Properties of High Quality Software

CSE219, Computer Science III
Stony Brook University

http://www.cs.stonybrook.edu/~cse219
Software Engineering Basics

Important Principles for creating a Software Solution:

• First, define the problem
• Design, then code
• Always Provide Feedback

• Learn a methodology for constructing software systems of high quality.
What properties make a software system of high quality?

- Correctness
- Efficiency
- Ease of use
  - for the user
  - for other programmers using your framework
- Reliability/robustness
- Reusability (i.e., code reuse with slight or no modification)
- Extensibility
- Scalability (i.e., to handle a growing amount of work in a capable manner)
- Maintainability, Readability, Modifiability, Testability, etc.
What properties make a software system of high quality?

- Correctness (think of GE or IBM large engineering systems)
- Efficiency (think of Google Search)
- Ease of use
  - for the user (think of Apple UI and products)
  - for other programmers using your framework (see MS Visual…)
- Reliability/robustness (think of NASA software)
- Reusability (see Apache Software Foundation software, e.g. HTTP server)
- Extensibility (see Android OS growth to most popular mobile platform)
- Scalability (think of Oracle DBs)
- Maintainability, Readability, Modifiability, Testability, etc.
Correctness

• Does the program perform its intended function?
  • And does it produce the correct results?
• This is not just an implementation (coding) issue
  • Correctness is a function of the problem definition
• A flawed Requirements Analysis results in a flawed Design
• A flawed Design results in a flawed program
  • Garbage In — Garbage Out
Efficiency

- Plan for efficiency
  - wisely choose your data structures & algorithms (including their complexity, e.g., $O(N)$) in the design phase.
  - tools & technologies too.
- Does the program meet user performance expectations?
  - If not, find the bottlenecks
    - done after implementation
  - called profiling
Ease of Use for End User

- Is the GUI easy to learn to use?
  - a gently sloped learning curve
- What makes a GUI easy to use?
  - familiar GUI structures
  - familiar icons when possible instead of text
  - components logically organized & grouped
  - appealing to look at
    - colors, alignment, balance, etc.
  - forgiving of user mistakes
  - help, tooltips, and other cues available
  - etc.
Ease of Use for other Programmers

• In particular for frameworks and tools
  • the Java API is developed to be easy to use

• Should you even build a framework?
  • Yes, you will be a software developer.

• What makes a framework easy to use?
  • logical structure
  • naming choices (classes, methods, etc.)
  • flexibility (usable for many purposes)
  • feedback (exceptions for improper use)
  • documentation (APIs & tutorials)
  • etc.
Reliability/Robustness

• Does your program:
  • anticipate erroneous input?
  • anticipate all potential program conditions?
  • handle erroneous input intelligently?
    • think about this in the design stage
• provide graceful degradation?
  • *Graceful degradation* (or *Fault-tolerance*) is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components.
    • If an error condition occurs in your program, should your program:
      ○ crash?, exit?, notify the user and exit?, provide an approximated service?
        Not always possible to save it.
    • For example: What should Web Browsers do with poorly formatted HTML?
Feedback

• Provide feedback to End users due to: bad input, equipment failure, missing files, etc.
  • How?
    • popup dialogs, highlighting (red text in Web form), etc.

• Provide feedback to other programmers using your framework due to: passing bad data, incorrect initialization, etc.
  • How?
    • exception throwing, error value returning, etc.
Flexibility in a Framework

• Programmers need to know:
  • when and why things in a framework might go wrong
  AND
  • when and why things in a framework do go wrong

• How?

  • customized response:
    • `System.out.println` notifications
    • `GUI` notifications
    • Web page generated and sent via Servlet notification
    • etc.
Applications Using Frameworks

- Making a framework is much more difficult than making a single application

Framework

App1 calls methods of Framework objects

Framework calls methods of App1 & App2 objects

Application #1

App2 calls methods of Framework objects

Application #2
Reusability

- Code serving multiple purposes.
- Who cares?
  - management does
    - avoid duplication of work (save $)
  - software engineering does
    - avoid duplication of work (save time & avoid mistakes)
- How can we achieve this?
  - careful program decomposition (from methods to classes and packages)
  - separate technology-dependent components
Extensibility

• Can the software easily be extended?
  • Can it be used for other purposes
    • plug-ins,
    • exporters,
    • add-ons,
    • etc

• Extensibility Example:
  • In NetBeans, Tools → Plugins
    • Anyone can make a plugin
    • Download, install, and use

  • In Eclipse IDE, Help → Install New Software plugin
Scalability

- How will the program perform when we increase:
  - # of users/connections
  - amount of data processed
  - # of geographic locations users are from
  - A function of design as well as technology
More Software Engineering steps

• Maintainability
• Readability
• Modifiability
• Testability
• etc.

