CSI1102: Introduction to Software Design

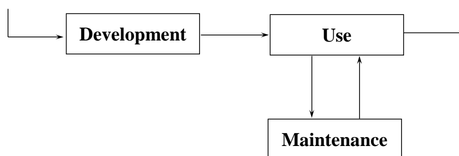
Chapter 10:
Introduction to Software Engineering

Learning objectives: Software Engineering

- The **quality** of the software is a direct result of the process we follow to create it
- Understand the need for and use of
  - software development models
  - the software life cycle and its implications
  - linear vs. iterative development approaches
  - goals and techniques of testing
  - an evolutionary approach to object-oriented development

The Software Life Cycle

- The overall *life cycle* of a program includes **use** and **maintenance:**

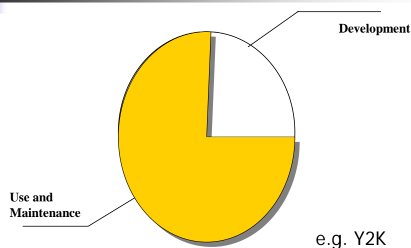


- A version of the software that is made available to user is called a *release*

About Maintenance

- *Maintenance* tasks include any modifications to an existing program;
  - enhancements and
  - defect removal
- Easy, careful design and velopment → easy to maintain
- Maintenance efforts tend to far outweigh the development effort in today's software
- **Software engineering GOAL**:
  - minimize the overall effort required to create and to maintain the program

Development vs. Maintenance



e.g. Y2K

Development and Maintenance Effort

Development	Maintenance

Development	Maintenance

Small increases in development effort can reduce maintenance effort →
Spend more time on design, documentation and implementation

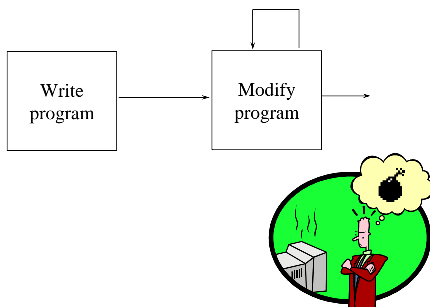
Development and Maintenance Effort

- Often the maintainers of a program are not the program's original developers (average 3 year "rule")
- Maintainers must be able to understand a program must be able to understand a program they didn't design
- The ability to read and understand a program depends on
  - how clearly the **requirements are established**
  - how well the program is **designed**
  - how well the program is **implemented**
  - how well the program is **documented**

Developing high quality software: Software Development Models

- A *software development model* is an organized approach to creating quality software
- Too many programmers follow a *build-and-fix* approach
  - They write a program and modify it until it is functional, without regard to system design
  - Errors are addressed haphazardly if and as they are discovered
  - **It is not really a development model at all**

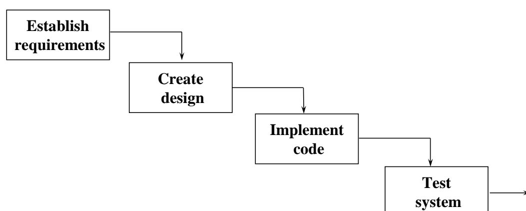
The Build-and-Fix Approach



The Waterfall Model

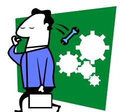
- The *waterfall model* was developed in the mid 1970s
- Activities that must be specifically addressed during development include:
  - Establishing clear and unambiguous requirements
  - Creating a clean design from the requirements
  - Implementing the design
  - Testing the implementation
- Originally it was proposed as a linear model, without backtracking

The Waterfall Model

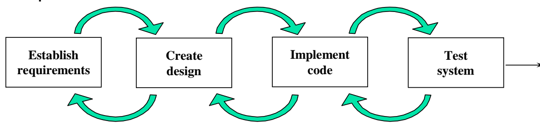


Iterative Development: Waterfall method with backtracking

- *Iterative development* allows the developer to cycle through the different development stages
- Backtracking should not be used irresponsibly
- When use backtracking?
  - To deal with unexpected problems arising only in later stages of development



