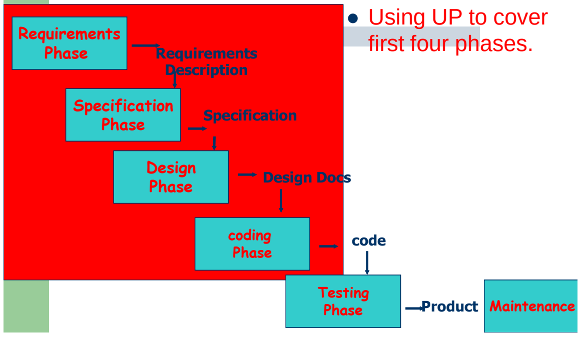
面向对象程序的分析与设计
Object-Oriented Analysis and Design

Lecture 12

Prof. S. Xu

Where We Are?

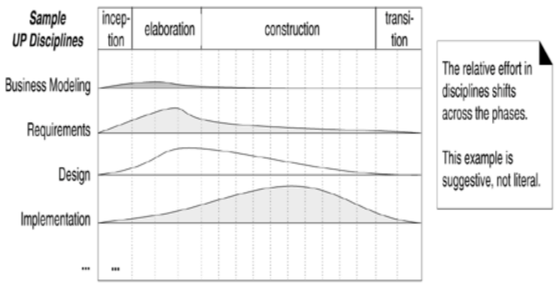


- Using UP to cover first four phases.

Contents

- Implementation
- Choice of language
- General Principles
- Coding Standards
  - Documentation
  - Name Convention

UP Disciplines and phases



4

Implementation

- Any issues with implementation?

Choice of Language

- Language may be specified in contract
- What if we have to choose it? Base on?
  - perform cost-benefit or risk analysis
    - training costs
    - morale
      - e.g., programmers happier w/ Java
    - performance requirements
    - suitability of language for application
      - e.g., COBOL good for data processing vs. C good for real-time systems

General Principles in Programming Practice

1. TRY TO RE-USE FIRST
2. ENFORCE INTENTIONS

If your code is intended to be used in particular ways only, write it so that the code *cannot be used in any other way*.

General Principles in Programming Practice

Applications of *"Enforce Intentions"*

- If a member is not intended to be used by other functions, enforce this by making it private or protected etc.
- Use qualifiers such as final and abstract etc. to enforce intentions

General Principles in Programming Practice

"Think Globally, Program Locally"

Make all members ...

- … as local as possible
- … as invisible as possible
  - attributes private:
  - access them through more public accessor methods if required.

- Visualization?

Coding Standards

- What are Coding Standards
  - Coding standards are guidelines for code style and documentation.
  - The dream is that any developer familiar with the guidelines can work on any code that followed them.
- Why Have Coding Standards
  - Greater consistency between developers
  - Easier to develop and maintain
  - Saves time and money

Areas Typically Covered

- Documentation
- Naming Conventions
- Formatting Conventions

Documentation

- Self-documenting code
  - code w/o comments that is easily understood
- Code should be clearly documented so that
  - it can be understood easily & unambiguously
- Techniques
  - never write code first & then say you will comment later
  - write comments first (as spec) & then write code to implement it – change comments as needed
  - good prologue comments at top of every module
  - good comments within the module
    - explain what is happening at a high level or anything non-obvious

Good Programming Practice

- Prologue comments
  - brief description of module
  - programmer's name
  - date coded
  - parameters & description
  - variable names & description
  - files accessed/updated
  - error handling
  - name of file of test data (for regression testing)
  - list of modifications made, date, & by whom
  - known faults
- Code layout
  - use indentation
  - use blank lines to break-up large blocks of code

Class comments

- what do users of the class need to know about it?
  - purpose & status of the class
  - information about the instance variables
  - collaboration with other classes
  - example usage
  - what do subclasses have to redefine
  - ….
- Information about the author
- history of changes
- [see javadoc conventions for detail]
  - http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

Method comments

- Member Functions/methods Documentation
  - What and why member function does what it does
  - Parameters / return value
  - How function modifies object
  - Preconditions /Postconditions
  - Concurrency issues
  - Restrictions
- Internal Documentation
  - Control Structures
  - Why as well as what the code does
  - Difficult or complex code
  - Processing order

Good Coding Style

- Names
  - Use full English descriptors
  - Use mixed case to make names readable
  - Use abbreviations sparingly and consistently
  - Avoid long names
  - Avoid leading/trailing underscores

Naming variables & parameters

- Use consistent & meaningful variable names
  - who is the audience for your variable names?
    - the future maintenance programmers!
  - bad example
    - module contains vars `freqAverage, frequenceyMax, minFr, frqncyTotl`
    - do `freq, frequency, fr, frqncy` all refer to the same thing????
    - ordering is inconsistent!
    - if so, use identical/meaningful word (e.g., frequency)
      - `frequencyAverage, frequencyMaximum, frequencyMinimum, frequencyTotal`

Naming variables & parameters

- name reflects type of parameter
  - addTask(Task aTask)
  - assignTask(Task aTask, Agent anAgent)
- multiple parts: initial letter of every word capital

Naming conventions

- Class variable: All upper case
- Class names: Initial capital (Noun)
- Temporary variables: Lower case
- Instance variables: Lower case
- Method parameters: Lower case

Naming methods

- Name should illustrate the purpose ((Verb)
- choose name so that one can read expressions as a sentence
  anAccount.deposit(100)
  aTask.addInput(anInput)
- Predicates (return boolean): verb like “is” or “equals” concatenated with whatever is being tested
  isWaiting(anObject)
  equals(anObject)

Methods

- as high up in the inheritance hierarchy as possible
  - code reuse
  - if method does not access any aspect of the class: move it
- method: single function (do one thing only)
  - split into several methods if not
- length: readable without scrolling the screen (10-20 lines)
- use accessor methods to access instance variables
- method should send messages to a limited set of other objects
- many messages send to some other object (not: this): maybe method should be moved to the other class

File layout

0. package statement & import list
1. comment
2. class name
3. class variables
4. instance variables
5. constructors
6. main method
7. class methods

Import list

- Minimize * import
  - No: import java.util.*
  - Yes: import java.util.Vector
- Check if all imports are really used

Main method

- Main method: simple test of demo
  - testing & examples
- applications should have an own class containing a main method separate from the normal classes

Visibility

- Hide your members as much as useful
- *Never* declare instance variables as public

Exception handling

- Use throw and catch to handle errors
- Methods should return an appropriate object OR throw an exception to indicate a problem
  - do not return a specific value to indicate an error
- Use exception handling only for handling problems NOT for normal behavior

- More on visualization