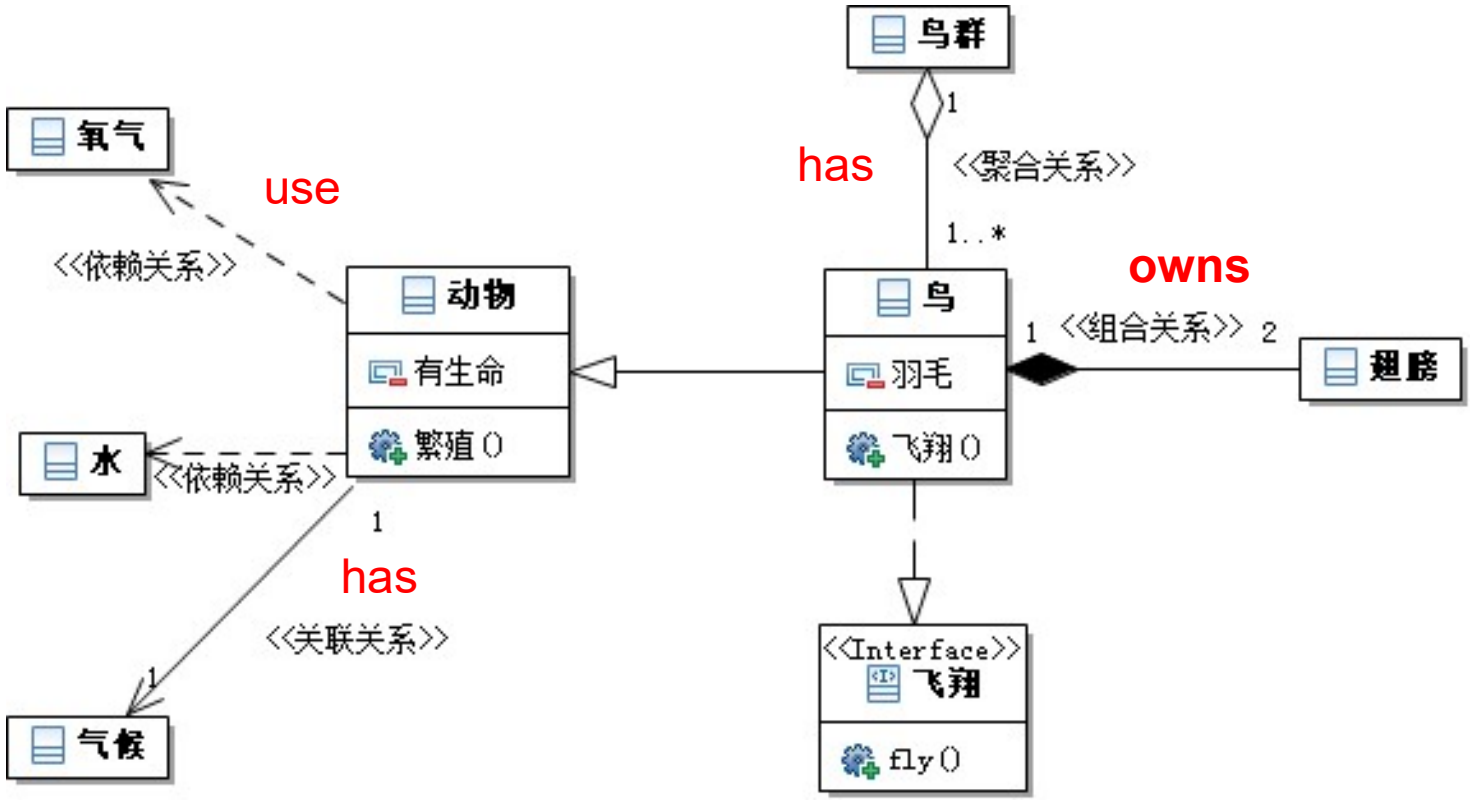
- Use **(A use B)** 使用
- Association **(A has B)** 关联
- Composition/aggregation **(A owns B)** 组合/聚合
  - 可以认为Composition/Aggregation是Association的两种具体形态

- **This classification is in terms of the "coupling degree耦合度" between the delegate and delegator.** *在委托人和被委托人之间*



