哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 5: Reusability-Oriented Software Construction Approaches

5.1 Metrics, Morphology and External Observations of Reusability

Ming Liu

March 27, 2019

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

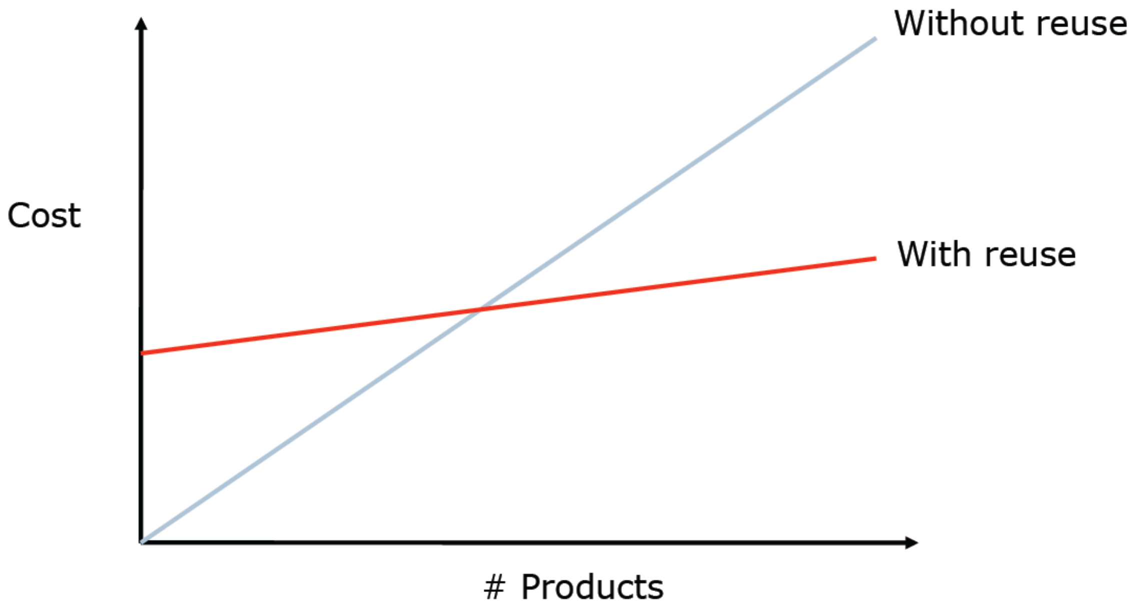
- **Software reuse is the process of implementing or updating software systems using existing software components.**
- **Two perspectives of software reuse**
  - Creation: creating reusable resources in a systematic way (programming for reuse)
  - Use: reusing resources as building blocks for creating new systems (programming with reuse)
- **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).

Why reuse?

- **Reuse is cost-effective and with timeliness**
  - 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**
  - Reusing functionality that has been around for a while and is debugged is a foundation for building on stable subsystems
- **Reuse yields standardization**
  - 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.**



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)
  - Portable and Standard Compliance
  - Adaptable and Flexible
  - Extensibility
  - Generic and Parameterization
  - Modularity
  - Localization of volatile (changeable) design assumptions
  - Stability under changing requirements

3 Levels and morphology of reusable components

Levels of Reuse

- **A reusable component may be code**
  - Most prevalent: 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

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

(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 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**
- **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

- ***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. here, the `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

대표

- **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`.

