SCD Finals

Defensive Programming

- Means to protect a routine from being broken by invalid inputs.

Rule 1: You protect yourself/code all the time.

Rule 2: Never trust user

=> Strategy/Ways

(1) Check the value of all ~~input~~ data from external sources

(2) Check the value of all routine input parameters

(3) Decide how to handle bad inputs.

Assertion

- Technique used for defensive programming to check input values.
- code that's used during development
- It is a logical formula inserted at some point in a program.
- An assertion usually takes two arguments: A boolean expression that describes that assumption that's supposed to be true, and a message to

display. Here's an example in Java

assert denominator != 0 : "Denominator is equal to 0.";

→ denominator != 0 => is boolean exp.

→ "Denominator is equal to 0."; => is message to display if first one is false.

Used

1- An input/output falls within expected range

2- A file is open/closed as expected

3- A pointer is not null.

4- Array index out of ~~range~~ bound.

5- Objects/Arrays initialized properly.

6- A container is empty/full as expected.

7- User assert to check the conditions that should never occurs.

8- Verify pre-conditions/post-conditions

- Avoid putting ~~code~~ executable code in assertion. → Putting the executable code in assertion raise a possibility that the compiler will eliminate the code when you turn off the assertion.

ee

Error handling typically checks for bad input data; assertion check for bugs in the code ""

⇒ In some circumstances, both assertion and error handling code might be used to address the same error.

Code Review

- Systematic Inspection of a software

Quality Assurance

- Set of systematic activities designed to ensure that the system meets its objectives

① Reviews

- Systematic inspection of a software
- Done by internally
- Whit box testing

② Audit

- Systematic ~~inspection~~ examination and evaluation of processes and activities.
- Done by externally (Black box)

Reviews Types

1- Peer Review

- After you finish part of a program

you explain your source code to another programmer.
- Offline version of pair programming

Pair Programming:

- Two programmer works on same code. One types / writes the code with keyboard while other inspect the code / watches for mistakes and thinks strategically about whether the code is being written correctly

Advantage

① Collaboration makes a program better in quality and stability.
② Catch most bugs.
③ Catch design flaws early.
④ More than one person has seen the code
⑤ Forces code authors to articulate their decision and participate in the discovery of flaws.
⑥ Accountability : Author + Reviewer
⑦ Allow juniors to learn from ~~code~~ seniors without lowers the code quality.

=> Who should review?

① Other developers / Partner

② Other developers from other teams

③ Other developers or team of developers outside from ~~team~~ your own team.

=> Where should conduct it?

① In meeting

② On a decided place

③ The artifact should be shared before.

Focuses

- Error prone code
- Previsouly discovered problem type
- Security
- Standard Checklist -> Rules or style of writing a code in an organization or company.

2- Managment Review

- The main parameter of managment reviews are project cost, schedule, scope and quality.
- It evaluates decision about:

① Corrective actions

② Changes in the allocation of resources.

③ change to the scope of the project
→P.Manager only checks the quality of the code.

3- Personal Reviews

- Reviews your own code to enhance and ensure quality.
- You review your own code either it is effecient and followers the rules.
- Reviews before Peer Review ~~and~~, Managment review or Audit.

② Audit

- SQA Testing (Software Quality Assurance)
- Conducted by SQA Team
- Black Box Testing (Don't tests the code internally.

Types

1- Product Assurance: Evaluating the quality of end product - Includes reviews of design documents, code, test cases. Involve assessing whether the product meets the specified requirements.

2- Process Assurance: Evaluating the

process used in the development
- Involves assessing efficiency of processes
- Includes reviews of process documentation, workflow and procedure and aims to enhance

Distinguishing Reviews types and Audit types

① Purpose ② Role ③ Activity
④ Level of independence
⑤ Tools and technique

Code Refactoring

- Improving a piece of software internally without altering its behaviour externally.
- "DRY Principle : Don't Repeat Yourself"
- "Copy and Paste is a design error"

Each part of your code has 3 purpose:
1- Execute functionality
2- Allow change → No tight coupling but can further grow. (Easy to maintain)
3- Communicate well to developers
↳ Well documented code (Add comments).

Why Refactoring?