• All of these, as with the others, must be considered early in design
How can these properties be achieved?

• By using well proven, established processes:
  • preferably while taking advantage of good tools

  • Software Development Life Cycle

  Requirements Analysis  →  Design & Document  →  Evaluate Design  →  Code  →  Test  →  Deploy

  Profile  →  Debug
Software Development Life Cycle

• Requirements Analysis & design stages:

• Correctness, Efficiency, Ease of use, Reliability/robustness, Reusability, Maintainability, Modifiability, Testability, Extensibility, Scalability

• do we consider these properties in the implementation stages?
  • Little because it is too late to make a big impact.
Where to begin?

- Understand and Define the problem
- the point of a requirements analysis
- What are system input & output?
- How will users interact with the system?
- What data must the system maintain?
- Generate a problem specification document
- defines the problem
- defines what needs to be done to solve the problem
Requirements Analysis

• i.e. Software Specification (spec.)
  • A textual document
  • It serves two roles. It:
    • defines the problem to be solved
    • explains how to solve it
  • This is the input into the software design stage

• What goes in a requirements analysis (RA)?
  • The why, where, when, what, how, and who:
    • Why are we making this software?
    • Where and when will it be created?
    • What, exactly, are we going to make?
    • How are we going to make it?
    • Who will be performing each role?
Requirements Analysis

• **What really goes in a RA?**
  • Detailed descriptions of all:
    • necessary data (including how to query it, views, forms, inserts)
    • program input and output
    • GUI screens & controls
    • user actions and program reactions

• **Where do you start?**
  • Interviews with the end users
    • What do they need?
    • What do they want?
UML Use Case Diagrams

• A set of scenarios that describe an interaction between a user and a system
• Done first in a project design
  • helps you to better understand the system requirements
• To draw a Use Case Diagram:
  • List a sequence of steps a user might take in order to complete an action.
  • Example actor: a user placing an order with a sales company
UML Use Case Diagrams

- **Human Actor**: Stick figure with name underneath. Name usually identifies type of actor.

- **Use Case**: Oval enclosing name of use case.

- **Non-Human Actor**: Stick figure, or a rectangle enclosing the stereotype `<<<actor>>` and the name of the actor. A stereotype indicates the type of UML element (when it isn’t evident from the shape).
UML Use Case Diagrams

• Relationships Between Actors and Use Cases:
  • Solid edge between an actor A and a use case U means that actor A participates in use case U.
UML Use Case Diagrams

- Relationships Between Use Cases:
  - **Include:** dashed arrow labeled <<include>> from use case U1 to use case U2 means **U2 is part of the primary flow of events of U1**.
  - **Extend:** dashed arrow labeled <<extend>> from use case U2 to use case U1 means **U2 is part of a secondary flow of events of U1**.

**Hint:** To remember the direction of the arrow, read the edge label as “includes” or “extends”.

Diagram:

- Withdraw Money <<include>> Authenticate
- Authenticate <<extend>> Authentication Error
Relationships Between Actors

- Generalization: Solid line with triangular arrowhead from actor A1 to actor A2 means that A2 is a generalization of A1. This implies that A1 participates in all use cases that A2 participates in. Generalization is similar to inheritance.

![Diagram showing generalization relationship between actors]

Member ➔ Make Reservation

Platinum Member ➔ Free Upgrade
Relationships Between Use Cases

- Generalization: Solid line with triangular arrowhead from use case U1 to use case U2 means that U2 is a generalization of U1 (equivalently, U1 is a specialized version of U2). Generalization is similar to inheritance.
Use Case Diagram For ATM

<<actor>>
Visa Authorization System

<<actor>>
Bank Information System

ATM System

Withdraw Using Visa Card
Withdraw Using Bank Card
Get Balance
Authenticate User
Deposit
Deposit Cash
Deposit Check

secondary

«includes»
«includes»
«includes»
«includes»

«extends»

Visa Cardholder
Bank Customer
Elite Bank Customer

(c) Paul Fodor & O'Reilly Media
Textbook example: Design well, then code.

- Head First Object Oriented Analysis and Design (chapter 1): “Rick decided to throw out his paper-based system for keeping track of guitars, and start using a computer-based system to store his inventory.”

Down and dirty!
Company produced:

AND

Rick started losing customers
More design needed to meet the user demands

Eliminate Strings and add enumerations of types

We’ve replaced most of those String properties with enumerated types.

The serial number is still unique, and we left model as a String since there are thousands of different guitar models out there way too many for an enum to be helpful.

The Guitar class uses these enumerated types to represent data, in a way that won’t get screwed up by case issues or errors in spelling.

Now the addGuitar() method takes in several enums, instead of Strings or integer constants.

Even though it looks like nothing’s changed in search(), now we’re using enums to make sure we don’t miss any matches because of spelling or capitalization.