Cornell University
Computing and Information Science

CS 5150 Software Engineering

Reuse and Legacy Systems

William Y. Arms

Economics

Software is expensive. Therefore most software development makes extensive use of existing software. Avoid building new software if it duplicates what already exists.

This has several aspects:

- Program and system design that makes use of existing components.
- Design that facilitates replacing or updating components in the future.
- Working with legacy code.

Software Reuse

It is often good to design a program to reuse existing components. This can lead to better software at lower cost.

Potential benefits of reuse

- Reduced development time and cost
- Improved reliability of mature components
- Shared maintenance cost

Potential disadvantages of reuse

- Difficulty in finding appropriate components
- Components may be a poor fit for application
- Quality control and security may be unknown

Software Reuse Examples

Substantial parts of all systems use software supplied with the operating system (e.g., Microsoft), from an established software vendor (e.g., Oracle), or from a mature open source (Apache).

System software

- device drivers
- file systems
- exception handling
- network protocols

Subsystems

- database management systems
- firewalls
- web servers

Software Reuse Examples

Standard functions

- mathematical methods
- format conversion

User interface and application development

- toolkits (e.g. Motif graphics toolkit)
- class libraries, (e.g., Swing for Java)
- web frameworks (e.g., Ruby on Rails)

Only under very special circumstances, is it economic to write new software for the purposes listed on these two slides.

But it is often difficult to evaluate alternatives.

Evaluating Software

Software from well established developers is likely to be well written and tested, but still will have bugs and security weaknesses, especially when incorporated in unusual applications.

The software is likely to be much better than a new development team would write.

But sometimes it is sensible to write code for a narrowly defined purpose rather than use general purpose software.

Maintenance

When evaluating software, both commercial and open source, pay attention to maintenance. Is the software supported by an organization that will continue maintenance over the long term?

Reuse: Open Source Software

Open source software varies enormously in quality.

- Because of the processes for reporting and fixing problems, major systems such as Linux, Apache, Python, Lucene, etc. tend to be very robust and free from problems. They are often better than the commercial equivalents.
- More experimental systems, such as Hadoop, have solid cores, but their lesser used features have not been subject to the rigorous quality control of the the best software products.
- Other open source software is of poor quality and should not be incorporated in production systems.

Reuse: Application Packages

Application package

- Supports a standard application (e.g., payroll)

Functionality can be enhanced by:

- Configuration parameters (e.g., table driven)
- Extensibility at defined interfaces
- Custom written source code

Evaluating Applications Packages

Applications packages for business functions are provided by companies such as SAP and Oracle. They provide **enormous capabilities** and relieve an organization from such tasks as updating financial systems when laws change.

They are very expensive:

- License fees to the vendor.
- Modifications to existing systems and special code from the vendor.
- Disruption to the organization when installing them.
- Long term maintenance costs.
- The costs of changing to a different vendor are huge.

Cornell's decision (about 1990) to move to PeopleSoft (now part of Oracle) has cost the university several hundred millions of dollars.

If you are involved in such a decision insist on a **very thorough feasibility study**. Be prepared to take a least a year and spend several million dollars before making the decision.

Design for Change: Replacement of Components

The software design should anticipate possible changes in the system over its life-cycle.

New vendor or new technology

Components are replaced because its supplier goes out of business, ceases to provide adequate support, increases its price, etc., or because better software from another sources provides better functionality, support, pricing, etc.

This can apply to either open source or vendor-supplied components.

Design for Change: Replacement of Components

New implementation

The original implementation may be problematic, e.g., poor performance, inadequate back-up and recovery, difficult to trouble-shoot, or unable to support growth and new features added to the system.

Example. The portal nsdl.org was originally implemented using uPortal. This did not support important extensions that were requested and proved awkward to maintain. It was reimplemented using PHP/MySQL.

Design for Change: Replacement of Components

Additions to the requirements

When a system goes into production, it is usual to reveal both weaknesses and opportunities for extra functionality and enhancement to the user interface design.

For example, in a data-intensive system it is almost certain that there will be requests for extra reports and ways of analyzing the data.

Requests for enhancements are often the sign of a successful system. Clients recognize latent possibilities.

