



Chapter 5: Reusability-Oriented Software Construction Approaches

5.1 Metrics, Morphology and External Observations of Reusability


Xu Hanchuan
xhc@hit.edu.cn

April 3, 2018

Outline

- **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
- **External observations of reusability**
 - Type Variation
 - Routine Grouping
 - Implementation Variation
 - Representation Independence
 - Factoring Out Common Behaviors
- **Summary**

Objective of this lecture

- 
- To discuss the **advantages and disadvantages** of software reuse
 - To describe **construction for reuse**
 - To discuss the **characteristics of generic reusable components**
 - To describe **methods of developing portable application systems**



1 What is Software Reuse?



Software reuse 软件复用/重用

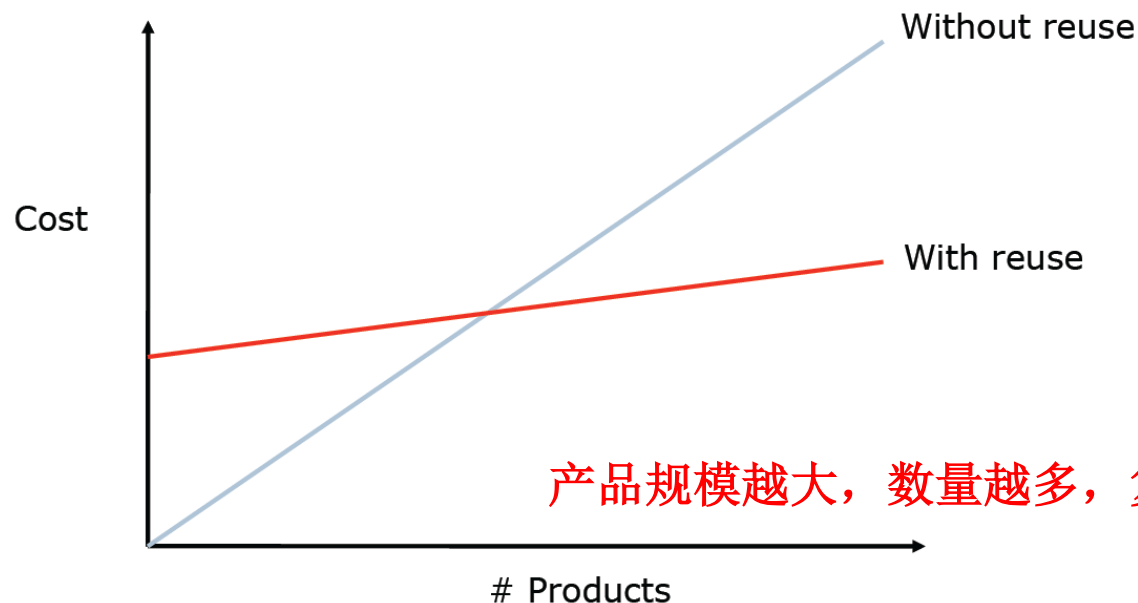
- **Software reuse is the process of implementing or updating software systems using existing software components.**
- **Two perspectives of software reuse**
 - Creation: creating reusable resources in a systematic way (**programming for reuse**) 为了复用编程
 - Use: reusing resources as building blocks for creating new systems (**programming with reuse**) 基于复用编程
- **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). “创造可重复使用而不是过渡性器物的动力，来自审美、知识和经济动机，以及人类对不朽的渴望。它将人与其他生物区分开来，体现了文明社会与原始社会的区别”

Why reuse?

- **Reuse is cost-effective and with timeliness 成本有效性和及时性**
 - 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 可生成可靠的软件**
 - 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**
 - Reuse of GUI libraries produces common look-and-feel in applications.
 - Consistency with regular, coherent design.

