Between Design and Implementation

Comp-303 : Programming Techniques
Lecture 16

Alexandre Denault
Computer Science
McGill University
Winter 2004

Announcements

- Assignment 3 will be handed out next week.
- Only 1 month left. You should be concentrating on your project.
- Today is the last day for software engineering. The last 6-7 classes will be dedicated to Design Patterns.

Last lecture . . .

- Functional requirements:
  - How does a correctly functioning program respond to correct and incorrect user interactions?
  - How does the program respond to hardware and software errors?
- Performance requirements:
  - How fast must certain actions perform?
  - What are the constraints on primary and secondary storage?
- Potential modifications:
  - What are likely changes or extensions to the product?
- Delivery schedule:
  - Which parts of the product need to be delivered early?

The purpose of design

- The purpose of design is to define a program structure consisting of a number of modules that
  - Can be implemented independently.
  - When implemented, will satisfy a requirements specification.
- The resulting structure should
  - be reasonably simple
  - avoid duplication of effort
  - support initial program development
  - support maintenance
  - support modification

The goals of design

- Meet the functional and performance requirements.
- Define a modular structure:
  - Define a structure that is simple & easy to implement.
  - Define a structure that facilitates modifications required by requirements analysis (modifications identified prior to design).

The process of design

- That topic is outside the scope of this course and is best left to a good software engineering course.
- There exist many proven software design techniques on the market.
- An example of this would be the Rational Unified Process which describes how to effectively deploy software using commercially proven techniques.
- Experience is very valuable when designing software.

Between Design and Implementation

- Before starting the implementation of a project, these two considerations often arise:
  - Evaluation of the design
  - Choice of a program development strategy

Why evaluate our design?

- Will all implementations of the design exhibit the desired functionality ?
- Are there implementations of the design that are inefficient ?
- Does the design describe a program structure that will make implementations easy to build, test and maintain ?
- How difficult will it be to enhance the design to incorporate future modifications, especially those identified during the requirements analysis ?

Correctness & performance

- Testing and informal verification were used previously to increase confidence that a program behaves as required.
- For designs, these are not feasible.
- Instead, we use design reviews.

Design Reviews

- Design reviews . . .
  - should be systematic
  - should examine both local and global properties
- Local properties: specification of individual modules
  - consistency
  - completeness
  - performance
- Global properties: how do the modules fit together?

Reviewing performance

```
void sort (Vector v) throws ClassCastException
  // MODIFIES v
  // EFFECTS: if v is not null, sorts it into
  // ascending order using the compareTo method
  // if some elements of v are null or are not
  // comparable throws ClassCastException
```

- The effect clause does not specify any performance constraints.
- Allows the programmer to build the sort however he wants.

Reviewing performance (cont.)

```
void sort (Vector v) throws ClassCastException
  // MODIFIES v
  // EFFECTS: if v is not null, sorts it into
  // ascending order using the compareTo method
  // if some elements of v are null or are not
  // comparable throws ClassCastException
  // worst case time = n*log(n) comparisons
  // where n is the size of v
```

- Allows performance estimates when using sort.
- Definition is more restrictive for the implementation.

Reviewing performance (cont.)

```
void sort (Vector v) throws ClassCastException
  // MODIFIES v
  // EFFECTS: if v is not null, sorts it into
  // ascending order using the compareTo method
  // if some elements of v are null or are not
  // comparable throws ClassCastException
  // worst case time = n*log(n) comparisons
  // where n is the size of v
  // Maximum temporary main memory allocated
  // is a small constant
```

- Eliminates a few possible implementation.

Walkthroughs

- Estimate performance of each module so that worst-case and average efficiency estimates for whole program can be constructed.
- Walkthroughs are laborious and imprecise.
  - Designers seldom examine their own designs adequately.
  - They should be performed by a team including but not dominated by designers.
- With a small set of inputs, the team can trace through entire design and discover gross errors in how the abstractions fit together.