Use
Composition
Aggregation
Association

Types of delegation



(1) Dependency: 临时性的delegation

- The simplest form of using classes is calling its methods;
- This form of relationship between two classes is called "uses-a" relationship, in which one class makes use of another without actually incorporating it as a property. -it may, for example, be a parameter or used locally in a method.
- **Dependency:** a temporary relationship that an object requires other objects (suppliers) for their implementation.
- 依赖：对象之间的动态的、临时的通信联系

```
Flyable f = new FlyWithWings();
Quackable q = new Quack();

Duck d = new Duck();

d.fly(f);
d.quack(q);
```

```
class Duck {
  //no field to keep Flyable object

  void fly(Flyable f) {
    f.fly();
  }
}
```

仅仅是在参数中出现，调用了相应方法，甚至都没有实例化对象

(2) Association: 永久性的delegation

- **Association:** a persistent relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf. 关联：对象之间的长期静态联系——has a
  - has_a: one class has another as a property/instance variable
  - This relationship is structural, because it specifies that objects of one kind are connected to objects of another and does not represent behavior.

```
Flyable f = new FlyWithWings();

Duck d = new Duck(f);
Duck d2 = new Duck();

d.fly();
```

```
class Duck {
  Flyable f = new CannotFly();

  void Duck(Flyable f) {
    this.f = f;
  }
  void Duck() {
    f = new FlyWithWings();
  }
  void fly() { f.fly(); }
}
```

一个类有另一个作为属性/实例变量，
但彼此仍可分

(3) Composition: 更强的association，但难以变化

- **Composition** is a way to combine simple objects or data types into more complex ones.
  - `is_part_of`: one class has another as a property/instance variable
  - Implemented such that an object contains another object.

不同于一般的Association，这里变成了其一部分属性，并且彼此不可分，在内部创建。

```
Duck d = new Duck();
```

```
class Duck {
  Flyable f = new FlyWithWings();

  void fly() {
    f.fly();
  }
}
```

```
d.fly();
```

(4) Aggregation：更弱的association，可动态变化

- **Aggregation:** the object exists outside the other, is created outside, so it is passed as an argument to the construtor.
  - has_a 对象存在于另一个对象之外，是在外部创建的，因此它作为参数传递给构造函数。

聚合：部分与整体的关系，但彼此可分——owns a

```
Flyable f = new FlyWithWings();

Duck d = new Duck(f);

d.fly();

d.setFlyBehavior(new CannotFly());

d.fly();
```

```
class Duck {
  Flyable f;

  void Duck(Flyable f) {
    this.f = f;
  }
  void setFlyBehavior(f) {
    this.f = f;
  }
  void fly() { f.fly();}
}
```

Composition vs. Aggregation

- **In composition, when the owning object is destroyed, so are the contained objects.**
  - A university owns various departments, and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist.

- **In aggregation, this is not necessarily true.**
  - A `University` can be seen as a composition of `departments`, whereas `departments` have an aggregation of `professors`. A `Professor` could work in more than one `department`, but a `department` could not be part of more than one `university`.

Composition vs. Aggregation

```
public class WebServer {
  private HttpListener listener;
  private RequestProcessor processor;
  public WebServer(HttpListener listener, RequestProcessor processor) {
    this.listener = listener;
    this.processor = processor;
  }
}                    //Aggregation
```

Which is Composition?
Which is Aggregation?

```
public class WebServer {
  private HttpListener listener;
  private RequestProcessor processor;
  public WebServer() {
    this.listener = new HttpListener(80);
    this.processor = new RequestProcessor( “/www/root” );
  }
}                                  //Composition
```

Types of delegation

- Use (A use one or multiple B)
- Association (A has one or multiple B)
- Composition/aggregation (A owns one or multiple B)
- 都支持1对多的delegation——