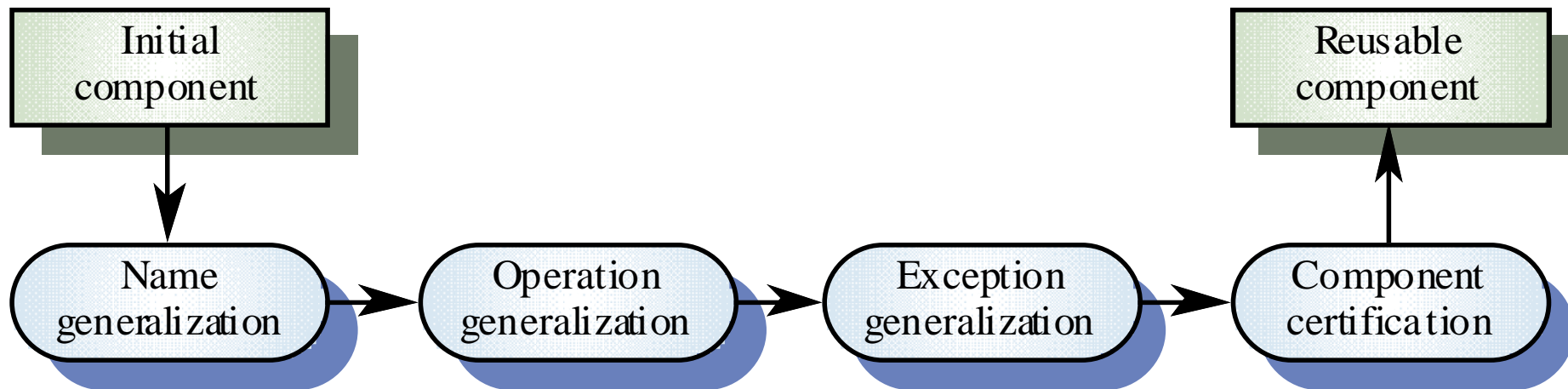
Reuse costs

- 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.
- **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.



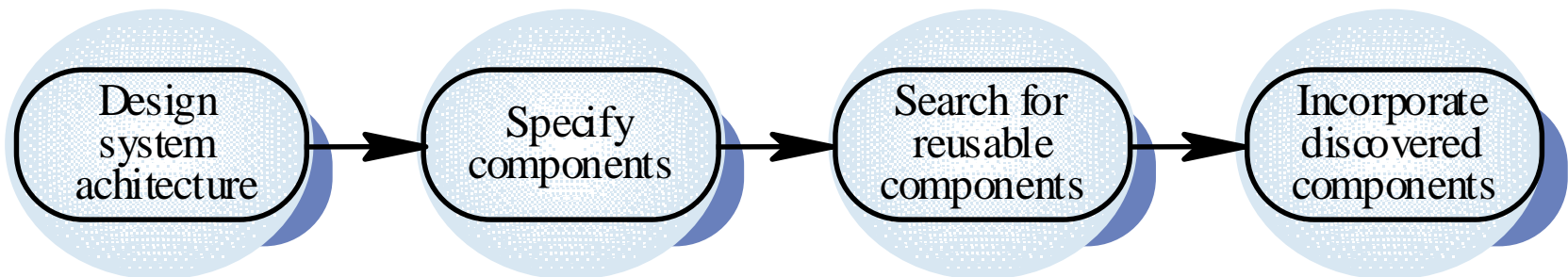
Development for reuse 面向复用开发

- 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.
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents.



Development with reuse 基于复用开发

- **Component management tools, such as repositories, for architectures, designs, documentation, and code must be developed and maintained.**



- **A key issue: adaptation of reusable components 适应问题**
 - Extra functionality may have to be added to a component. When this has been added, the new component may be made available for reuse. 增加功能
 - Unneeded functionality may be removed from a component to improve its performance or reduce its space requirements 削减功能
 - The implementation of some component operations may have to be modified. 修改功能



2 How to measure “reusability”?



Measure resuability

- **How frequently** can a software asset be reused in different application scenarios?
 - 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?
 - Cost to buy the asset and other mandatory libraries
 - Cost for adapting it
 - Cost for instantiating it
 - Cost for changing other parts of the system that interact with it

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) 简单
 - Portable and Standard Compliance 可移植性和兼容性好
 - Adaptable and Flexible 灵活
 - Extensibility 可扩展
 - Generic and Parameterization 通用和参数化
 - Modularity 模块化
 - Localization of volatile (changeable) design assumptions 将变化限制在局部
 - Stability under changing requirements 稳定



