Software Design & Documentation

Engineering 7800 Guest Lecture

Dennis Peters
dpeters@engr.mun.ca
http://www.engr.mun.ca/~dpeters/
EN-3061, 737-8929

Spring 2004

Motivation

- A large percentage of both EE and CoE projects involve some software.
- Software Design is not intuitive to many.
- Most EE students have had little training in software design.
- SW design documentation is best done using standard notations that aren't common with other disciplines.

Outline

1) SW Requirements
2) SW System Decomposition
3) UML Diagrams

How is Software Different from Other Engineering?

- No natural (internal) boundaries.
  - Requires special skills to choose designs.
  - Components can interact in many ways.
- It's not constructed from materials that obey physical laws.
  - Interpolation is rarely valid.
  - Difficult to build in safety margins (can't extrapolate).
  - Doesn't wear out or break.
- Possible interactions with other systems (e.g., operating system, other programs etc.) are essentially infinite.

Requirements

- It's very difficult to get the design right if you don't know what it should do.
- Before you start designing your software invest some time in understanding the requirements.
- Describe "what" not "how".
- Functional requirements: what is to be done.
- Non-functional requirements: properties of the system that aren't related to how it behaves.
- Some useful tools:
  - Sketch of user interface.
  - Use cases.

Use Cases

Describe the interaction between users (*actors*) and the system to accomplish some user-defined goal.

- What actors are involved?
- What is the sequence of steps that occur in the 'normal' interaction.
- Are there deviations/extensions for special cases?

Example Use case: Robot Rescue

Actors: Rescue commander, victim.
System components: Robot, control station.

1) Robot is initialized at the entrance to a building.
2) Rescue commander enters command on control station for robot to search for a victim.
3) Control station relays command to Robot.
4) Robot enters building and begins to search.
5) Robot locates a potential victim.
6) Robot takes an image of the victim and sends it to the control station.
7) Control station displays the image.

8) Rescue commander indicates that the image is a victim.
9) Control station relays instruction to Robot.
10) Robot picks up victim.
11) Robot navigates its way out of the building.

Alternatives

At 8 commander indicates that the image is not a victim, robot continues to search, repeat from 5.

SW System Decomposition

- Goal is to define a set of *design entities* (modules, classes, packages, functions)
- Together the entities interact to implement the SW system behaviour.
- For each entity decide (and document):

  Name — try to be precise and descriptive.
  Responsibilities — what part(s) of the behaviour does it accomplish?
  Interface — what other entities know about this entity (for a class the public operations and attributes).
  Relationships with other entities — how is it related to other entities.

Techniques for Decomposition

- *Abstraction* — The interface to an entity should be much simpler than the implementation.
- *Encapsulation* (a.k.a. *information hiding*) — Each entity should 'hide' a design decision, such as
  - Implementation of algorithms,
  - Interface with some hardware/other system,
  - Data representation,
  - Increments.
- Strive for simple and small interfaces between entities (low coupling).
- The decisions to hide are the ones that are most likely to change.

Relations On Entities

Uses – if M_1 requires the presence of M_2 then M_1 *uses* M_2 (a.k.a. is a client of).

- not the same as *calls* or *associated with* (these are specializations of uses)
- if it is not a hierarchy then all modules in cycle must be implemented together (strong coupling)
- sub-trees in hierarchy indicate possible increments/family members
- aim for low fan-out and high fan-in

is component of – (a.k.a. aggregation)

passes data to

specialization of — (a.k.a. inheritance)

Incremental Design

It often makes sense (particularly so in this course) to approach the SW design and implementation problem as a sequence of *increments*.

- Increments occur in sequence — you finish working on one increment before you start the next.
- Each increment results in a running system.
- The success (or not) of an increment is apparent to the user of the system — the behaviour that the system should have is clearly defined.
- Increments are fixed in duration and short (two – four weeks).
- Plan for more increments than you expect to accomplish (i.e., some will be left for "future work") — if you are successful in the early increments then you can adjust to do more.

- Make sure that the highest priority parts are done in the early increments — if you run in to trouble you can survive with an early increment.

UML Diagrams

Class diagram Shows classes, their attributes, operations and relations.

Sequence diagram Shows interactions between objects to accomplish a particular behaviour.

References

[1] Martin Fowler. *UML Distilled: A Brief guide to the Standard Object Modeling Language*. Addison-Wesley, third edition, 2004.