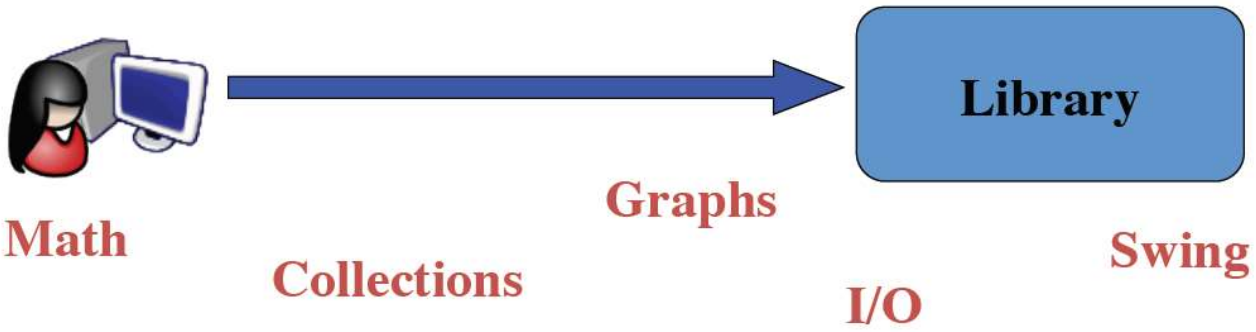
```
class Professor {
  List<Student> ls;              //永久保存delegation关系

  void enroll(Student s) {
    this.ls.add(s);              //建立delegation关系
  }
  void evaluate() {
    double score = 0;
    for(Student s : ls)
       score += s.evaluate(this);  //逐个delegate
  }
}
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

6 Designing system-level reusable API libraries and Frameworks

Libraries

- **Library:** A set of classes and methods (APIs) that provide reusable functionality



Why is API design important?

- **If you program, you are an API designer, and APIs can be among your greatest assets API是程序员最重要的资产和“荣耀”，吸引外部用户，提高声誉**
  - Good code is modular – each module has an API
  - Users invest heavily: acquiring, writing, learning
  - Thinking in terms of APIs improves code quality
  - Successful public APIs capture users
- **Can also be among your greatest liabilities**
  - Bad API can cause unending stream of support calls
  - Can inhibit ability to move forward
- **Public APIs are forever – one chance to get it right**
  - Once module has users, can’t change API at will

建议：始终以开发API的标准面对任何开发任务

面向“复用”编程
而不是面向“应用”编程

难度：要有足够良好的设计，一旦发布就无法再自由改变

Whitebox and Blackbox frameworks

- **Whitebox frameworks**
  - Extension via subclassing and overriding methods
  - Common design pattern(s): Template Method
  - Subclass has main method but gives control to framework

- **Blackbox frameworks**
  - Extension via implementing a plugin interface
  - Common design pattern(s): Strategy, Observer
  - Plugin-loading mechanism loads plugins and gives control to the framework

A calculator example (without a framework)


```
public class Calc extends JFrame {
  private JTextField textField;
  public Calc() {
      JPanel contentPane = new JPanel(new BorderLayout());
      contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
      JButton button = new JButton();
      button.setText("calculate");
      contentPane.add(button, BorderLayout.EAST);
      textField = new JTextField("");
      textField.setText("10 / 2 + 6");
      textField.setPreferredSize(new Dimension(200, 20));
      contentPane.add(textfield, BorderLayout.WEST);
      button.addActionListener(/* calculation code */);
      this.setContentPane(contentPane);
      this.pack();
      this.setLocation(100, 100);
      this.setTitle("My Great Calculator");
      ...
  }
}
```

A simple whitebox framework

```
public abstract class Application extends JFrame {
    protected String getApplicationTitle() { return ""; }
    protected String getButtonText() { return ""; }
    protected String getInitialText() { return ""; }
    protected void buttonClicked() { }
    private JTextField textField;
    public Application() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(getButtonText());
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        textField.setText(getInitialText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        button.addActionListener((e) -> { buttonClicked(); });
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
        ...
    }
```

子类中由开发者通过override完成定制的功能

Using the whitebox framework

Overriding

```
public class Calculator extends Application {
  protected String getApplicationTitle() { return "My Great Calculator"; }
  protected String getButtonText() { return "calculate"; }
  protected String getInititalText() { return "(10 – 3) * 6"; }
  protected void buttonClicked() {
    JOptionPane.showMessageDialog(this, "The result of " + getInput() +
        " is " + calculate(getInput()));
  }
  private String calculate(String text) { ... }
}
```

Extension via subclassing and overriding methods
Subclass has main method but gives control to framework

```
public class Ping extends Application {
  protected String getApplicationTitle() { return "Ping"; }
  protected String getButtonText() { return "ping"; }
  protected String getInititalText() { return "127.0.0.1"; }
  protected void buttonClicked() { ... }
}
```

Overriding

An example blackbox framework