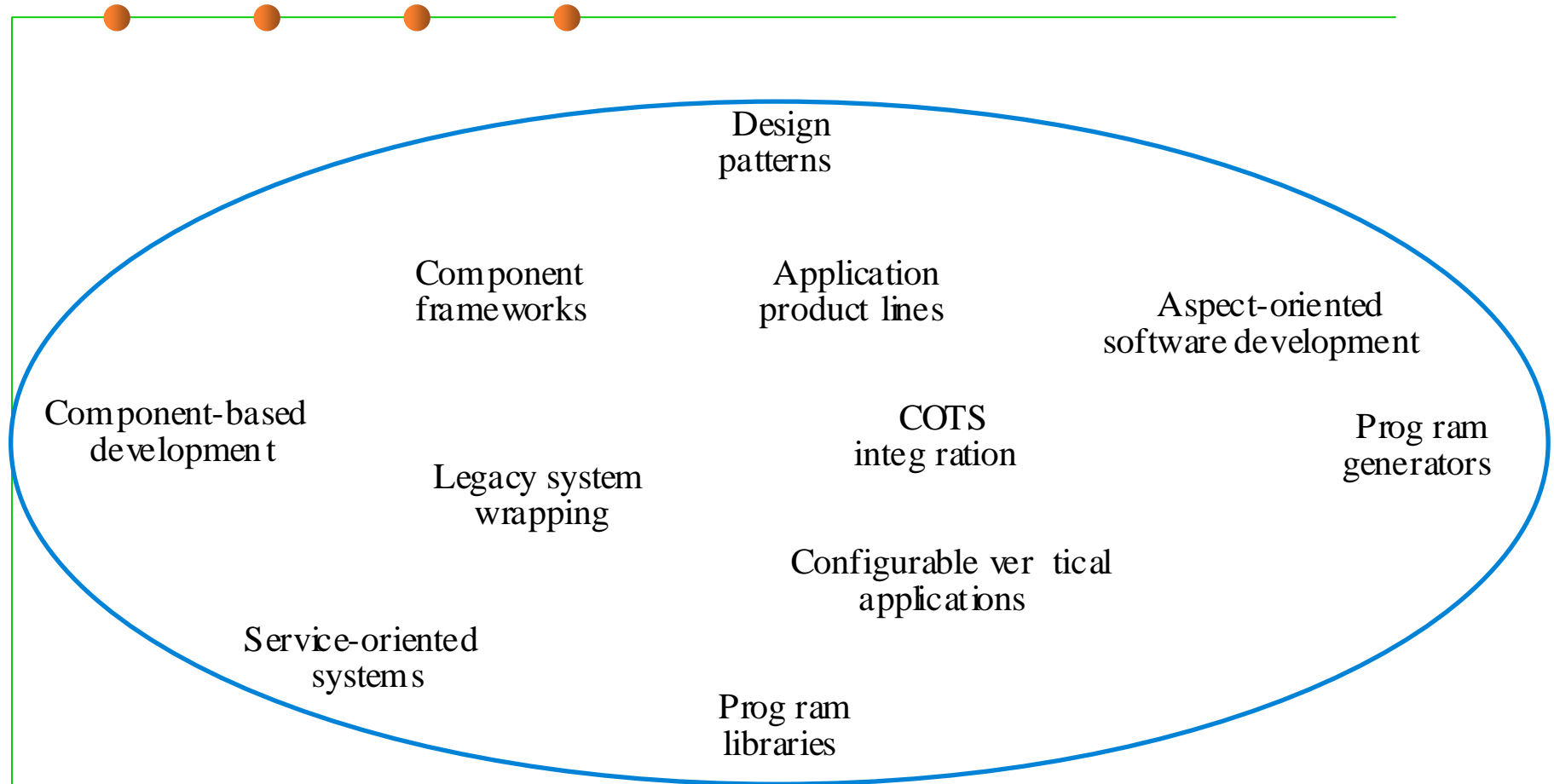
3 Levels and morphology(形态) of reusable components



Levels of Reuse

- 
- **A reusable component may be code**
 - Most prevalent(普遍的): what most programmers relate with reuse
 - **But benefits result from a broader and higher-level view of what can be reused:**
 - Requirements
 - Design and specifications
 - Data
 - Test cases
 - Documentation

The reuse landscape



Reuse morphology

Design patterns

Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.

Component-based development

Systems are developed by integrating components (collections of objects) that conform to component-model standards.

Application frameworks

Collections of abstract and concrete classes that can be adapted and extended to create application systems.

Legacy system wrapping

Legacy systems that can be ‘wrapped’ by defining a set of interfaces and providing access to these legacy systems through these interfaces.


Service-oriented systems

Systems are developed by linking shared services that may be externally provided.

Reuse morphology

Application product lines	An application type is generalised around a common architecture so that it can be adapted in different ways for different customers.
COTS integration	Systems are developed by integrating existing application systems.
Configurable vertical applications	A generic system is designed so that it can be configured to the needs of specific system customers.
Program libraries	Class and function libraries implementing commonly-used abstractions are available for reuse.
Program generators	A generator system embeds knowledge of particular types of application and can generate systems or system fragments in that domain.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled.

