***Multidisciplinary Projects***

***Class 4b: Implementation***

Contents

[**Agenda** 2](#_Toc140157009)

[**Learning Objectives** 2](#_Toc140157010)

[**Program Design** 3](#_Toc140157011)

[**UML Models** 3](#_Toc140157012)

[**Class Diagram** 4](#_Toc140157013)

[**The "Hello, World!" Applet** 4](#_Toc140157014)

[**The HelloWorld Class** 5](#_Toc140157015)

[**Notation: Relationships** 5](#_Toc140157016)

[**The HelloWorld Class** 5](#_Toc140157017)

[**Notation: Association** 6](#_Toc140157018)

[**Association** 6](#_Toc140157019)

[**Deciding which Classes to Use** 6](#_Toc140157020)

[**Application Classes and Solution Classes** 7](#_Toc140157021)

[**Noun Identification: a Library Example** 7](#_Toc140157022)

[**Candidate Classes** 8](#_Toc140157023)

[**Relations between Classes** 8](#_Toc140157024)

[**Methods** 8](#_Toc140157025)

[**A Possible Class Diagram** 9](#_Toc140157026)

[**From Candidate Classes to Completed Design** 9](#_Toc140157027)

[**Notation: Interaction** 10](#_Toc140157028)

[**Sequence Diagram: Borrow Copy of a Book** 10](#_Toc140157029)

[**Sequence Diagram: Change in Cornell Program** 11](#_Toc140157030)

[**Sequence Diagram: Painting Mechanism** 11](#_Toc140157031)

[**Integrated Development Environments** 11](#_Toc140157032)

[**Program Design: Integrated Development Environment** 13](#_Toc140157033)

[**The Design — Code — Redesign Loop** 13](#_Toc140157034)

[**Class Hierarchies** 14](#_Toc140157035)

[**Web Development Frameworks** 15](#_Toc140157036)

[**Web Development Frameworks: Django** 15](#_Toc140157037)

[**Advanced Development Environments** 15](#_Toc140157038)

[**Using Development Frameworks** 16](#_Toc140157039)

[**Production Programming** 17](#_Toc140157040)

[**Software Reuse** 18](#_Toc140157041)

[**Evaluating Software** 18](#_Toc140157042)

[**Reuse: Open Source Software** 19](#_Toc140157043)

[**Evaluating Applications Packages** 19](#_Toc140157044)

[**Design for Change: Replacement of Components** 20](#_Toc140157045)

[**Design Patterns** 21](#_Toc140157046)

[**Structural** 21](#_Toc140157047)

[**Legacy Systems** 22](#_Toc140157048)

[**Legacy Requirements** 22](#_Toc140157049)

[**Legacy Code** 23](#_Toc140157050)

[**Testing: System and Sub-System Testing** 24](#_Toc140157051)

# **Agenda**

* Models for program design
* Program development
* Reuse and design patterns

# **Learning Objectives**

* Understanding the program design
* Understanding the program development
* (Understanding the design patterns)

# **Program Design**

* The task of program design is to represent the software architecture in a form that can be implemented as one or more executable programs. Given a system architecture, the program design specifies:
  + programs, components, packages, classes, class hierarchies, etc.
  + interfaces, protocols (where not part of the system architecture)
  + algorithms, data structures, security mechanisms, operational procedures
* If the program design is done properly, all significant design decisions should be made before implementation. Implementation should focus on the actual coding.

# **UML Models**

* UML models (diagrams and specifications) can be used for almost all aspects of program design.
  + Diagrams give a general overview of the design, showing the principal elements and how they relate to each other.
  + Specifications provides details about each element of the design. The specification should have sufficient detail that they can be used to write code from.
* In heavyweight software development processes, the entire specification is completed before coding begins.
* In lightweight software development processes, an outline specification is made before coding, but the details are completed as part of the coding process, using language based tools such as Javadocs.
* Models used mainly for requirements
  + Use case diagram shows a set of use cases and actors, and their relationships.
* Models used mainly for systems architecture
  + Component diagram shows the organization and dependencies among a set of components.
  + Deployment diagram shows the configuration of processing nodes and the components that live on them.
* Models used mainly for program design
  + Class diagram shows a set of classes, interfaces, and collaborations with their relationships.
  + Object diagram or sequence diagram show a set of objects and their relationships.

# **Class Diagram**

* A class is a description of a set of objects that share the same attributes, methods, relationships, and semantics.

A diagram of a diagram

Description automatically generated

# **The "Hello, World!" Applet**

import java.awt.Graphics;

class HelloWorld extends java.applet.Applet {

public void paint (Graphics g) {

g.drawString ("Hello, World!", 10, 20);

}

}

# **The HelloWorld Class**

A diagram of a diagram

Description automatically generated

# **Notation: Relationships**

* A dependency is a semantic relationship between two things in which a change to one may effect the semantics of the other.



* A generalization is a relationship is which objects of the specialized element (child) are substitutable for objects of the generalized element (parent).