- Readability
- Reusability
- Maintainability
- Performance & Optimization

Reasons to Refactor

1. Duplicate code.
2. A long routine - A routine is too long.
3. A loop is too long or too deeply nested.
4. A class has poor cohesion.
5. A class interface does not provide a consistent level of abstraction.
6. A parameter list has too many parameters.
7. Changes within a class tend to be compartmentalized.
8. Changes require parallel modifications to multiple classes.
9. Inheritance hierarchies have not modified in parallel.
10. Related data items that are used together are not organized into classes.
11. A routine uses more features of another class than of its own class.

(12) Primitive datatypes are overloaded.
(13) A class doesn't do very much.
(14) A chain of routines passes tramp data.
(15) A middleman object isn't doing anything.
(16) Data members are public
(17) Global variables are used.
(18) Comment are used to explain difficult code
(19) Subclass use only a small percentage of its parent's routines.
(20) One class is overly intimate with another.

When to refactor

- Best practice is continuously. As a part of the development proces
- It gets harder to refactor at end.
- Changes effects huge part of software.

Principles

1- Low level Refactoring

(1) Naming

1- Don't use dummy names, use proper names for variables, routines and classes.

- Avoid using 'magic' constants.

Magic constants => Anti pattern of using numbers for constant names. Eg.

int var = 3.1415 → This is value of PI.

2- Procedure / Routine

1- Extra code into method.

2- Extra common function into method.

3- Inlining an operation.

Inlining

Compiler copies the code from function definition directly into code of calling function. rather then creating seperate set of instructions in memory.

4- Changing operation signatures → Overloading.

Reordering

1- Split one operation (Big operation) into several smaller methods.

→ To improve cohesion and readability.

2- Put the semantically related statements near each other physically in your program.

GOD class => A class that do everything

How to refactor a GOD class?

Step 1

- Identify / catagorize related attributes and operations
  - ↳ From class diagram
- Put together related items (cohesion)
- Find the natural home of operations.

Step 2

- Remove all transient association.

Transient: A property of any element in the system that is temporary.

Transient association: Association that should not be accessed directly.

500% ROI

ROI → Return over investment

Software Evolution

Various experts have asserted that most of the cost of software ownership arise after delivering software i.e maintainence.

Types

1- Corrective maintainence

This encomparses fixing bugs / features.

2- Adaptive maintainence

This includes software adaption to changes.

3- Preventive maintainence

This type of maintanence deals with the improved software by fixing bugs before they activate.

4- Perfective maintainence

This caters improved software in terms of performance & maintainability.

Manny Lehman

Lehman Laws

S-type → Static

E-type → Evolutionary

E-type → (mostely used in real world)

Real world system

8 Laws:

1- Law of Continuing Change

An E-type system must be continuing / continually adapted else it becomes progressively less satisfactory in use.

2 - Law of Increasing Complexity

As an E-type system is evolved its complexity increases unless work is done to maintain it or reduce it.

3- Self Regulation

Global E-type system evolution processes are self - Regulating.

4- Conservation of Organizational Stability

Average activity rate in an E-type process tends to remain constant over system lifetime.

5- Conservation of Familiarity

The average and incremental growth of E-type system tends to decline.

6- Continuing Growth

The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.

7- Declining Quality

The quality of an E-type system.

will appear to be declining as it is evolved unless rigorously (adapted to ~~maintained~~ to operational environment changes

ee Unless rigorously adapted, quality will appear to decline over time "

8- Feedback System

E-Type evolution processes are multi-level, multi-loop, multi-agent feedback system.

Legacy systems

Outdated systems that are still in use. Problem include:

① Architecture degradation
② Reliance on unmaintained software or hardware
③ Loss of experties.
④ Not designed for evolution.

② High level Refactoring

→ More important than low level refactoring
→ Improves overall structure of your problem / code.

Principles

1- Exchange obscure language idioms with safer alternatives.

2- Clarify statements that has evolved over time using comments.

3- Performance Optimization.

- Design level
- Algorithm
- Data structure
- Source code

ce Compared to low level refactoring, high level is not well supported by tools

SCD

13-12

Source Code Layout & Style

"Any tool can write code that a computer can understand. Good programmers write code that human can understand."

— Martin Foules.

Layout

- It does not affect execution speed and memory consumption.
- It affects how easy is it to understand the code, review & revision after months.
- It also affects other developers readability,

CH# 31

understanding and modification in your absence.

Fundamentals

① Logically Organized

- Proper use of wide spaces (indentation, new lines,

② Consistent

Fundamental theorm of Formatting

- Good visual layout shows the logical structure of a program.