What we concern in this lecture

- 
- **Source code level: methods, statements, etc**
 - **Module level: class and interface**
 - **Library level: API**
 - Java Library
 - Maven
 - **System level: frameworks**

Types of Code Reuse



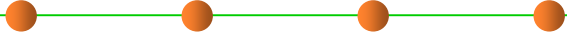
■ White box reuse

- Reuse of code when code itself is available. Usually requires some kind of modification or adaptation

■ Black box reuse

- 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

Formats for reusable component distribution

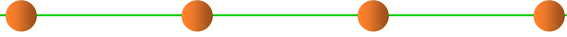
- 
-
- **Source code**
 - **Package such as .jar**



(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)
 - **High risk of error during process**
 - **May require knowledge about how the used software works**
 - **Requires access to source code**



(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)
- Encapsulation helps reuse
- Less code to manage
- Versioning, backwards-compatibility(兼容旧版本) still problem
- Need to package related classes together

Approaches of reusing a class: **inheritance**

- Java provides a way of code reuse named *Inheritance*
- In inheritance, classes **extend** the properties/behavior of existing classes
- In addition, they might **override/redefine** existing behavior
- No need to put dummy methods that just forward or delegate work(虚拟方法: 通过调用或者转发实现功能的方法)
- Captures the real world better
- Usually need to design inheritance hierarchy before implementation
- Cannot cancel out properties or methods, so must be careful not to overdo it

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)
 - e.g. Sorter is delegating functionality to some Comparator
- **Judicious delegation enables code reuse**
 - Sorter can be reused with arbitrary sort orders
 - Comparators can be reused with arbitrary client code that needs to compare integers
- **Explicit delegation:** passing the sending object to the receiving object
- **Implicit delegation:** by the member lookup rules of the language
- **Delegation** can be described as a low level mechanism for sharing code and data between entities.

Using delegation to extend functionality

- Consider `java.util.List`

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

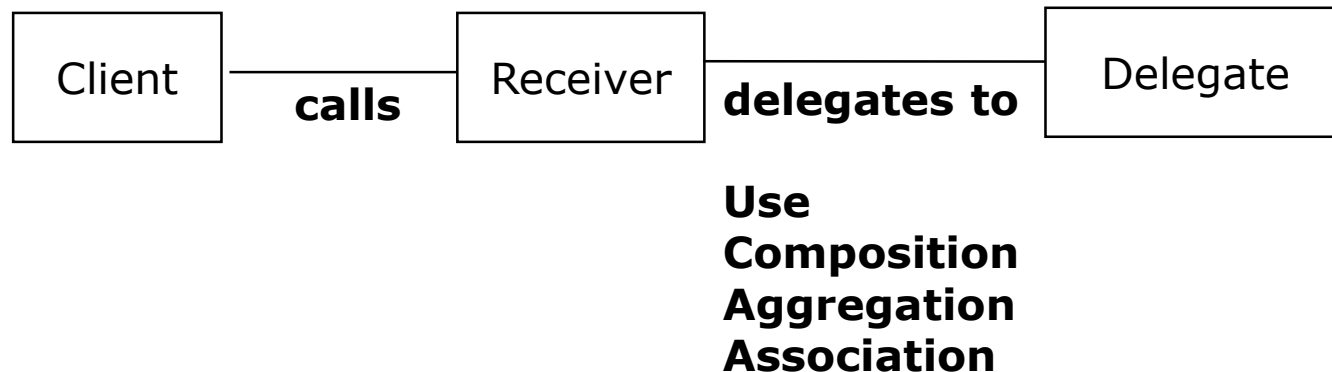
- Suppose we want a list that logs its operations to the console...

