A beginners introduction to software development, OO development, and UML
Introduction

• Coding represents 15 to 20 percent of time to develop production software

• Strategies for writing small programs are not effective for large programs

• Virtually all real-world software is at least medium sized
  – Millions of lines of code
  – Dozens of programmers
  – Operating systems, database programs, e-commerce applications
The Challenges of Software Development

• Complexity
  – How do you manage the development of that much code?

• Longevity and Evolution
  – How do you manage the maintenance of that much code?
  – Lots of companies still working with and maintaining “legacy code”
Managing these challenges

• The creation of a methodology to formally define and organize the activities that are part of this “software development process”

• The term “software engineering” was first used in the late 60s (but is still a hotly contested term) No universal agreement on exact steps in software development
Managing these challenges

• Preparation required before writing code
  – Specify the problem
  – Design the overall structure of the solution
  – Select and analyze algorithms and data structures

• After work completed:
  – Testing
  – Documentation
  – Support
  – Maintenance
The Software Life Cycle

• No universal agreement on exact steps in software development

• A TYPICAL Software life cycle
  – Problem specification
  – Program design
  – Selection of algorithms and data structures
  – Coding and debugging
  – Testing and verification
  – Documentation and support
  – Maintenance
1. Problem Specifications

- In order to solve a problem correctly you have to solve the correct problem

- **Problem specification phase**
  - Complete, accurate, unambiguous statement of exact problem
  - List inputs provided to the program
  - For each input, list what outputs the program produces

- **Problem specification document** is an input/output document
1. Problem Specifications

[FIGURE 1-3] Basic structure of a problem specification document
1. Problem Specifications

• Developers must include description of program’s behavior for every possible input
  – Computer programs have no common sense
  – Describe software behavior for all inputs, even if unexpected

• Majority of specification document describes responses to unusual, unexpected, illegal inputs

• More realistic diagram of a problem specification
1. Problem Specifications

Challenges with the Problem Specifications

• Natural language is poor notation for accurate specifications
  – Language has multiple meanings, differences in interpretation
  – Having said that, it’s almost universally used
2. Software Design

• Follows completion of specification document
• Specify an integrated set of software components
• **Component** is a separately compiled program unit
  – Function, procedure, class, template, package, interface
  – Organizes and manages coding task
• **Divide-and-conquer** – large problem divided into smaller, simpler subproblems
2. Software Design

• In traditional software engineering, for each component, specify three items
  – Interface
  – Preconditions
  – Postconditions

• Software design document – component specifications and their relationships

• If all preconditions are true, the program unit guarantees postconditions will be true
[FIGURE 1-5] First-level software design

[FIGURE 1-6] Expanded software design
**CLASS** WaitingLine

**METHODS**
- `putAtEnd(c)`
  - Put a new customer `c` at the end of this line.
- `c = getFirstCustomer()`
  - Remove the first customer; return it in `c`.
- `isEmpty()`
  - True if this line is empty; false otherwise.
- `isFull()`
  - True if this line is full; false otherwise.

**CLASS** Teller

**METHODS**
- `isBusy()`
  - True if this teller is busy; false otherwise.
- `serve(c)`
  - This teller begins serving customer `c`.

**CLASS** Customer

**METHODS**
- `depart()`
  - This customer leaves the bank.

**CLASS** Transaction

**METHODS**
- `t = transactionType()`
  - Return the type of this transaction.
- `a = transactionAmount()`
  - Return the dollar amount of this transaction.

**Figure 1-7** Example of an object-oriented design
2. Software Design

• Formal specification languages more precise
  – More difficult to read and interpret
• UML (Unified Modeling Language) visually diagrams relationships and dependencies
  – We will consider this more in a little bit
3. Algorithms and Data Structures

• Select data structures and choose algorithms before coding
  – Efficiency influenced most by algorithms and data structures
  – Efficient algorithm remains efficient even if poorly coded
  – Inefficient algorithms cannot be made efficient by clever programming or machine speed
4. Coding and Debugging

• **Coding** is the software development phase covered in the first courses in computer science

• **Software reuse** – using software that already exists
  – Java 1.10 has 314 packages, 6003 classes