```
public interface Plugin {
  String getApplicationTitle();
  String getButtonText();
  String getInititalText();
  void buttonClicked() ;
  void setApplication(Application app);
}
```

```
public class Application extends JFrame {
   private JTextField textField;
   private Plugin plugin;
   public Application() { }
   protected void init(Plugin p) {
     p.setApplication(this);
     this.plugin = p;
     JPanel contentPane = new JPanel(new BorderLayout());
     contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
     JButton button = new JButton();
     button.setText(plugin != null ? plugin.getButtonText() : "ok");
     contentPane.add(button, BorderLayout.EAST);
     textField = new JTextField("");
     if (plugin != null)
         textField.setText(plugin.getInititalText());
     textField.setPreferredSize(new Dimension(200, 20));
     contentPane.add(textField, BorderLayout.WEST);
     if (plugin != null)
         button.addActionListener((e) -> { plugin.buttonClicked(); } );
     this.setContentPane(contentPane);
    ...
  }
  public String getInput() { return textField.getText(); }
}
```

Using the blackbox framework

```
public class CalcPlugin implements Plugin {
  private Application app;
  public void setApplication(Application app) { this.app = app; }
  public String getButtonText() { return "calculate"; }
  public String getInititalText() { return "10 / 2 + 6"; }
  public void buttonClicked() {
      JOptionPane.showMessageDialog(null, "The result of "
              + application.getInput() + " is "
              + calculate(application.getInput()));
  }
  public String getApplicationTitle() { return "My Great Calculator"; }
}
```

Extension via implementing a plugin interface
Plugin-loading mechanism loads plugins and gives control to the framework

Another example of white-box framework

```
public abstract class PrintOnScreen {

  public void print() {
    JFrame frame = new JFrame();
    JOptionPane.showMessageDialog(frame, textToShow());
    frame.dispose();
  }
  protected abstract String textToShow();
}

public class MyApplication extends PrintOnScreen {

  @Override
  protected String textToShow() {
    return "printing this text on "
        + "screen using PrintOnScreen "
        + "white Box Framework";
  }
}
```

Main body of framework REUSE

Extension point

Overriding

```
MyApplication m = new MyApplication();
m.print();
```

Another example of black-box framework

```
public final class PrintOnScreen {

  TextToShow textToShow;
  public PrintOnScreen(TextToShow tx) {
    this.textToShow = tx;
  }

  public void print() {
    JFrame frame = new JFrame();
    JOptionPane.showMessageDialog(frame,
            textToShow.text());
    frame.dispose();
  }
}
public interface TextToShow {
  String text();
}
```

Extension point by composition

```
public class MyTextToShow
    implements TextToShow {

  @Override
  public String text() {
    return "Printing";
  }
}
```

Plugin

```
PrintOnScreen m =
  new PrintOnScreen(new MyTextToShow());
m.print();
```

Another example of black-box framework

```
public final class PrintOnScreen {

  TextToShow textToShow;
  public PrintOnScreen(TextToShow tx) {
    this.textToShow = tx;
  }

  public void print() {
    JFrame frame = new JFrame();
    JOptionPane.showMessageDialog(frame,
            textToShow.text());
    frame.dispose();
  }
}
public interface TextToShow {
  String text();
}
```

