COMP 354
Introduction to Software Engineering

Greg Butler

Computer Science and Software Engineering
Concordia University, Montreal, Canada

Office: EV 3.219 Email: gregb@cs.concordia.ca

Winter 2015

Course Web Site:
http://users.encs.concordia.ca/~gregb/home/comp354-w2015.html

Principles of Software Engineering

Software engineering ...

... is a rapidly changing field ..

so what can we teach you

... that will still be important in 10–20 years time?

Principles of Software Engineering

People are Human

Separation of Concerns

Modularity

Incrementality

Abstraction

Generality

Anticipation of Change

Rigour and Formality

People are Human

People make mistakes

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

Change is inevitable

People are ... Main Obstacles to Software Engineering

Communication ... with people

Re-work .. due to mistakes

Complexity ... because we push the envelope

Change ... because we like it

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

separation by qualities

separation by views

separation into parts

Separation of Concerns

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 of Concerns

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

Separation of Concerns

Modularity

Separation into parts
Separate interface from implementation
Separate responsibilities
Nested modules = hierarchical decomposition

Incrementality

Separate process into increments
... by subsets of features or use cases

is separation into parts **and** separation in time

Modularity

Modularity is when a product or process is composed of components/modules

greatly aids separation of concerns

- better understandability
- easier to change
- modules reusable

high cohesion within a module when all its elements are strongly related

that is, there is a very good reason why they are together

low coupling between modules

coupling measures the interdependence between modules
eg module A and module B share a variable x is high coupling

Incrementality

increment = a small step/change

Incremental prototyping

identify a useful, but small, subset of the problem and implement it
build a sequence of prototypes by adding features one at a time
finally have a complete system

Advantages

- early delivery of a (sub)system
- early feedback from users
- later (more complicated) features may not be necessary
- more time to think about complicated features
- separation of concerns

Separation of Concerns — Examples

- Specification (what) *vs* implementation (how).
- 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 $\rightarrow$ block 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: CSS, Latex style files.

Abstraction

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

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

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

Abstraction — Examples

- "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.

Generality

more general = able to handle more cases

use of general purpose tools, eg awk
rather than write specific pattern matcher software

more general program may be simpler and reusable

- fewer special cases in specification
- able to handle a broader range of inputs/uses

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

Rigour 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 — a spectrum

formality is highest degree of rigour: it requires the process to be driven and evaluated by mathematical laws

Rigour and Formality

Remember ...

software engineering is systematic, disciplined, follows a process

want predictible ... quality, cost, schedule

this means .... we need **rigour**!

Remember what causes problems in Software Engineering

Requirements

Architecture

Change

Complexity