Design for Change: Replacement of Components

Changes in the application domain

Most application domains change continually, e.g., because of business opportunities, external changes (such as new laws), mergers and take-overs, new groups of users, new technology, etc., etc.,

It is rarely feasible to implement a completely new system when the application domain changes. Therefore existing systems must be modified. This may involve extensive restructuring, but it is important to reuse existing code as much as possible.

Design for Reuse: Class Hierarchies

Example: Java

Java is a relatively straightforward language with a very rich set of class hierarchies.

- Java programs derive much of their functionality from standard classes.
- Learning and understanding the classes is difficult.
- Experienced Java programmers can write complex systems quickly.
- Inexperienced Java programmers write inelegant and buggy programs.

Languages such as Java and Python steadily change their class hierarchies over time. Commonly the changes replace special purpose functionality with more general frameworks.

If you design your programs to use the class hierarchies in the style intended by the language developers, it is likely to help with long term maintenance.

Design for Reuse: Inheritance and Abstract Classes

Inheritance and abstract classes

Many of the standard design patterns anticipate future changes in a system, either by replacing components or by adding new ones. These patterns make extensive use of inheritance and abstract classes, and of delegation.

Classes can be defined in terms of other classes using inheritance. The generalization class is called the superclass and the specialization is called the subclass.

If the inheritance relationship serves only to model shared attributes and operations, i.e., the generalization is not intended to be implemented, the class is called an abstract class.

For a fuller discussion of design for reuse see the book by Bruegge and Dutoit in the readings.

Design for Reuse: Specification Inheritance

Specification inheritance

Specification Inheritance is the classification of concepts into type hierarchies, so that an object from a specified class can be replaced by an object from one of its subclasses.

In particular:

- Pre-conditions cannot be strengthened in a subclass.
- Post-conditions cannot be weakened in a subclass.

Design for Reuse: Specification Inheritance

Liskov Substitution Principle (strict inheritance)

If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

Interpretation

The Liskov Substitution Principle means that if all classes are subtypes of their superclasses, all inheritance relationships are specification inheritance relationships. New subclasses of T can be added without modifying the methods of T. This leads to an extensible system.

Design for Reuse: Delegation

Delegation

A class is said to delegate to another class if it implements an operation by resending a message to another class.

Delegation is an alternative to inheritance that can be used when reuse is anticipated.

Legacy Systems

The Worst Case

A large, complex system that was developed several decades ago:

- Widely used either within a big organization or by an unknown number of customers.
- All the developers have retired or left.
- No list of requirements. It is uncertain what functionality the system provides and who uses which functions.
- System and program documentation incomplete and not kept up to date.
- Written in out-of-date versions of programming languages using system software that is also out of date.
- Numerous patches over the years that have ignored the original system architecture and program design.
- Extensive code duplication and redundancy.
- The source code libraries and production binaries may be incompatible.

Legacy Requirements

Planning

In conjunction with the client develop a plan for rebuilding the system.

Requirements as seen by the customers and users

- Who are the users?
- What do they actually use the system for?
- Does the system have undocumented features that are important or bugs that users rely on?
- How many people use the fringe parts of the system? Where are they flexible?

Requirements as implied by the system design

- If there is any system documentation, what does it say about the requirements?
- Does the source code include any hints about the requirements?
- Is there code to support obsolete hardware or services? If so, does anybody still use them?

Legacy Code

Source code management

- Use a source code management system to establish a starting version of the source code and binaries that are built from this source code.
- Create a test environment so that the rebuilt system can be compared with the current system. Begin to collect test cases.
- Check the licenses for all vendor software.

Legacy Code

Rebuilding the software

The following tasks may be tackled in any appropriate order, based on the condition of the code. Usually the strategy will be to work on different parts of the system in a series of phases.

An incremental software development process is often appropriate, with each increment released when completed.

- Understand the original systems architecture and program design.
- Establish a component architecture, with defined interfaces, even if much of the code violates the architecture and needs adapters.
- Move to current versions of programming languages and systems software.
- If there are any subsystems that do not have source code, carry out a development cycle to create new code that implements the requirements.
- If there is duplicate code, replace with a single version.
- Remove redundant code and obsolete requirements.
- Clean up as you go along.