An Iterative Development Process



13

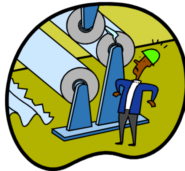
Important: Iterative Testing

- The results of each stage **should be evaluated carefully** prior to going on to the next stage
- Before moving on to the design, for example, the requirements should be evaluated to ensure
  - completeness, consistency, and clarity
- A design evaluation should ensure that each requirement was addressed adequately

14

Testing Techniques: Walkthrough

- A design or an implementation may be evaluated during a *walkthrough*
- The goal of a walkthrough is to **identify problems**, not to solve them



15

Testing Techniques: Create *test cases*

- Generally, the goal of testing is to find errors
- Often it is called *defect testing*
- A good test uncovers problems in a program
- A *test case* includes
  - a set of inputs
  - user actions or other initial conditions
  - expected output
- It is not feasible to test every possible case

16

Testing technique: Black-Box Testing

- *Black-box testing* maps a set of specific inputs to a set of expected outputs
- An *equivalence category* is a collection of input sets
  - E.g. positive integer category, 0..99
  - Test cases: -9, -500, 5, 12, 101, 300
- Two input sets belong to the same equivalence category if there is reason to believe that if one works, it will work for the other
  - Therefore testing one input set essentially tests the entire category

17

Testing technique: White-Box Testing

- *White-box testing* also is referred to as *glass-box testing*
  - It focuses on the internal logic such as the implementation of a method → we walk through the code
- *Statement coverage* guarantees that all statements in a method are executed
- *Condition coverage* guarantees that all paths through a method are executed

18

Prototypes: Use to sell your ideas

- A *prototype* is a program created to explore a particular concept
- Prototyping is more useful, time-effective, and cost-effective than merely acting on an assumption that later may backfire
- Usually a prototype is created to communicate to the client:
  - a particular task
  - the feasibility of a requirement
  - a user interface
- It is a way of **validating requirements**

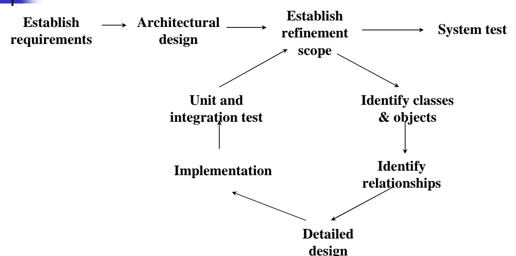
Throw-away vs. Evolutionary Prototypes

- A "quick and dirty" prototype to test an idea or a concept is called a *throw-away prototype*
  - Throw-away prototypes are not incorporated into final systems
- Because it is designed more carefully, an *evolutionary prototype* can be incorporated into the final system
  - Evolutionary prototypes provide a double benefit, but at a higher cost

Developing large software: Evolutionary Development Approach

- *Evolutionary development* divides the design process into
  - *architectural (high level) design*- primary classes and interaction
  - *detailed design* - specific classes, methods, and algorithms
- This allows us to create *refinement cycles*
- Each refinement cycle focuses on one aspect of the system
- As each refinement cycle is addressed, the system evolves

An Evolutionary Development Model



Refinement Cycle: #1 Establish the Scope

- First, we **establish the refinement scope** to define the specific nature of the next refinement
- For example:
  - the user interface
  - the feasibility of a particular requirement
  - utility classes for general program support
- Object-oriented programming is well suited to this approach
- Choosing the most appropriate next refinement is important and requires experience

Refinement Cycle: #2 Identify relevant classes/objects

- Identify **classes** and **objects** that relate to the current refinement
  - Look at the nouns in the requirements document
- Candidates categories include
  - physical objects (books, balls, etc.)
  - People (student, clerk, professor, etc.)
  - Places (room, airport, etc.)
  - Containers (bookcase, transaction list, etc.)
  - Occurrences (sale, meeting, accident, etc.)
  - Information stores (catalog, event log)
- Categories may overlap
- Consider reusing existing classes

Refinement Cycle: #3 Identify relationships

- Identify relationships among classes
  - general association ("uses")
  - aggregation ("has-a")
  - inheritance ("is-a")
- *Associated objects* use each other for the services they provide
- *Aggregation*, also called *composition*, permits one object to become part of another object
  - *Cardinality* describes the numeric relationship between the objects
  - For example, a car might have four wheels associated with it