Reviewing design structure

- It is also important to evaluating the appropriateness of the module boundaries.
  - Have we failed to identify an abstraction that would lead to a better modularization ?
  - Have we grouped things together that do not belong in the same module ?
- There are no magic formula for detecting bad modularization, but there are symptoms.

Coherence of procedures

- Each procedure should represent a single coherent abstraction.
- A procedure should perform a single abstract operation on its arguments.

Signs of incoherence

- The easiest way to specify a procedure is to describe its internal structure (how it works instead of what it does).
- If the best name you can come up with is *procedure1*.

Type coherence

- Each type should provide an abstraction that users can conveniently think of as a set of values and a set of methods intimately associated with those values.
- Check each method and see whether it really belongs in a type.
- A type should be adequate: provide enough methods so that common uses are efficient.
- Badly designed types contain methods that are not relevant to the abstraction.
- For example, in a Stack data type, a method to square the value of the top element would not be relevant.

Communication between modules

- Modules should have narrow interfaces.
- If too much information is passed between modules, this indicates that a type abstraction is needed.
- Even if a type is passed, we must make sure the type may be too heavy.
  - For example, passing a student record when only the address is required.
  - Instead we should provide an Address type.

Reducing dependencies

- The use of narrow interfaces (e.g. no global variables) reduce dependencies.
- Strong dependencies can be changed into weak ones using types that do not depend on the interface.
- For example:

  A Dictionary class depends strongly on the Doc class because Dictionary.correct() calls Doc.words() to get all the words of the document. However, if the programmer provided Dictionary with Doc's word iterator, Dictionary would require no knowledge of Doc.

- Reducing dependencies reduces the risk of anomalous behavior after modification.

The development process

- Basic choice: top-down or bottom-up
- Top-down: all modules that use module M are implemented and tested before M is implemented.
- Bottom-up: all modules used by M are implemented and tested before M is implemented.

Bottom-up

- Advantage:
  - We can avoid the use of stubs (dummy code for missing modules).
  - Bottom-up may place less demand on system resources (e.g. tests run in less memory than full system tests).
- Disadvantage:
  - The next implemented module may depend on errors or implementation specific behavior of used modules.
  - Useless efforts may be spent on a module if we discover that the previous modules are badly designed.
- Both:
  - Bottom-up leads to development of useful subsystems usually wider applicable than top-down partial systems.

Top-down

- Advantage:
  - We can avoid the use of drivers (testing code for modules).
  - Serious design errors will be caught earlier.
  - Top-down is required for type hierarchies (because we need decide which functionality belongs in the root).
- Disadvantage:
  - If the using module does not the depending module thoroughly, specific use of a module may reveal errors.

The development strategy

- A development strategy should be defined explicitly before beginning the implementation.
- Generally a mixed strategy should be used with preference for top-down.
- Bottom-up should be used for modules that are easier to implement than to simulate.
- Top-down simplifies system integration and test.
- Top-down produces useful partial versions of system.
- Top-down catches critical high-level design errors early.

Summary

- The purposes, goals and process of design.
- Between design and implementation, a systematic design review should take place.
  - Procedures
  - Types
  - Modules
- Before the implementation, several big decision must be made : top-down vs. bottom-up.

CSGames 2004

- The event Team Software Engineering tested the communication skills and the design skills of a team.
- The event was won or lost in the 30 first minutes.
- Teams that understood the value of design and the knew their limit did well.
- Very few school understood this properly …
  - Universite du Quebec en Outaouais
  - McGill University
  - Bishop's University
  - Queen's University
  - University of Waterloo

Tool of the day: Ant

- Ant is a Java-based build tool (like make).
- Why another build tool?
  - Ant is written in Java, so it is truly multi-platform.
  - Ant is extended using Java classes, not shell script.
  - Ant has a simple XML file format.
- Many IDE support Ant out of the box:
  - NetBeans
  - jEdit
  - Eclipse
  - JBuilder
- More information on Ant is available at:
  `http://ant.apache.org/`