Software Engineering

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

October 27, 1996

Aspects of Software Quality

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

October 27, 1996

Outline

- Quality Attributes of Software
- Software Principles that Support Quality

References:

This is material from Chapter 12 of Sommerville as well as from
Shach's *Classical and Object-Oriented Software Engineering*
Booch's *Object-Oriented Analysis and Design*
Meyer's *Object-Oriented Software Construction*

Quality Attributes I

External aspects

- Correctness

  Performs exactly to specifications

- Robustness

  Handles abnormal conditions gracefully

- Adaptability

  Adaptable to changes in specifications

- Reusability

  Reusable wholly or partly for new applications

- Compatibility

  Usable in combination with other products

Quality Attributes II

More external aspects

- Efficiency

  Good use of time and computer resources

- Portability

  Easily transferrable to other environments

- Verifiability

  Easily monitored and permits easy preparation of test cases

- Integrity

  Secure against unauthorized acess and modification

- Human Factors

  Easy to learn and use

Influencing Quality

Internal factors

- Design simplicity
- Design consistency
- Decentralized modularity
  - Decomposability
  - Composability
  - Understandability
  - Continuity
  - Protection

Decomposition

Top-down design

- Functionality
  - Start with high-level view of process
  - Refine process into logical components
  - Look for similar components; combine and abstract them
  - Review other possible arrangements
  - Repeat where necessary on components
- Data
  - Start with high-level view of information
  - Refine information into logical components
  - Look for similar components; combine and abstract them
  - Review other possible arrangements
  - Repeat where necessary on components

Composition

Reusable design

- Review combinations and abstractions
- Look for
  - Well-defined, concrete, general tasks
  - Tasks useful in widely different contexts
- If possible, do further
  - Generalization, abstraction, combination
- Provide utilites

Understandibility

Accessable to the human reader

- Clear, streamlined documentation
- Consistency of style
- Clear, logical, consistent naming
- Design simplicity
- Short units $(6 + -2)$
- Independent units

Continuity

Small specification or environment changes affect few modules

- Structure of system should be left intact
- Limited module interdependence
- Symbolic constants
- Localized repository of definitions and dependencies

Protection

Errors do not propagate; modules are encapsulated

- One module independent of another's internals
- Strict control of data access
- Abnormal conditions remain within module
- Limit on error propagation
- Input validated by each module
- Provide for exception handling

Software Principles that Support Quality

- Linguistic modular units
- Coupling
  - Few interfaces
  - Small interfaces
  - Explicit interfaces
- Encapsulation/Information hiding
- Locality
- Minimality
- Cohesion
- Abstraction
- Composition
- Adaptability

Linguistic Units

Modules correspond language's syntactic units

- Separately compilable/instanceable to support
  - Decomposability

    Separate tasks clearly identified
  - Composability

    Can be combined separately
  - Protection

    Syntactically delimited
- C
  - procedure/function
  - struct
  - `.h`/`.c` file pair
- C++
  - procedure/function
  - class
  - `.h`/`.cc` file pair

Coupling

As few pairs of agents as possible should interact with each other, and when two agents interact, the complexity of their interaction and the variety of information they exchange should be minimized. (observe: long argument lists to procedures should be avoided, bunches of data used together should be combined under a common name, strive for data-flow and interaction structures that involve few nodes and few edges, minimize the number of member functions in classes)

Few Interfaces

Modules communicate with as few others as possible

- Understandibility

  Each link complicates the structure

- Continuity

  Each link is an interdependance

  Each link can propagate errors

- Composability

  Each link further limits composability

- Protection

  Each link reveals some information

Small Interfaces

Modules exchange as little information as possible

- Tight coupling

  Many, explicit details exchanged

  Weak coupling

  Few, encapsulated items exchanged

- Understandibility, Continuity, Composability, Protection

  Should be obvious

- Bad example

  `computePay(EmplName,EmplSIN,HrsWrkd,EmplRate)`

- Good example

  `computePay(Employee)`

Small Interfaces and Information Hiding

Modules' information exchange must be obvious

- Explicit coupling

    Modules `A` and `B` exchange through procedure calls

  Implicit coupling

    Modules `A` and `B` share (global) data

Each detail of module should be private unless declared public

- C
    - Scoped within files
        - ▹ **`static`** within texttt.c with few exceptions
        - ▹ **`extern`** within **`.h`** only for selections
    - Scoped within procedures/functions
    - But no access control on **`struct`**
- C++
    - Scoped within files
        - ▹ As in C
    - Scoped within procedures/functions
        - ▹ As in C
    - Scoped within class
        - ▹ **`protected`** only where necessary for subclasses
        - ▹ **`private`** with few exceptions

Encapsulation/Information Hiding

The details of an agent's representation and implementation should not be revealed unless absolutely necessary. If details have to be revealed, the design probably needs to be improved. (at the very least: no system-global variables, no public variables, but that is not the whole issue, merely a trivial observation.)

- Design simplicity
  - Details in using agent are minimized
- Decomposability
  - Agents provide natural decompostion units
- Composability
  - Agents can be designed for plug-in use
- Understandability
  - What is hidden cannot confuse
- Protection
  - If access is restricted, harm cannot be introduced
- Continuity
  - Changes to implementation do not propagate outward

Locality

Code that provides related services should be organized into a common, coordinated set of agents (and files). Each agent should be responsible for a clearly defined, clearly delimited number of services and data.

- Design simplicity
  - Avoids changes scattered about the project
- Understandability
  - Items conceptially close are spatially close

Minimality

No unit, be it class, procedure, module, or hierarchy, should do too many things or gratuitously offer varieties of ways to do the same thing. Each agent should be responsible for a clearly defined, clearly delimited number of services and data. The clearly delimited number of services should be as small and focussed as possible.

- Design simplicity
- Understandability
- Composability and Decomposability
  - Providing minimality requires careful decomposition
  - Well-designed decomposition often leads to reusable components

Cohesion

Distinct functionality should be in distinct agents. Each agent should represent a logical, focussed coordinated set of services.

- Decomposability
- Composability
- Understandability

Abstraction

Look for generalizations for groups of agents you are thinking about designing. Concentrate the common behavior and the shared data that you find for a group of agents into a single, abstract agent, and represent the particularities of the actual agents as variations and additions.

- Design simplicity
  - Good abstractions make clean design
- Design consistency
  - If abstractions do not “mix metaphors,” good abstractions contribute to consistent design
- Understandability
- Decomposability and Composability
  - Good abstractions lead to generic code
- Continuity and Protection
  - Abstractions lead to encapsulated design

Composition

Agents that must carry out a broad range of activities are best designed as a composite of simpler agents, each of which specializes on some related set of activities.

- Design simplicity
- Design consistency
- Understandability

Adaptability

Good design puts everything in a locality, as much a possible, so that it is easy to find all the places that changes need to be made, and all those places are proximate. If that is impossible, then cross referencing and highly visible and indexed linking must be provided to ensure that all dependencies are recognized and found.

- Composability
- Decomposability
- Understandability

Managerial Input

Design Consistency and Understandability

Various style conventions should be established and strictly enforced

- Naming conventions
- Code formatting conventions
- Call/Return conventions
- Look-and-Feel conventions
- Documentation and commenting conventions
- • •