- The `LoggingList` is composed of a `List`, and delegates (the non-logging) functionality to that `List`. (实现一个可将操作日志输出到控制台的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);
    }
}
```

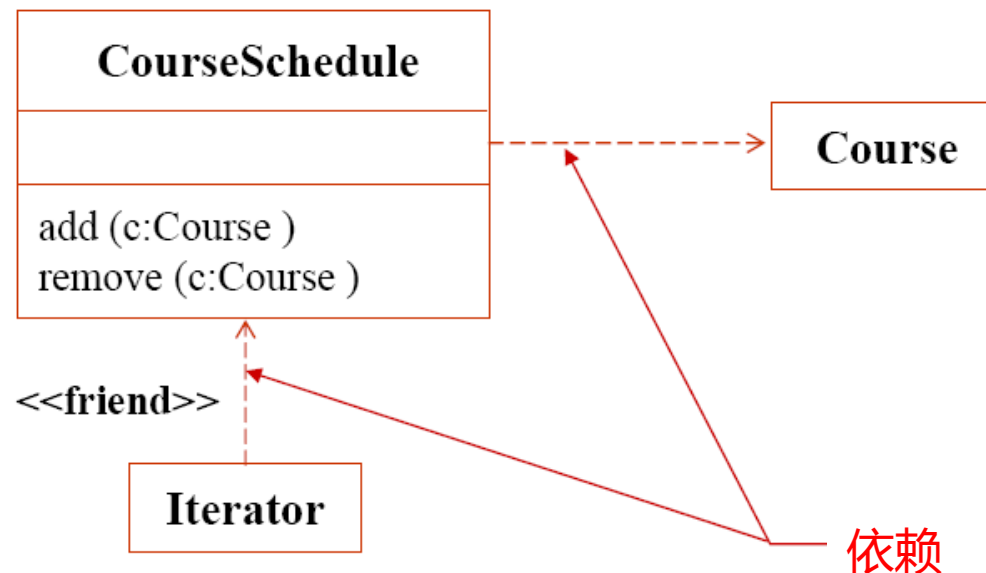
Types of delegation

- Use (A use B)
- Composition/aggregation (A owns B)
- Association (A has B)



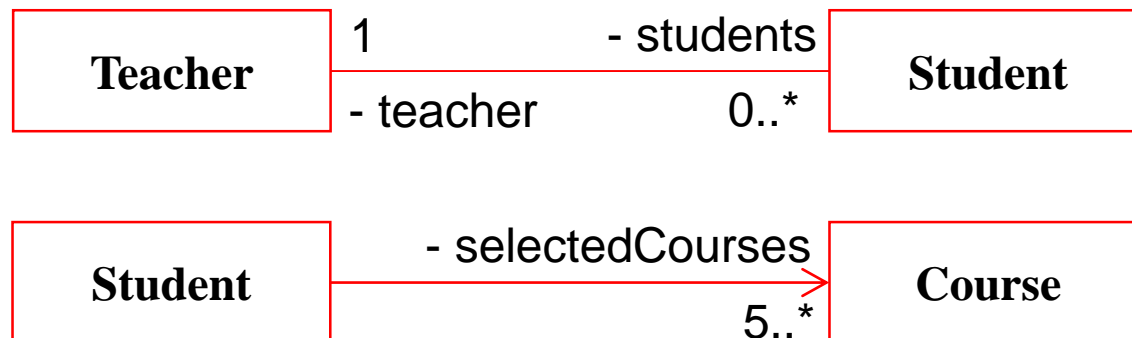
Approaches of reusing a class: use

- The simplest form of using classes is **calling its methods**;
- This form of relationship between two classes is called “**uses-a**” relationship
- Uses (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**)



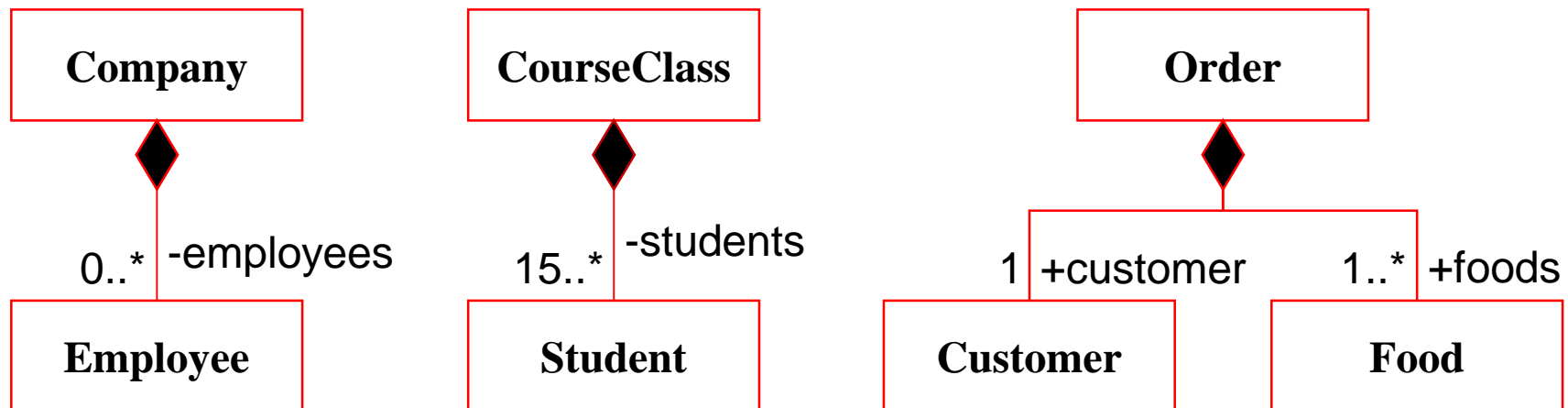
Approaches of reusing a class: association

- A closer form of reuse is association
- Association (or **has_a** in which one class has another as a property/instance variable)



Approaches of reusing a class: composition

- Another closer form of reuse is composition
- Composition (or **owns_a** in which one class has another as a property/instance variable)



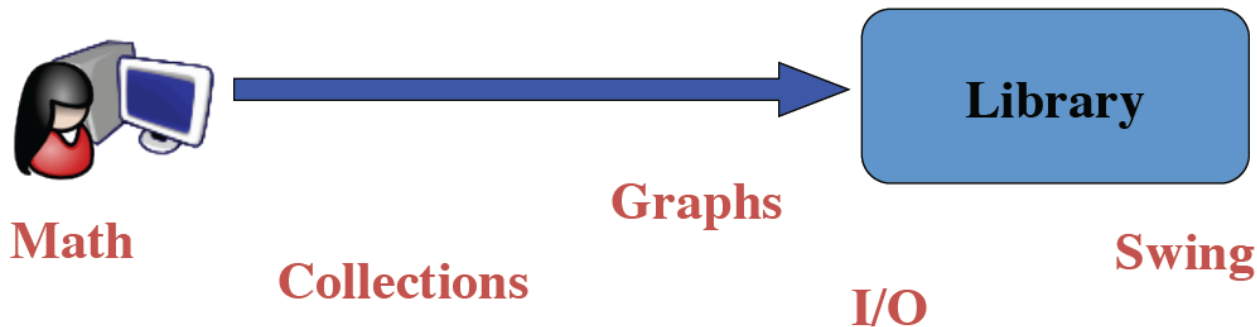


(3) Library-level reuse: API/Package




Libraries

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



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**



(4) System-level reuse: Framework



Application Frameworks

- 
- ***Frameworks*** are for sub-system design containing collection of abstract and concrete classes along with interfaces between each class.
 - A sub-system is implemented by adding components to fill in missing design elements and by instantiating the abstract classes
 - Frameworks are reusable entities

Frameworks

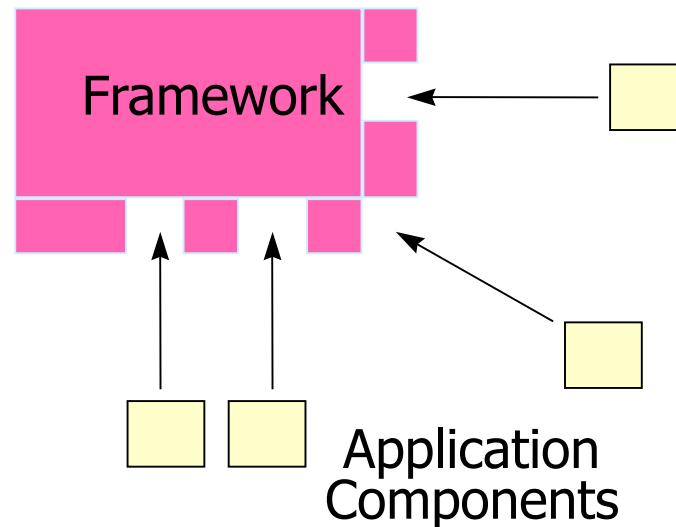
- **A framework is a reusable partial application that can be specialized to produce custom applications.**
 - Frameworks are targeted to particular technologies, such as data processing or cellular communications, or to application domains, such as user interfaces or real-time avionics.
- **The key benefits of frameworks are reusability and extensibility.**