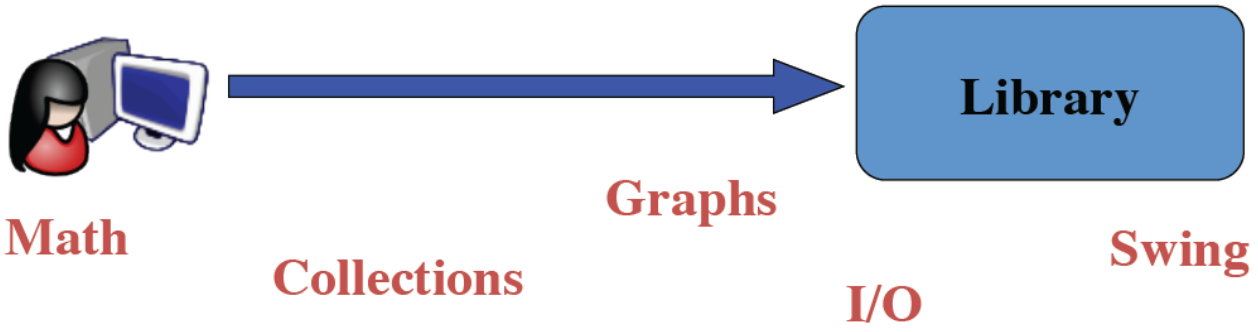
```
public class LoggingList<E> implements List<E> {
  private final List<E> list;
  public LoggingList<E>(List<E> list) { this.list = list; }
  public boolean add(E e) {
      System.out.println("Adding " + e);
      return list.add(e);
  }
  public E remove(int index) {
      System.out.println("Removing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

application program interface

(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 sub-system design containing a 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

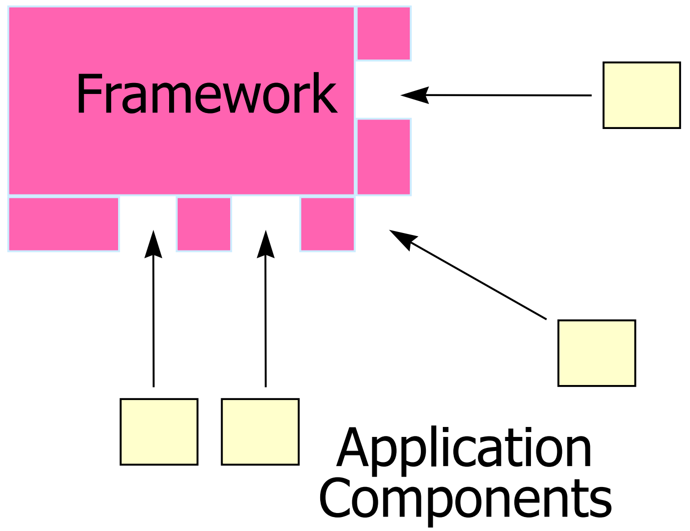
- **A framework is a reusable partial application that can be specialized to produce custom applications.**
  - Frameworks are targeted to particular technologies, such as data processing or cellular communications, or to application domains, such as user interfaces or real-time avionics.
- **The key benefits of frameworks are reusability and extensibility.**
  - Reusability 영향력 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 분리시키다 decouple the interfaces and behaviors of an application domain from the variations required by an application in a particular context.

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
  - adding methods that will be called in response to events recognized by the framework
- **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.**



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

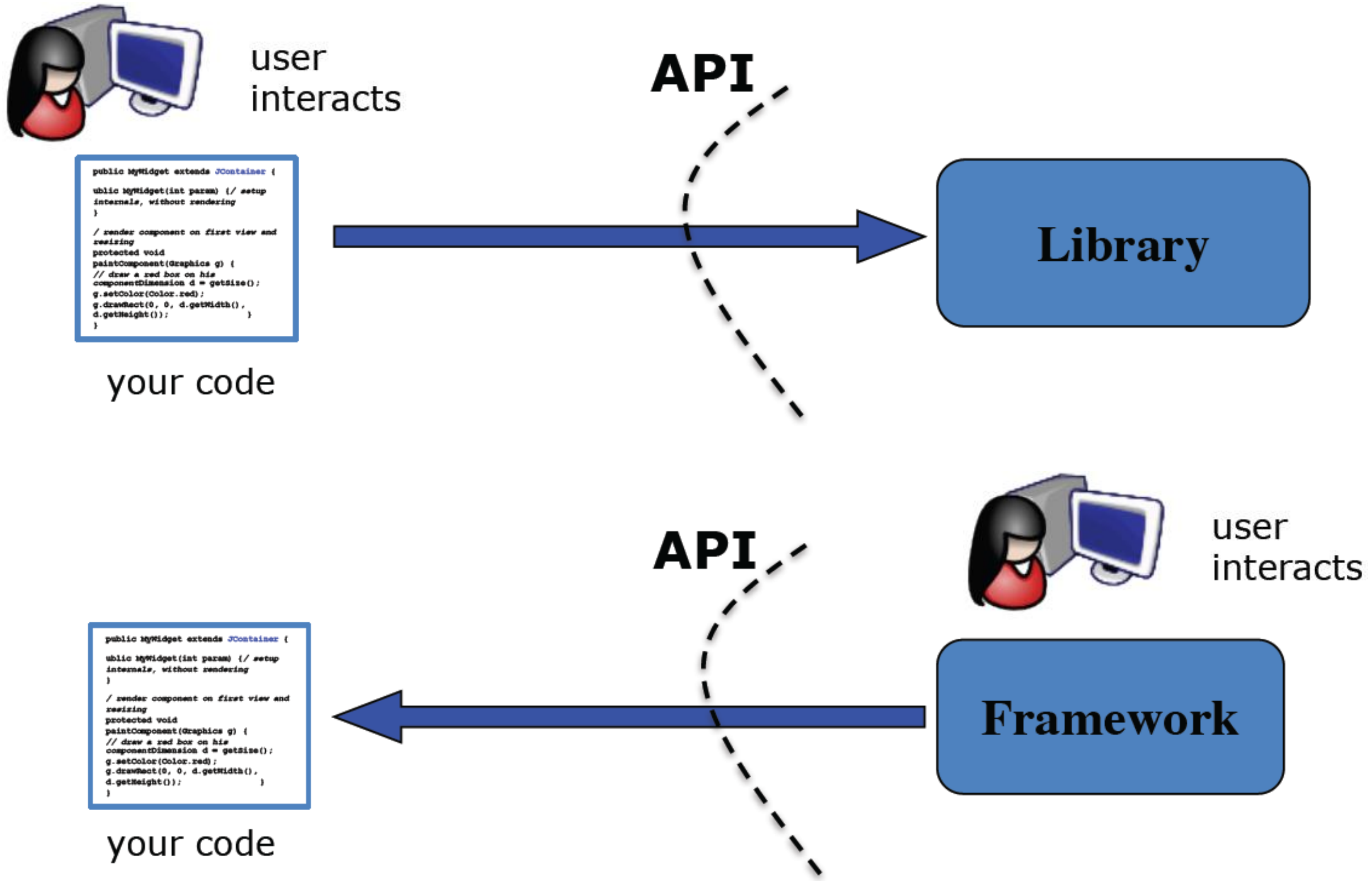
White-box and Black-Box Frameworks

- **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.
- **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.

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….)

General distinction: Library vs. framework



```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                    }
}
```

```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                    }
}
```

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.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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 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.**

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.

- Representation Independence reflects the client's view of reusability – the ability to ignore internal implementation details and variants

Factoring Out Common Behaviors

- Factoring Out 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.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

March 27, 2019