COMP354

Software Engineering

Lecture 6

Principles

Definitions for Software Engineering

Product — what we are trying to build.

Process — the methods we use to build the product.

Method — a guideline that describes an activity. Methods are general, abstract, widely applicable. Example: top-down design.

Technique — a precise rule that defines an activity. Techniques are precise, particular, and limited. Example: loop termination proof.

Tool — a mechanical/automated aid to assist in the application of a methodology. Examples: editor, compiler, . . .

Methodology — a collection of techniques and tools.

Rigor — careful and precise reasoning. Example: an SRD should be rigorous.

Formal — reasoning based on a mechanical set of rules ("formal system"). Example: prog. language, predicate calculus.

Use *rigor* as much as possible.
Use *formality* when suitable tools are available (compilers, parser generators, proof checkers, . . .

Software Qualities

We need to be concerned with *quality* of *products* and *process* from an *internal* and *external* perspective.

qualities have different importance depending on project — maintainability is generally the most important

Correctness: software is functionally correct if it behaves according to the specification

Reliability: Can the user can depend on it?
eg what is the probability of error in a given time unit

Robustness: behaves "reasonably" even in circumstances not anticipated in the requirements specification

Performance: The software should have good space/time utilization, fast response times, and the worst response time should not be too different from the average response time.

Predicting performance

- measurement — monitor/profile actual product
- analysis — Big-Oh, queuing models
- simulation — run a physical simulation model

Friendly: The software should be easy to use, should not irritate the user, and should be consistent.

More Software Qualities

Maintainable: How easy is it to perform the following

corrective maintenance removal of residual defects in product

adaptive maintenance adapting product to changes in the environment

perfective maintenance improving the product
— better performance
— higher quality code and documentation
— requests from user/customer

For maintainability, software needs

- Easy to correct or upgrade.
- Code traceable to design; design traceable to requirements.
- Clear simple code; no hacker's tricks.
- Good documentation.
- Simple interfaces between modules.
- More later — *DESIGN FOR CHANGE*.

More Software Qualities

Verifiable: It should be easy to verify properties of the product such as correctness and performance

Reusable: How easy is it to reuse the software (in whole or in part) for other applications

Also reuse

- software libraries, eg scientific, Unix utilities
- window interfaces, eg X, Motif
- experience of engineer — application domain, tools
- design of components
- specifications
- components of methodologies and processes

Portable: software is portable if it can run in different environments eg hardware platforms, operating systems

Portability and efficiency are incompatible. Highly portable systems consist of many layers, each layer hiding local details. Recent achievements in portability depend on fast processors and large memories.

Interoperable: The software should be able to cooperate with other software (word-processors, spread-sheets, graphics packages, . . .).

Productivity: efficiency of the software development process

Timeliness: the ability to deliver a product to market on time

Principles of Software Engineering

People are Human

Separation of Concerns

- Modularity
- Incrementality

Abstraction

- Generality

Anticipation of Change

Rigor and Formality

People are Human

People make mistakes

Human intellect can handle only limited amounts of information (unless it is *structured*!)

Change is inevitable

Main Obstacles to Software Engineering

Re-work

Complexity

Change

Separation of Concerns

Very important principle with many applications in SE

To deal with *complexity*, **separate concerns** and look at each concern separately.

- separation in time
  - do requirements analysis before design
  - creative design in the morning, meetings in afternoon
  - do study during the week, personal interests on weekends
- separation by qualities
  - deal with correctness, then deal with efficiency
- separation by views
  - data flow through a system
  - control/decision making within system
  - concurrency and synchronization
  - timing restrictions of a real time system

  each view highlights different concerns
- separation into parts — modularity, abstraction

Examples of Separation of Concerns

- Specification (what) *vs* implementation (how). This applies to program modules, procedures, functions, statements, ADTs, . . .
- Correctness *vs* efficiency. Get it working first, then attend to performance.
- Functional *vs* non-functional requirements.
- Spreadsheet design: cell manipulation *vs* display.
- Application code *vs* user interface code.
- In-memory processing *vs* disk access.
  A typical sequence is logical data $\rightarrow$ blocked data $\rightarrow$ disk buffers $\rightarrow$ disk controller.

  In DOS: file name (user level) $\rightarrow$ file descriptor (DOS level) $\rightarrow$ disk address (BIOS level) $\rightarrow$ driver routines.
- Form *vs* content in word processing.
  Example: Latex style files.

Abstraction

concentrate on **general aspects** of the problem while carefully removing **detailed aspects**

concentrate on **important** aspects and downplay the **unimportant**

NB what is *important* or *general* varies
eg "person" record in medical database *vs* payroll *vs* sports performance analysis

- "How" abstracts to "what".
- Memory addresses abstract to variable names.
- Code performing a task abstracts to a procedure name.
- Bit string (C) abstracts to set (Pascal).
- In concurrent programming, we abstract away from linear time: given events E and E', we need to know only which must occur first. If delay is important, we have abstracted too much!
- Mathematics is the language of abstraction! Sets, functions, relations, graphs, trees, logic . . . are used because they provide models of things that we need.

Anticipation of Change

Change will occur!

write all documents and code under the assumption that they will subsequently be corrected, adapted, or changed by somebody else.

requires special effort to prepare for change/evolution

- of software: documentation, modularization, configuration management
- of processes: document the process, modularize the process
- of personnel: documentation, training

Rigor and Formality

rigour is achieved by conforming to given constraints

Examples of constraints

- using a given format for the requirements document
- documenting input and output of each procedure/function
- limits on number and size of modules
- following company standards for the development process

rigour is a necessary complement to creativity in SE

— it provides shape, direction, precision, and the ability to compare and measure

there are many levels of rigour

formality is highest degree of rigour:

it requires the process to be driven and evaluated by mathematical laws

Modularity

product or process composed of components/modules

greatly aids separation of concerns
— better understandability
— easier to change
— modules reusable

high **cohesion** within a module if all its elements are strongly related: ie there is a very good reason why they are together

low **coupling** between modules if the communication between them is simple

Incrementality

increment = a small step/change

It is easier to make small changes to a working system than to rebuild the system. Why? Because if the modified system does not work, the errors must have been introduced by the small changes

incremental prototyping

Generality

more general = able to handle more cases

use of general purpose tools, eg awk

more general program may be simpler and reusable
— fewer special cases in specification
— able to handle a broader range of inputs/uses