 - Reusability leverages of the application domain knowledge and prior effort of experienced developers
 - Extensibility is provided by hook methods, which are overwritten by the application to extend the framework.
 - Hook methods systematically decouple the interfaces and behaviors of an application domain from the variations required by an application in a particular context. (Hook methods 钩子方法： 是对于抽象方法或者接口中定义的方法的一个空实现， 允许需要时通过继承， override将具体实现挂载上)

Extending Frameworks


- **Generic frameworks need to be extended to create specific applications or sub-systems**
- **Frameworks can be extend by**
 - defining concrete classes that inherit operations from abstract class ancestors 定义具体类从抽象类继承操作
 - adding methods that will be called in response to events recognized by the framework 增加能够被框架调用的方法
- **Frameworks are extremely complex and it takes time to learn to use them (e.g. DirectX or MFC)**

Object-Oriented Frameworks

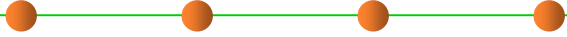
- The reusable design of a system or subsystem implemented through a set of classes and their collaborations.
- Users complete or extend the framework by adding or customizing application specific components to produce an application.




Users and Developers of Frameworks

- 
- **There are three main roles associated with frameworks:**
 - **Framework designers**, also called framework developers or framework builders, develop the original framework
 - **Framework users**, also called framework clients or application developers, use the framework to develop applications.
 - **Framework maintainers** refine and redevelop the framework to fit new requirements.

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.
 - to accommodate the family of problems, the framework is incomplete, incorporating hot spots and hooks to allow customization
 - **Frameworks must be designed for flexibility, extensibility, completeness and ease of use.**

Classification of Frameworks

- 
