

# Software Development Methods

## Course Introduction

Yih-Kuen Tsay  
(with contributions by Bow-Yaw Wang)

Department of Information Management  
National Taiwan University

# Stages/Activities in Software Development

- 🌐 Requirements Solicitation/Analysis
- 🌐 Specification
- 🌐 Design
- 🌐 Validation (+ Verification)
- 🌐 Implementation
- 🌐 Verification (+ Validation)
  - ☀ testing
  - ☀ simulation
  - ☀ formal verification
- 🌐 Deployment and Maintenance
- 🌐 Others: code review, documentation, etc.

# About Software Development Process

- 🌐 The stages do not necessarily follow one another sequentially; jumping back to a previous stage is common.
- 🌐 Different parts of a software application may be in different stages.
- 🌐 The ultimate goal is to deliver quality software on schedule, not following a particular process.
- 🌐 Best practices give you helpful guidelines, but you should try to do whatever is best for the project.

- 🌐 What do people ask of a program/software?
  - ☀️ **Correct**, doing what it is supposed to do
  - ☀️ **Efficient**, performing its tasks efficiently
  - ☀️ **Friendly**, easy to use
  - ☀️ **Well-structured** and hence easy to maintain
  - ☀️ **Fast** and **cheap** to develop
  - ☀️ **Secure** as it should be
  - ☀️ Etc.

- 🌐 What do people ask of a program/software?
  - ☀️ **Correct**, doing what it is supposed to do
  - ☀️ **Efficient**, performing its tasks efficiently
  - ☀️ **Friendly**, easy to use
  - ☀️ **Well-structured** and hence easy to maintain
  - ☀️ **Fast** and **cheap** to develop
  - ☀️ **Secure** as it should be
  - ☀️ Etc.
- 🌐 These demands pose quite a challenge!

# Are You Up to That Challenge?

- 🌐 Many students (who would become practicing programmers)
  - ☀ rarely care about writing “good” programs,
  - ☀ know few useful programming techniques, and
  - ☀ cannot use development tools effectively.

Note: in this course, a **good** program is one that is at least **correct** and **well-structured**.

# Are You Up to That Challenge?

- 🌐 Many students (who would become practicing programmers)
  - ☀️ rarely care about writing “good” programs,
  - ☀️ know few useful programming techniques, and
  - ☀️ cannot use development tools effectively.

Note: in this course, a **good** program is one that is at least **correct** and **well-structured**.

- 🌐 Consequence: low quality software!

# Are You Up to That Challenge?

- 🌐 Many students (who would become practicing programmers)
  - ☀ rarely care about writing “good” programs,
  - ☀ know few useful programming techniques, and
  - ☀ cannot use development tools effectively.

Note: in this course, a **good** program is one that is at least **correct** and **well-structured**.

- 🌐 Consequence: low quality software!
- 🌐 Shouldn't you start to get serious?



# Course Objectives

- 🌐 Learn how to develop correct and high-quality software with better engineering skills:
  - ☀️ **Software modeling** (with the UML in particular)
  - ☀️ **Domain modeling**
  - ☀️ **Design patterns**
  - ☀️ **Development/productivity tools**
  - ☀️ **Verification/analysis tools**
- 🌐 Practice these skills and **team work** with a substantial term project that reflects real-world situations.
- 🌐 Also, get exposed to a bit of formality so that you will be able to describe and reason about programs more precisely.

Note: there are numerous other software development methods. You are encouraged to explore them through course taking or self-study.

# Programming in Class

- 🌐 Environment is controlled.
- 🌐 Problems are well-defined (sorting, BFS, etc.).
- 🌐 Solutions are well-defined (in your algorithm textbooks).
- 🌐 Programs seldom change (write once, use once).
- 🌐 Correctness may not be an issue.
- 🌐 Robustness has rarely been an issue.

- 🌐 Environment is open.
- 🌐 Problems are *not* well-defined.
- 🌐 There may be multiple options available.
- 🌐 Programs change all the time.
- 🌐 Correctness is most important.
- 🌐 Robustness is necessary.

## Example: An Inventory System

A 24-hour store asks you to develop an inventory system:

- 🌐 The system will be used by many people.
- 🌐 It is impossible to know what goods or categories the store will have.
- 🌐 What database and user interface packages would you use?
- 🌐 What if they ask you to add new features?
- 🌐 Your system should better not be confused by different calendar systems (particularly in Taiwan).
- 🌐 Your system should better be able to be working all year long.