Refinement Cycle: #3 (cont.) Inheritance

- *Inheritance*, discussed in detail in Chapter 7, may lead to the creation of a new "parent" abstract class whose sole purpose is to
  - gather common data and common methods in one place
- Use UML class diagrams to show the relationships

Refinement Cycle: #4-6 Detailed design, implement and test

- Finally, a refinement cycle includes detailed design, implementation, and testing
  - All the members of each class need to be defined
  - Each class must be implemented (coded)
  - *Stubs* sometimes are created to permit the refinement code to be tested
- A *unit test* focuses on one particular component, such as a method or a class
- An *integration test* focuses on the interaction between components

Specification of code

- Specification of design details
  - Invariant: collection of facts which are true
  - Precondition/Postcondition
    - Preconditions: Conditions which are required for code to execute correctly
    - Postconditions: Correct changes which result after code has been executed

Specification of code: An Example (AlarmClock class)

Invariant

An `Alarmclock` object
- Keeps track of a single alarm time in terms of time and minutes
- Cannot distinguish between AM and PM times
- Has attribute values restricted to the following ranges:
  - 1 <= hour <= 12 and 0 <= minutes <= 59

Update Methods

`public void advanceOneHour()`

precondition
hour < 12
modifies
hour
postcondition
The value of hour is one unit larger than before

Specification of code: An Example (AlarmClock class)

Update Methods (cont...)

`public void advanceTenMinutes()`

precondition
minute < 50
modifies
minute
postcondition
The value of minute is 10 units higher than before

- We can use formal logic to express the invariant, pre/post conditions
- We can then use formal logic to prove that a piece of code is "True".

Source: "The object of Jva, David D Riley, Addison-Wesley, 2002

Obtaining the requirements: The PaintBox project

- TASK (High level):
  - Create a program which allows the user to draw various shapes and sizes on the screen
- How will we go about accomplishing this?

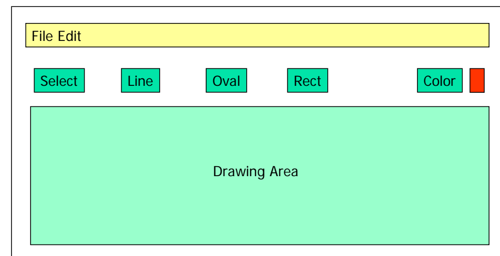
The PaintBox project: Requirements

- Create a mouse driven GUI
- Allow user to draw lines, circles, ovals, rectangles and squares
- Allow user to change drawing color
- Allow user to fill a shape, except a line, with a color.
- Allow user to being new drawing
- Allow user to create polylines

The PaintBox Project: Initial Refinement steps

- create the basic user interface
- allow the user to draw and fill shapes and to change color
- allow the user to select, to move, and to modify shapes
- allow the user to cut, copy, and paste shapes
- allow the user to save and to reload drawings
- allow the user to begin a new drawing at any time

The PaintBox Project: The Basic User Interface



The PaintBox Project

- Discussions with the client lead to additional requirements which are integrated into the requirements document
- Next the architectural design is prepared
- Refinement steps are determined
  - #1: the basic user interface
  - #2: drawing basic shapes using different stroke colors
  - #3: cutting, copying, and pasting shapes
  - #4: selecting, moving, and filling shapes
  - #5: modifying the dimensions of shapes
  - #6: saving and reloading drawings
  - #7: final touches such as the splash screen

Remaining PaintBox Refinements

- The full implementation can be downloaded for the Book's Website
- See 10.4 of the text book

Obtaining user requirements: The "toughest part"

- "Knowledge acquisition bottleneck":
  - Difficulty to extract information from humans
- Different personality types:
  - Levels of detail, concepts, thinking holistic, etc.
  - Myers Briggs (16 types), amongst others
    - Processing data and information: concrete versus abstract
    - Decision making: logical and objective versus value related and subjective
    - Introvert versus extravert: stimuli from outside or inside
    - Judgment: random versus "open-ended"
  - See WWW for tests and for sceptics!!!!

37

Summary: Chapter 10

- Chapter 10 has focused on:
  - software development models
  - the software life cycle and its implications
  - linear vs. iterative development approaches
  - goals and techniques of testing
  - an evolutionary approach to object-oriented development

38