Software Engineering Fundamentals

Q: What makes a good program?

A: A good programmer.
A joke, but . . .

- Studies have shown that productivity of professionals varies by a factor of at least 10.
- Quality of product also varies greatly.

What makes a good programmer?

- Aptitude
- Attitude — Enthusiasm, professionalism.
- Education

Attributes of good software:

- Utility — is it useful? Does it work? (correctness)
- Performance.
- Cost (of production).
- Modifiability — requirements *will* change.
- Reusability.
- Understandability (affects correctness, modifiability).
- Documentation (affects modifiability, reusability).

Q: Does it matter if my software is good?

A: Yes! Bad software can, has, and will cost lives and $.

Modularity

Module — A collection of components that together achieve a common purpose (the *service* provided by the module).

Many languages have explicit or implicit support for modules:

- Ada — packages
- Modula — modules
- Turbo Pascal — units
- Java — packages, classes

In C a module is usually represented by two files: ".h" (interface) and ".c"(implementation)

In C++ classes and objects support modular design (also usually in .h and .cpp files).

Modular Design

- Consider how to decompose system into modules.
  - What are the abstract concepts involved in the solution?
  - Which design decisions are most likely to change?
  - Can we isolate the effect of likely changes?
- Design the interfaces to the modules.
  - The *interface* is defined by the set of assumptions that other modules may make about a module.
  - Try to make the interface small, but effective (what *service* does the module provide).
  - Be wary of implicit aspects of the interface — those that don't appear in the code.
  - Document the interface carefully!
- Implement the modules.

Information Hiding

Each module hides one or more "secrets" — things that the implementer of the module needs to know, but the implementers of other modules don't need to know.

This controls complexity: separation of concerns.

Some common varieties of secrets:

- Format of inputs and outputs
- Gross behaviour of the system
- Arbitrary facts (e.g., taxation rules)
- Algorithms
- Machine dependencies
- Methods of data representation (a.k.a. data structures)

Example: Chess Program

Module	Service	Secret
Command	Executes user commands	Command meaning
Read	Gets commands from the user	Command input method/format
Display	Displays the board to the user	Format of display, nature of display device
Select	Selects the move to make	Game strategy
MoveGen	Produces a list of legal moves	Rules of the game
Board	Records states of the game	Method of storing game state

Advantages of Modular Design

- Reduced complexity
- Units of work
  - Design
  - Implementation
  - Testing/validation
- Simplify change
- Reusability
- Program families

Procedural Abstraction

Get the job done without worrying about how it is done (like management).

Most languages have functions such as `sqrt` or `sin`

- we use them knowing **what** they do (their specification),
- but not **how** they do it (their implementation).

Another example: `void sort(int a[], int n)`

We know what it does — rearranges the elements of `a` in ascending order.

We don't care how it does it.

Defn: *Procedural Abstraction* — Describing subroutines (procedures, functions) in terms of what they do rather than how they do it.

Data Abstraction

- Modularity applied to data representation.
- Goal: localize all code that depends on the method of representing the data.
- Many languages allow localization to be enforced:
  - C: "static" variables
  - Java, C++: "private" data members
- ADT: interface to data storage modules is independent of the method of representing the data.

Objects

An *object* is a region of memory (a.k.a., variable).

We say an object o is an *instance* of class c if the type of o is c.

In OOP a class will usually define *methods* (a.k.a., *member functions*), which can be classified as:

Creators (a.k.a., constructors) — create new objects

Destroyers (a.k.a., destructors) — clean up when object is no longer needed.

Accessors — return information about the object without changing it.

Mutators — change the value stored in an object

Complex ADT

- Creators
  - `Complex(double r, double i)` — Creates a complex number approximately equal to $r + ij$
  - `Complex(const Complex& x)` — Creates a complex number that is a copy of x
  - `Complex operator+(const Complex& x)` — Creates a complex number approximately equal to x+ this complex number (the *recipient*).
  - . . . similarly for other math operations
- Destroyers
  - `~Complex()` — Destroys a complex number

- Accessors
  - `double getReal()` — Returns the real part of this complex number.
  - `double getImaginary()` — Returns the imaginary part of this complex number.
  - `double getMagnitude()` — Returns the magnitude of this complex number.
  - `double getPhase()` — Returns the phase of this complex number.
  - `bool operator==(const complex& x)` — Returns `true` if `x` is approximately equal to this complex number.
- Mutators
  - `Complex& operator=(const Complex& x)` — (Assignment operator) Copies the value of x over this complex number.

Comments

A good ADT is:

- Easy to understand without reference to implementation.
- Efficient to implement.
- Useful as a component of a larger program.

In C++ the "public" part of the "class" construct is a good place to document the ADT.

The "private" part of the "class" construct is a good place to document the method of representation.

In C++ normally the data members are private, and member functions are public.