- **Frameworks can be classified by their position in the software development process.**
 - **Frameworks can also be classified by the techniques used to extend them.**
 - Whitebox frameworks
 - Blackbox frameworks

White-box and Black-Box Frameworks

■ Whitebox frameworks:

- 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.

(通过继承和动态绑定实现可扩展性，通过继承框架基类并重写预定义的钩子方法来扩展现有功能)

■ Blackbox frameworks

- 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.

(通过为可插入框架的组件定义接口来实现可扩展性，通过定义符合特定接口的组件来重用现有功能，这些组件通过委托与框架集成)

Class libraries and Frameworks



■ Class Libraries:

- Less domain specific
- Provide a smaller scope of reuse.
- Class libraries are passive; no constraint on control flow.

■ Framework:

- Classes cooperate for a family of related applications.
- Frameworks are active; affect the flow of control.

■ In practice, developers often use both:

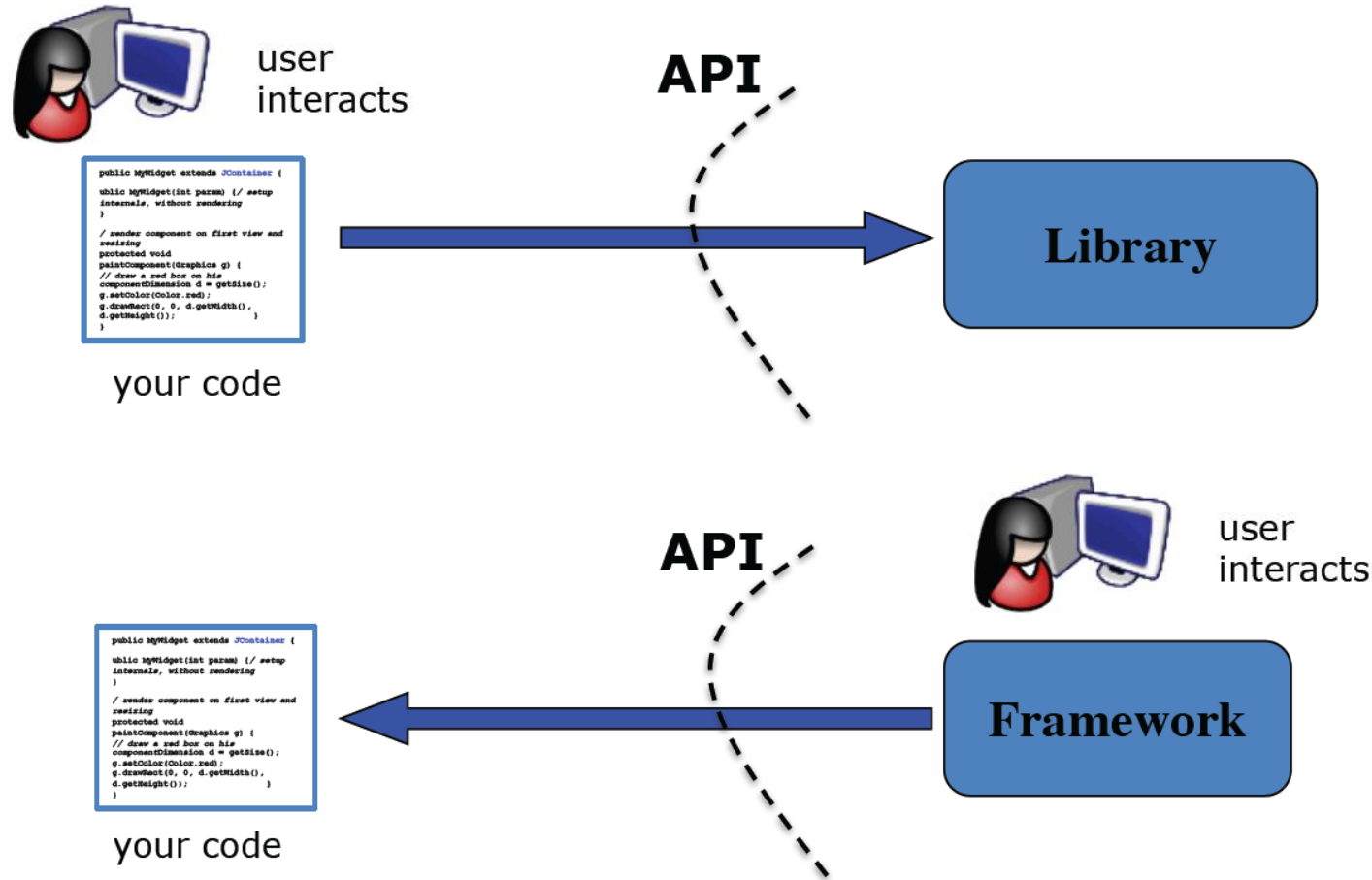
- Frameworks often use class libraries internally to simplify the development of the framework.
- Framework event handlers use class libraries to perform basic tasks (e.g. string processing, file management, numerical analysis....)

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



Components and Frameworks

■ Components

- Self-contained instances of classes
- Plugged together to form complete applications.
- Blackbox that defines a cohesive set of operations,
- Can be used based on the syntax and semantics of the interface.
- Components can even be reused on the binary code level.
 - The advantage is that applications do not always have to be recompiled when components change.

■ Frameworks:


- Often used to develop components
- Components are often plugged into blackbox frameworks.



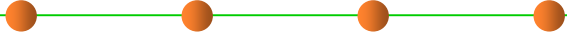
4 External observations of reusability



External observations of reusability

- 
- **Type Variation**
 - **Routine Grouping**
 - **Implementation Variation**
 - **Representation Independence**
 - **Factoring Out Common Behaviors**

Type Variation