Note: Technique make good code look and bad code look bad.

Techniques:

- Proper use of ~~white~~ whitespaces
  (a) Grouping
  (b) Blank lines
  (c) Indentation
- Proper use of paranthesis

Style

A block of related/similer code use "~~begin~~" "begin" and "end". It is clear by looking at the code that particular block of code starts or end.

Control Structure Layout

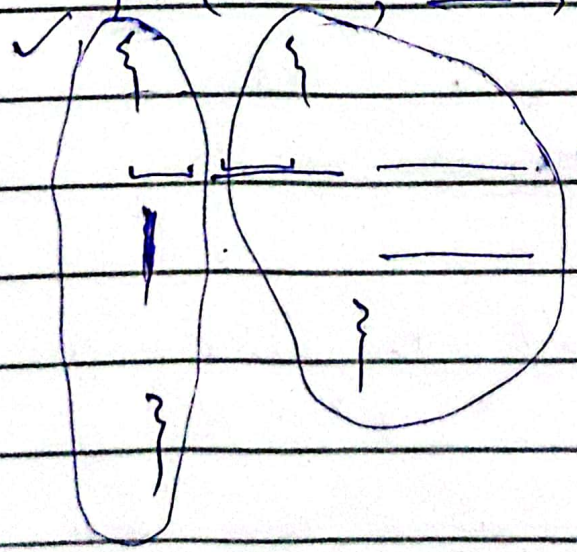
(a) Avoid unintend begin-end pairs

e.g

```
for(int i=0; i<10; i++)
{  Statement A
   Statement B
}
```

(b) Avoid double intendation with begin and end

e.g




```
if(exp)
  stat. A;
if (exp) {
  stat. A;
}
if(exp)
{
  StatementA;
}
```

(c) Use blank lines b/w paragraphs

(d) Format single statement block consistently

```
if (exp) statement A;
```

(e) For complicated expressions put separate expressions on seperate lines.

```
if (expA &&
    expB ||
    expC ) {
      statement A;
    }
```

(f) Avoid GoTOs (It makes program hard to format)

(g) No endline for case statement (exceptional)

Not recommended

```
switch (exp) {
  case A: statement;
          break;
  case B: statement;
          break;
  ...
  default: statement;
           break;
}
```

```
switch (exp) {
  case A:
    statement;
    break;
  case B:
    statement;
    break;
  ....
}
```

Recommended

Individual Statement layout

(a) Statement length

outdated rule : 80 characters max

nowadays : 90 characters usually

(b) Use spaces for clarity & readability
- spaces in logical expression
- spaces in array refrences
- space in parameters

(C) Formatting continuous lines

(i) → Make incomplete statements obvious

```
while (exp A) &&
(exp B) && (exp C) {
    ——
    ——
}
```

Not recommended

```
while ((exp A) &&
       (exp B) &&
       (exp C)) {
    ——
    ——
}
```

Recommended

(ii) → Keep closely related elements close

(iii) → Indent routine call continuation lines the standard amount.

(iv) → Make it easy to find the end of a continuation assignment statements.

(v) → Indent control-statement / assignment statement / continuation lines the standared amount.

(vi) Don't align right side of assignment

statements.

(d) Use one statement per line.

(e) Data Declaration

- Only one data declaration per line
- Declare variables close to where they are first use
- Order declaration sensibly.
- In C++, put ~~article~~ asterisk of pointer with variable name.

Comments layout

-> indent a comment with its corresponding code

->set off each comment with atleast a line

Routine layout

->Use blanks to seperate parts of routine

->Use standard indentation for routine arguments

Class layout

-> If a file has more than one class, identify each class clearly.

-> Put one class in one file.

-> File name should be related to class.

-> Seperate routines within a file clearly.

-> Sequence routine alphabetically

SCD 20-12

Book : SWEBOK → CH#6 Section 1

Configuration

↳ set up

Software Configuration Managment

→ Closest to SQA

→ Knows all the phases of software development life cycle (SDLC)

Configuration

Functional and physical charateristic of hardware and software mentioned in techical documentation or achieved in a product.

Planning for SCM

Software release Management

→ Identification Pachage and deliever a elgant product.

Release:

→ Executable program → Release Notes

→ Documentation → Configuration Data

Concerns:

→ When to issue a release

→ Product delivery items