* A realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.



# **The HelloWorld Class**

* Note that the Applet and Graphics classes are shown briefly, i.e., just the name is shown, not the attributes or operations.

A diagram of a computer

Description automatically generated

# **Notation: Association**

* An association is a structural relationship that describes a set of links, a link being a connection among objects.

A blue and white text

Description automatically generated

# **Association**

A diagram of a parking lot

Description automatically generated

# **Deciding which Classes to Use**

* Given a real-life system, how do you decide what classes to use?
* Step 1. Identify a set of candidate classes that represent the system design.
  + What terms do the users and implementers use to describe the system? These terms are candidates for classes.
  + Is each candidate class crisply defined?
  + For each class, what is its set of responsibilities? Are the responsibilities evenly balanced among the classes?
  + What attributes and methods does each class need to carry out its responsibilities?
* Step 2. Modify the set of classes
* Goals:
  + Improve the clarity of the design
    - If the purpose of each class is clear, with easily understood methods and relationships, developers are likely to write simple code, which future maintainers can understand and modify.
  + Increase coherence within classes, and lower coupling between classes.
  + Aim for high cohesion within classes and weak coupling between them.

# **Application Classes and Solution Classes**

* A good design is often a combination of application classes and solution classes.
* Application classes represent application concepts. Noun identification is an effective technique to generate candidate application classes.
* Solution classes represent system concepts, e.g., user interface objects, databases, etc.

# **Noun Identification: a Library Example**

* The library contains books and journals. It may have several copies of a given book. Some of the books are reserved for short-term loans only.
* All others may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.
* The system must keep track of when books and journals are borrowed and returned, and enforce the rules.

# **Candidate Classes**

A screenshot of a computer

Description automatically generated

# **Relations between Classes**

A white rectangular object with black text

Description automatically generated

# **Methods**

A white rectangular object with black text

Description automatically generated

# **A Possible Class Diagram**

A diagram of a company

Description automatically generated

# **From Candidate Classes to Completed Design**

* Methods used to move to final design
* Reuse: Wherever possible use existing components, or class libraries. They may need extensions.
* Restructuring: Change the design to improve understandability, maintainability, etc. Techniques include merging similar classes, splitting complex classes, etc.
* Optimization: Ensure that the system meets anticipated performance requirements, e.g., by changed algorithms or restructuring.
* Completion: Fill all gaps, specify interfaces, etc.
* Design is iterative
  + As the process moves from preliminary design to specification, implementation, and testing it is common to find weaknesses in the program design. Be prepared to make major modifications.

UML Notation for Classes and Objects

A screenshot of a computer

Description automatically generated

Modeling Dynamic Aspects of Systems

* Interaction diagram: shows set of objects and their relationships including messages that may be dispatched among them.
  + Sequence diagrams: Time ordering of messages

# **Notation: Interaction**

* An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.

A close-up of a blue line

Description automatically generated

# **Sequence Diagram: Borrow Copy of a Book**

A diagram of a diagram

Description automatically generated

# **Sequence Diagram: Change in Cornell Program** A diagram of a programming language Description automatically generated

# **Sequence Diagram: Painting Mechanism** A diagram of a software application Description automatically generated

# **Integrated Development Environments**

* Basic software development requires:
  + text editor (e.g., vi editor for Linux)
  + compiler for individual files
  + build system (e.g., make for Linux)
* Integrated development environments combine:
  + source code editor
  + incremental compiler
  + build automation tools
  + a debugger

A screenshot of a computer

Description automatically generated

* Eclipse is a modern integrated development environment. It was originally created by IBM’s Rational division. There are versions for many languages including Java, C/C++, Python, etc. The Java system provides:
  + source code editor
  + debugger
  + incremental compiler
  + programming documentation
  + build automation tools
  + version control
  + XML editor and tools
  + web development tools Much more is available via plug-ins

# **Program Design: Integrated Development Environment**

* Integrated development environments provide little help in designing a program.
* They assume that you have already have a design:
  + classes
  + methods
  + data structures
  + interfaces
* Options for program design:
  + program design using modeling tools, such as UML
  + design while coding: design — code — redesign loop (small programs only)
  + existing frameworks
  + advanced environments that combine frameworks and development tools
* It is often good to combine aspects of these different approaches

# **The Design — Code — Redesign Loop**

* If the class structure is straightforward it may be possible to use the integrated development environment to:
  + create an outline of the class structure and interfaces
  + write code
  + modify the class structure as needed and rework the code as necessary
* This is only possible with small teams with close communication.
* The maximum size of program depends on experience of programmer(s) and complexity of the program. It may be possible to complete a single agile sprint.
* Eventually the amount of rework becomes overwhelming.

# **Class Hierarchies**

* Since the design of class hierarchies is difficult it is good practice to use existing frameworks.
* Often many of the classes will have been written for you, or abstract classes are provided that you can use as a basis for your own subclasses.
* Examples:
  + class hierarchies that are part of programming languages
  + toolkits (e.g., for graphical user interfaces)
  + design patterns
  + frameworks for web development and mobile apps
* 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.