# About Software Project Management

- 🌐 Software development, after all, will be done by engineers.
- 🌐 Project leaders need to know what engineering options they have.
- 🌐 We will look at the software development problem from an engineer's point of view.
- 🌐 The course material should be complementary to related software project management courses.

- 🌐 After several meetings with your client, you have an informal idea of what your client wants.
- 🌐 You bring the informal idea back and start developing the system with your colleagues.
- 🌐 But your colleagues did not participate in the meetings. They are not as familiar with the domain knowledge as you are.
- 🌐 What would you do?

# Example: Sorting Template

- 🌐 Suppose you would like to develop a sorting algorithm for any totally ordered set.  
(Note: a set  $S$  is totally ordered if either  $a < b$ ,  $a = b$ , or  $a > b$  for any  $a, b \in S$ .)
- 🌐 How do you convey the idea to your colleague?

# A Probable Attempt

- 🌐 An element of a totally ordered set is an object of class `TOSet`.
- 🌐 We can create an object and assign its value.
- 🌐 The class `TOSet` has a static member function `compare(TOSet &, TOSet &)` that compares two elements.
- 🌐 The sorting function accepts an array of `TOSet` objects as inputs.
- 🌐 It uses `compare` to compare elements in the array.
- 🌐 It outputs a permutation of the input array such that the elements in the permutation are ordered by the `compare` function.



- 🌐 It is still ambiguous. (What do you mean by “ordered by the compare function?”)
- 🌐 It is incomplete. (What is a permutation?)
- 🌐 It is written in natural language.
- 🌐 It is already very complicated. (What if you have 30 classes in your system?)

# About the Unified Modeling Language

- 🌐 The UML is designed for software/program specification.
- 🌐 It is a graphical language.
- 🌐 It can be used to describe the relation among different classes.
- 🌐 It is convenient to illustrate the interactions among different objects.
- 🌐 It has a more rigorous semantics.
- 🌐 There are tools that can simulate your UML designs or convert them into code skeleton.
- 🌐 Etc.

# From Specification to Design

- 🌐 Software development is more than writing down the specification.
- 🌐 UML specification is a way of communication.
- 🌐 Like using natural languages, you may know the words and grammar of English, but you may not be able to compose a good essay in English.
- 🌐 After learning some basics of UML, we will discuss useful programming techniques for system design.

# It's Like Solving a Mathematical Problem

Compute

$$\int x^3 \ln^3 x dx = ?$$

# Solution

$$\begin{aligned}\int x^3 \ln^3 x dx &= \frac{x^4}{4} \ln^3 x - \int \frac{x^4}{4} \frac{3 \ln^2 x}{x} dx \\ &= \frac{x^4}{4} \ln^3 x - \frac{3}{4} \int x^3 \ln^2 x dx \\ &= \frac{x^4}{4} \ln^3 x - \frac{3}{4} \left( \frac{x^4}{4} \ln^2 x - \int \frac{x^4}{4} \frac{2 \ln x}{x} dx \right) \\ &= \frac{x^4}{4} \ln^3 x - \frac{3}{16} x^4 \ln^2 x + \frac{3}{8} \int x^3 \ln x dx \\ &= \frac{x^4}{4} \ln^3 x - \frac{3}{16} x^4 \ln^2 x + \frac{3}{8} \left( \frac{x^4}{4} \ln x - \int \frac{x^4}{4} \frac{1}{x} dx \right) \\ &= \frac{x^4}{4} \ln^3 x - \frac{3}{16} x^4 \ln^2 x + \frac{3}{32} x^4 \ln x - \frac{3}{32} \int x^3 dx \\ &= \frac{x^4}{4} \ln^3 x - \frac{3}{16} x^4 \ln^2 x + \frac{3}{32} x^4 \ln x - \frac{3}{128} x^4\end{aligned}$$

# Strategies and Patterns

- 🌐 What strategies do we have?
  - ☀ polynomial integration
  - ☀ integral of  $\ln x$
  - ☀ variable substitution
  - ☀ integration by parts
- 🌐 The problem is solved by choosing combinations of strategies.
- 🌐 What about program development?
- 🌐 Is there any strategy or pattern for programming?

Note: integration by parts

$$\int f(x)g'(x)dx = f(x)g(x) - \int f'(x)g(x)dx$$

- 🌐 Suppose you want to implement a database system.
- 🌐 The user may ask you to search or sort by field.
- 🌐 You may use sorting algorithms, search algorithms, even balanced tree data structures.
- 🌐 For different situations, you may use different sorting algorithms (e.g., memory versus disk-based).
- 🌐 You do not develop your program from scratch.