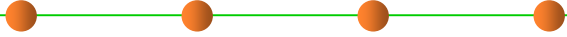
- 
- Reusable components should be **type-parameterized** so that they can adapt to different data types (input, computation, and output);
 - Genericity: reusable components should be **generic**.

Implementation Variation

- In practice, there are a wide variety of applicable data structures and algorithms.
- Such variety indeed that we cannot expect a single module to take care of all possibilities; it would be enormous.
- We will need a family of modules to cover all the different implementations.

(难以做到单一模块满足所有需求，需要一组可复用模块)

Routine Grouping

- 
- **A self-sufficient reusable module would need to include a set of routines, one for each of the operations.**
 - **Completeness**
 - **Package**

Representation Independence

- A general form of reusable module should enable clients to specify an operation without knowing how it is implemented.
- Representation Independence as an extension of the rule of Information Hiding, essential for smooth development of large systems: **implementation decisions will often change, and clients should be protected.** 实现会发生变化, 应保护client不受影响)
- Representation Independence reflects the client's view of reusability
 - **the ability to ignore internal implementation details and variants**

Factoring Out Common Behaviors

- Factoring Out(提取) Common Behaviors, **reflects the view of the supplier and, more generally, the view of developers of reusable classes.** Their goal will be to take advantage of any commonality that may exist within a family or sub-family of implementations.
- The variety of implementations available in certain problem areas will usually demand, as noted, a solution based on a family of modules. Often the family is so large that it is natural to look for sub-families.
- Each of these categories covers many variants, but it is usually possible to find significant commonality between these variants.

(可复用类的开发者角度：从各种大量的变化中提取出通用的共性的行为加以复用)



Summary





The end

April 3, 2018