# **Web Development Frameworks**

* A web development framework provides a skeleton for building web applications.
* An early example was Cold Fusion, which implements a three tier architecture. Modern example, such as Ruby on Rails and Django, often use a MVC architecture. For example, Ruby on Rails provides:
  + a database
  + a web server
  + web pages
* It is intended to be used with web standards, e.g., XML , HTML, CSS, and JavaScript.

# **Web Development Frameworks: Django**

* Django is a Python framework for developing web sites
  + loosely based on MVC architecture
  + supports a variety of web and database servers
  + web template system
  + authentication system
  + administrative interface
  + mitigation of web attacks
* Django is a complex framework. Teams should allow plenty of time for learning

# **Advanced Development Environments**

* Application frameworks can be used with any program development environment, e.g., Django and Eclipse (Python version)
* An advanced development environment combines:
  + integrated development environment (IDE)
  + application framework
  + user interface layout manager and more
* Example: Apple’s Xcode for iOS

A screenshot of a computer

Description automatically generated

* An advanced development environment is intended to provide everything that a developer needs.
* The developer is expected to follow the program choices that are provided.
* For example, when Xcode is used with iOS it has a very specific purpose: mobile apps for Apple devices such as iPhones, iPads.
  + Special programming language (Swift or Objective C)
  + MVC framework (Apple version)
* If you accept the overall program design it is very powerful:
  + Auto layout of graphical interfaces
  + Comprehensive set of classes for user interfaces and navigation
  + Simulators for all Apple devices

# **Using Development Frameworks**

* Development frameworks are powerful and flexible.
* If your application fits the framework, they do much of the program design and provide high quality code for many of the standard parts of any application.
* Some parts of the application may need be designed separately.
* But beware:
  + You are forced to build your application within the framework that is provided.
  + The frameworks are continually modified.
  + These frameworks are complex and take a long time to learn.

# **Production Programming**

* Murphy's Law:
  + If anything can go wrong, it will.
* Challenges:
  + Code has to be maintained over the long term, with different system software.
  + Interfaces will be used in new and unexpected ways.
  + Every possible error will eventually occur at the worst possible time (bad data, failures of hardware and system software).
  + There are likely to be security attacks.
* Robust programming
  + Write simple code.
  + Avoid risky programming constructs.
  + If code is difficult to read, rewrite it.
  + Incorporate redundant code to check system state after modifications.
  + Test implicit assumptions explicitly, e.g., check all parameters received from other routines.
  + Eliminate all warnings from source code.
  + Have a thorough set of test cases for all your code.
* In a production environment, expect to spend longer on coding and testing than in an academic setting.

# **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

# **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 best software products.
* Other open source software is of poor quality and should not be incorporated in production systems.

# **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 software  from  another source provides better functionality, support,  pricing, etc.
* This can apply to either open source or vendor-supplied 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
* 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.
* 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 Patterns**

* Design patterns are template designs that can be used  in a variety of systems. They are particularly appropriate in situations where classes are likely to be reused in a system that evolves over time.

# **Structural**

* These concern class and object composition. They use  inheritance to compose interfaces and define ways to  compose objects to obtain new functionality.
* Adapter allows classes with incompatible interfaces to  work together by wrapping its own interface around  that of an already existing class.
* Bridge decouples an abstraction from its  implementation so that the two can vary independently.
* Composite composes zero-or-more similar objects so  that they can be manipulated as one object.
* Decorator dynamically adds/overrides behavior in an existing method of an object.
* Facade provides a simplified interface to a large body  of code.
* Flyweight reduces the cost of creating and manipulating a large number of similar objects.
* Proxy provides a placeholder for another object to  control access, reduce cost, and reduce complexity.

# **Legacy Systems**

* Many data intensive systems, e.g., those used by banks,  universities, etc. are legacy systems. They may have been  developed forty years ago as  batch processing, master file update systems and been continually modified.
  + Recent modifications might include customer interfaces for  the web, smartphones, etc.
  + The systems will have migrated from computer to computer,  across operating systems, to different database systems, etc.
  + The organizations may have changed through mergers, etc.
* Maintaining a coherent system architecture for such legacy  systems is an enormous challenge, yet the complexity of  building new systems is so great that it is rarely attempted.
* 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.
* Rebuilding the software
  + An incremental software development process is often  appropriate, with each increment released when completed.
  + 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.
    - 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.

# **Testing: System and Sub-System Testing**

* Tests on components or complete system, combining  units that have already been thoroughly tested
* Emphasis on integration and interfaces
* Trial data that is typical of the actual data, and/or  stresses the boundaries of the system, e.g., failures, restart
* Carried out systematically, adding components until  the entire system is assembled
* Can be open or closed box: by development team or by  special testers
* System testing is finished fastest if each component is  completely debugged before assembling the next