# What about System Architecture?

- 🌐 Suppose you want to develop a system for
  - ☀️ vehicle controller
  - ☀️ user interface
  - ☀️ data management
- 🌐 Is there any known strategy or pattern that could be applied?



## Example: Vehicle

- Let's suppose we want to define a vehicle rental system at seashore resorts.
- They have bikes, cars, sailboats, and yachts.
  - Class **LandVehicle** for bikes and cars
  - Class **WaterVehicle** for sailboats and yachts
- One day, a resort management team decides to introduce hovercrafts.
- How would you modify the class hierarchy to include the new product?

# Design Patterns

- 🌐 A design pattern is the re-usable form of a solution to a design problem.
- 🌐 For software, design patterns are formalized best practices that a programmer can use to solve common problems when designing an application.
- 🌐 Object-oriented design patterns typically show relationships and interactions between classes or objects.
  - ☀ structural
  - ☀ behavioral
- 🌐 They are frequently used in commercial tools and systems.







# Managing Changes

- 🌐 Design and implementation may change
- 🌐 How should the changes be managed?
  - ☀️ version control
  - ☀️ issues/bugs tracking



# From Design/Implementation to Validation/Verification

- 🌐 A software developed by proper methodologies does not necessarily entail quality.
- 🌐 UML specifications allow clients, system architects, and programmers to communicate.
- 🌐 Design patterns help system architects and programmers to deploy software structures sensibly.
- 🌐 But they do not imply the system cannot go wrong.

# Some Systems Are Critical

-  Device drivers
-  Medical instruments
-  Automotive control
-  Online banking
-  Stock exchange
-  Etc.

# What Are the Problems?

-  Design flaws
-  Programming errors

# A Lesson from the Hardware Industry

- 🌐 The first Pentium was found to have the infamous F00F bug.
- 🌐 IC manufacturing costs lots of money.
- 🌐 No company would want to have a buggy design to be sent to the foundry.
- 🌐 But how?

Note: the “Pentium floating point divide” bug (in 1993) ultimately cost Intel US\$ 475 million.

# Testing and Formal Verification

- 🌐 IC design houses use tools to help them catch bugs.
  - ☀️ Testing: run simulation on designs to find bugs
  - ☀️ Verification: analyze designs to prove they are correct
- 🌐 Software houses are increasingly using similar tools.






- 🌐 Testing is usually performed after the system is implemented.
- 🌐 Nonetheless, one can test the system design before it is implemented.
- 🌐 Simulator generates random inputs.
- 🌐 Erroneous behaviors can be observed if the proper inputs are generated.

# Formal Verification

- 🌐 It can check the system before it is implemented.
- 🌐 Verification tools try all possible inputs.
- 🌐 Erroneous behaviors can be observed if the proper inputs are generated.
- 🌐 Correctness can be ensured if all inputs have been tested.

# Ingredients of Formal Verification

-  Behavior Modeling
-  Property Specification
-  Verification Algorithm/Tool  
(or, if that fails, Proof and Proof Checker)

# Behavior Modeling

- 🌐 It describes system behaviors at a suitable abstraction level, hiding irrelevant details.
- 🌐 We need a formal language to avoid ambiguity.
- 🌐 The actual control flow of a program (at run time) is of main concern.
- 🌐 Users specify their systems as models in modeling languages.

# Property Specification

- 🌐 It specifies what properties are of interest.
- 🌐 Another formal language is needed.
- 🌐 High-level properties are independent of the implementation.
- 🌐 Users specify the requirements in property specification languages.

- 🌐 A verification tool takes the model and property specification as input.
- 🌐 It checks whether the model satisfies the property or not.
- 🌐 Many verification problems are undecidable and some work-around techniques (e.g., abstraction) may help.

# Correctness Proofs and Proof Checking

- 🌐 Correctness proofs are the last resort, when everything else fails.
- 🌐 Unfortunately, proofs are usually hard to produce.
- 🌐 Even worse, you can make mistakes in a proof.
- 🌐 Fortunately, checking if a proof is really a proof can be automated.

# Programming in the Small

- 🌐 We will also study development methods that probably only work for small programs.
- 🌐 However, a large program is usually composed of smaller ones.
- 🌐 A large program may also be a result of refinement from a smaller program.
- 🌐 Making the smaller programs correct helps improve the overall quality of the larger ones.



# Conclusion

- 🌐 This is a course that views software development from an engineer's viewpoint.
- 🌐 It covers design and programming techniques for software development.
- 🌐 It also introduces you to useful verification methods and tools.
- 🌐 We hope you will appreciate the methodologies and improve software quality with better engineering skills.