• Java contains features vital to modern software development
5. Testing and Verification

• **Program verification** formally proves the correctness of a program
  - Develop correctness proof as you write the code
  - Difficult and time consuming
  - Not widely used

• **Empirical testing** demonstrates correctness with carefully chosen test data

• **Unit testing** – test each program unit on carefully chosen data
5. Testing and Verification

- **Flow path** – test every execution sequence through a program unit
  - Virtually infinite number of flow paths
  - **Exhaustive testing** usually impossible
  - Verify every statement executed at least once

- **Integration testing** – test the correctness of routines working together
  - Usually proceeds from the bottom up
5. Testing and Verification

• **Clear box testing** – look inside the code to decide what test cases are needed
  – Also called *alpha testing*

• **Black box testing** – put the program in a realistic environment and run it
  – Also called *beta testing, acceptance testing*
  – Data related to actual user operations

• If beta tests show the program operates correctly, problem specification has been met
  – Program can be delivered to the user
6. Documentation and Support

- **User documentation** – complete user’s manual
  - May include online documentation

- **Technical documentation**
  - Problem specification document
  - Program design document
  - Description of algorithms and data structures
  - Program listing with inline comments
  - Description of testing and acceptance procedures
6. Documentation and Support

• **Javadoc comment** placed between /*** and */ in the program code
  - Comments contain tags that provide information about the program unit

• Four common javadoc tags
  - @author *name*
  - @version *text*
  - @param *name description*
  - @return *description*
7. Program Maintenance

• **Program maintenance** adapts the software to maintain correctness
  - Stays current with changing specifications and new equipment
  - Implies software life cycle repeated more than once

• Maintenance easier if program well designed from the start
Object-Oriented Design

• **Requirements phase** of software life cycle
  − User needs identified and documented
  − Requirements document describes behavior of proposed software from user’s perspective
  − Focus on what to do, not how to do it
Object-Oriented Design

• Design Phase
  – Define explicit behavior of program
  – Specification document identifies specific outputs for all possible inputs
  – Specific system requirements documented
  – Requirements and specification documents
  – Identifies and defines classes, behaviors of each class, and interactions
Object-Oriented Design (continued)

• Design phase is one of the most difficult
  − Design allows programmers to develop software that is efficient, easy to build and maintain
A beginners process

• Begin with analysis
  – What are the objects in my program?
  – In order to identify similar objects, what do they need to know (state)?
  – What actions should each object be able to take? (behavior)
  – What messages should each object be able to respond to? (behavior)
Finding Classes

• Primary goals of design phase
  – Identify classes
  – Determine relationships between classes

• Decisions require talent and experience

• Majority of classes identified in requirements and specification documents

• Nouns in requirements document provide list of potential classes
  – List must be refined
Object-Oriented Design

Several tools capture these features of a design:

• CRC cards
• UML class diagrams
• Interaction/Sequence diagrams
CRC Cards

CRC cards focus us on the *who* and the *what* and discourage the *how*.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class: Timed Book Morph</td>
<td>Collaborators:</td>
</tr>
<tr>
<td></td>
<td>Super class: BookMorph</td>
<td>Timing Morph</td>
</tr>
<tr>
<td></td>
<td>Subclass: PresentationMorph</td>
<td>Timing Morph</td>
</tr>
<tr>
<td>Responsibilities:</td>
<td>- Start Timing Morph</td>
<td>Timing Morph</td>
</tr>
<tr>
<td></td>
<td>- Inform Timing Morph of current slide and total slides</td>
<td>Timing Morph</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing Morph</td>
</tr>
</tbody>
</table>
CRC Cards

CRC cards focus us on the who and the what and discourage the how.

We design by decomposing responsibilities into small, cohesive, and well-defined sets.