Documentation

- An essential part of Engineering: *Design* documentation
- Distinct from *User* documentation
- Audience: other programmers
  - Designing/implementing other modules in the system.
  - Testing this module.
  - Modifying of this module.
- Rarely need to document algorithms (**how**) — the code describes that.
- Concentrate on the **what** and **why**
- Strive for **precision** and **clarity**

ADT Documentation

Describe the ADT in abstract terms:

- How is the *state* of an object represented (usually) in terms of mathematical types (e.g., integer, real, set, sequence, function)?
  - *invariant* — a predicate that is true whenever no member function is executing (i.e., before and after each function executes).
- What are the operations on the ADT (creators, destroyers, accessors, mutators)?
- What is the behaviour of each operation in terms of the **abstract** state?
  - *pre-condition* — a predicate that the function assumes to be true before it executes.
  - *post-condition* — a predicate that the function will ensure is true when it finishes.

This is a design document that is independent of any code.

Example: Chess Board ADT

Description Represents the state of a chess board.

State A set of pairs: $\mathbf{Board} \subset \mathbf{Piece} \times \mathbf{Location}$

where $\mathbf{Piece} \stackrel{\text{df}}{=} \left\{ \begin{array}{l} \text{BlackPawn1}, \text{BlackPawn2}, \ldots \text{BlackPawn8}, \\ \text{BlackRook1}, \text{BlackRook2} \\ \ldots \text{ (all the chess pieces)}, \\ \text{Empty} \end{array} \right\}$

$\mathbf{Location} \stackrel{\text{df}}{=} \{(col, row) \in \{\text{a} \ldots \text{h}\} \times \{1 \ldots 8\}\} \cup \{\text{Taken}\}$

Invariant Each piece, except Empty, is in exactly one location, and each location, except Taken, has exactly one piece.

$(\forall p \in \mathbf{Piece}, p \neq \text{Empty} \rightarrow |\{l \mid (p, l) \in \mathbf{Board}\}| = 1) \wedge$
$(\forall l \in \mathbf{Location}, l \neq \text{Taken} \rightarrow |\{p \mid (p, l) \in \mathbf{Board}\}| = 1)$

Operations

– $\mathit{ChessBoard}()$
  Constructor. Initializes the board to contain the pieces in the standard chess starting positions.
  Post: $\mathbf{Board} = \{(\text{WhiteRook1}, (a, 1)), \ldots$ (standard positions)$\}$
– $\tilde{}\mathit{ChessBoard}()$
  Destructor.
– $\mathbf{Piece}\ \mathit{getPiece}(\mathbf{Location}\ l)$
  Returns the piece at location l on the chess board for any location except Taken.
  Pre: $l \neq \text{Taken}$
  Post: $(\mathit{Result}, l) \in \mathbf{Board} \wedge \mathbf{Board}' = \mathbf{Board}$
– $\mathbf{Location}\ \mathit{getLocation}(\mathbf{Piece}\ p)$
  Returns the location of piece p for any piece except Empty.
  Pre: $p \neq \text{Empty}$
  Post: $(p, \mathit{Result}) \in \mathbf{Board} \wedge \mathbf{Board}' = \mathbf{Board}$

– $\mathbf{void}\ \mathit{setLocation}(\mathbf{Piece}\ p, \mathbf{Location}\ l)$
  Sets the location of p to be l for any piece except Empty.
  Pre: $p \neq \text{Empty} \wedge (\text{Empty}, l) \in \mathbf{Board}$
  Post: $(p, l) \in \mathbf{Board}' \wedge$
  $\neg(\exists l_0, (l_0 \neq l \wedge (p, l_0) \in \mathbf{Board}')) \wedge$
  $(l \neq \text{Taken} \rightarrow \neg(\exists p_0, (p_0 \neq p \wedge (p_0, l) \in \mathbf{Board}'))$

Notes

Result refers to the value returned by a function.

The ‘primed’ variables (e.g., $\mathbf{Board}'$) refer to the state after the function has executed, undecorated variables refer to the state before it’s executed.

Formal (mathematical) notation is very precise, and (if written correctly) shouldn’t be misunderstood, but

- it’s easy to get it wrong, and
- it can be difficult to express some ideas.

Be formal where you can, but don’t let clarity suffer because of it.

I’d rather get correct informal documentation than incorrect formal.

Class Documentation

Two parts:

Interface (public part)

– What ADT is represented.
– Either give the ADT documentation, or refer to another document that contains it.

Implementation (private part)

– Relationship between variables and ADT state.
– What other classes are used.

Example

```
class ChessBoard
{
  // ...
  private:
    Piece board[8][8]; // Piece in each board location
    // board[r][int(c-'a')] = p <-> (p, (c, r)) in Board

    list<Piece> taken; // Pieces with location = Taken
};
```

See chessBoard.h.

Method Internal Documentation

In the implementation (.cpp) file:

- Algorithm should be essentially self-documenting. If not, explain it.
- Explain what each local variable represents (ideally in terms of ADT).
- *loop invariant* — a predicate that's true before each iteration of a loop, and when the loop is exited. It should help to show that the loop accomplishes its role in the program.