```
public class MyTextToShow
   implements TextToShow {

  @Override
  public String text() {
    return "Printing";
  }
}
```

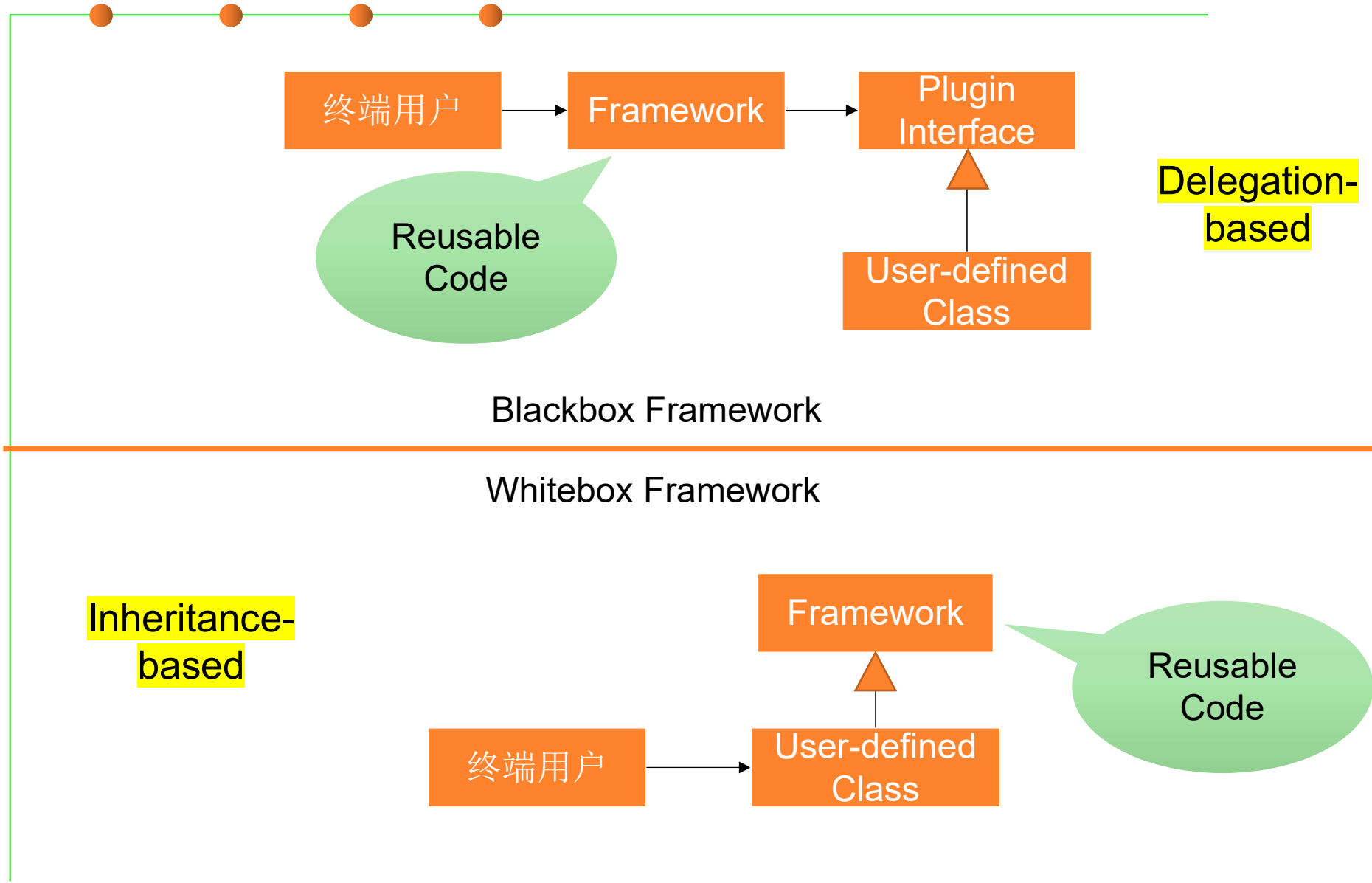
如果有多个plugin怎么办？
如果通过配置文件选择加载某个plugin，怎么做？

```
PrintOnScreen m =
  new PrintOnScreen(new MyTextToShow());
m.print();
```

Whitebox vs. Blackbox Frameworks

- **Whitebox frameworks use subclassing/subtyping ---继承**
  - Allows extension of every non-private method
  - Need to understand implementation of superclass
  - Only one extension at a time
  - Compiled together
  - Often so-called developer frameworks
- **Blackbox frameworks use composition -- 委派/组合**
  - Allows extension of functionality exposed in interface
  - Only need to understand the interface
  - Multiple plugins
  - Often provides more modularity
  - Separate deployment possible (.jar, .dll, …)
  - Often so-called end-user frameworks, platforms

Whitebox vs. Blackbox Frameworks



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Summary

Summary

- **What is software reuse?**
- **How to measure “reusability”?**
- **Levels and morphology of reusable components**
  - Source code level reuse
  - Module-level reuse: class/interface
  - Library-level: API/package
  - System-level reuse: framework

Summary

- **Designing reusable classes**
  - Inheritance and overriding
  - Overloading
  - Parametric polymorphism and generic programming
  - Behavioral subtyping and Liskov Substitution Principle (LSP)
  - Composition and delegation

- **Designing system-level reusable libraries and frameworks**
  - API and Library
  - Framework

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

June 16, 2021