- When the responsibilities won’t fit on a CRC card, the set probably isn’t small enough.
- When the responsibilities are not cohesive, we split off new objects.
- When the responsibilities are not well-defined, we consider whether some other object should have some of the responsibilities.
A great observation

• If someone asks you what your class does and you use the word "and", it probably does too much.
  - Shawn Sparks
  - A former student who now works at Principal
Unified Modeling Language (UML)

• Software design must be clearly, concisely, correctly described to programmers

• Unified Modeling Language (UML) – a graphical language
  – Visually expresses software system design
  – Represents a design in standard format

• UML provides five views, nine diagram types
  – View – diagrams that highlight a system aspect

• UML has no rigid rules for what is included
Class Diagrams

• Describe static structure of a system in terms of classes and their relationships
• Class drawn as a rectangle divided into three compartments
• Class names appear in bold text, centered at the top of the rectangle
• Compartments describing state and behavior are optional
  - Method type information also optional
  - Names of routines that take parameters must be followed by parentheses
### Class Diagrams

<table>
<thead>
<tr>
<th>ClassName</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
</tr>
<tr>
<td>attribute : DataType</td>
</tr>
<tr>
<td>attribute : DataType = default value</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation (arg1, arg2, ...) : ResultType</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Class Diagrams (continued)

**Figure 2-13** Clock class diagram

- **State**
  - seconds: int
  - minutes: int
  - hours: int

- **Name**

- **Behavior**
  - start()
  - adjustTime()
  - reset()
Class Diagrams (continued)

![Class Diagrams](image)

**Figure 2-14** UML model of a Clock class

(a) Clock

(b) Clock

- secs: int
- mins: int
- hours: int
- setTime()
- adjustTime()
- reset()

(c) Clock

- secs: int
- mins: int
- hours: int
- setTime(): void
- adjustTime(): void
- reset(): void
Class Diagrams (continued)

• Specify parameter and return types with colon notation
• Include only as much information as necessary
• Most class diagrams do not include accessors or mutators
  − Also omit void return values
• Relationship between classes defined
  − One instance invokes a method or accesses the state of an instance of another
Class Diagrams (continued)

• UML defines relationships that describe how classes are related in a class diagram

• Association
  – State of one class contains a reference to an instance of another class
  – Permanent structural relationship
  – Solid line drawn between two classes to represent association
  – Not always bidirectional

• Navigability – arrow indicates direction of a relationship
Class Diagrams (continued)

• **Multiplicity** – several instances of a class
  - Denoted with an asterisk and a number
  - Number indicates the number of class instances

• **Dependency** – change in one class affects another class that uses it
  - Inherently one-way relationship

• Association usually implemented as part of the class state

• Dependency usually implemented as local variable, parameter, return value
  - Called **automatic** variables
Class Diagrams (continued)

![Diagram of Gas Pedal and Engine classes with methods and attributes]

**Figure 2-16** Adding navigability information to an association

![Diagram of ATM and Bank classes with multiplicity information]

**Figure 2-17** Multiplicity information added to an association
Class Diagrams (continued)

• **Dependency**
  - Represented as a dashed line
  - Arrow points to independent element

• **Generalization**
  - Superclass is a generalization of its subclass
  - Denoted by a triangle connecting a subclass to its parent class

• **Associations and generalizations represented in a single diagram**
Class Diagrams (continued)

[FIGURE 2-24] Banking system
Class Diagrams

[Figure 2-26] Printing system
Sequence Diagrams

- **Sequence diagrams** describe how groups of objects dynamically collaborate
- Single sequence diagram describes single behavior
- Time flows from bottom to top
  - Vertical lines denote lifetime of an object
- Each line labeled with object name followed by name of class from which object instantiated
  - Colon separates object name from class name
  - Class name always given, object name optional
    - Class name preceded by colon
Sequence Diagrams (continued)

Sequence diagram for the printing system of Figure 2-26

[FIGURE 2-27]
Sequence Diagrams (continued)

• Illustrates sequencing of events
• Does not show algorithms
  – Shows order in which messages are sent
• Each class described in a separate class diagram
• If sequence diagram shows message passing, corresponding class diagram must agree
• Separate sequence diagram for each major behavior in system being modeled
  – Includes behaviors central to system understanding
Summary

• Object-oriented design – think in terms of encapsulated entities, external behaviors
• Internal behavior may change
  – Does not affect user’s program
• Encapsulation and inheritance important characteristics
• UML is an industrial standard to express system design