SCD-15 (14/11/23)

Defensive Programming

Rule 1: You protect yourself (code) all the time.

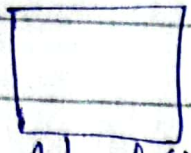
Rule 2: Never trust users.

Strategy

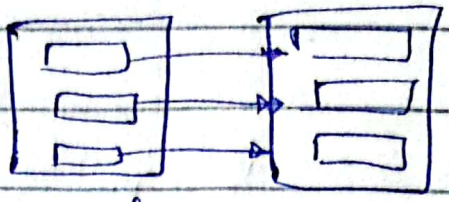
- Check the values of all data from external.
- Check the values of all routine input parameters.
- Decide how to handle bad data.

Solution

- Possible input.
  - GUI
  - CLI
  - Real-time
  - External process
  - other sources.
- Barricode ⇒ Class name that validates the input.
- Internal class
  - ↳ Student class - only performs its function don't validates the input.



Internal Class



Barricade GUI,

• Assertions

It is a logical formula inserted at same point in the program.

⇒ A type of cross-check.

① Forward reasoning.

② Backward reasoning.

↳ Checking by entering wrong input.

↳ Introduce bug & test cases.

Usage:

An input/output falls within expected range, A life is open/closed as expected.

- Pointer is not null.
- Array index out of bound.
- Objects/Arrays initialized properly.

A container is empty/full as expected verify pre-conditions/post-conditions.

Use assertion to verify the conditions that should never occur.

Avoid putting executable code into assertion. (Don't use .exe file or very big file that takes too much time)

Code Review

- Systematic Inspection of a software.
- Phase in between implementation, before testing.

Quality Assurance

1) Reviews → Personal Reviews.
   → Peer Review
   Management Review
2) Audit → SQA testing
   (Ready for deploy)

↳ complete SQA team test the application code.

SCD-16 (16/11/23)

Code Review

Systematic inspection of a software.

1- Peer Review

After you finish part of a program you explain your source code to another programmer.

- offline version of pair programming.
- common practice.

(Pair programming: 2 programmers work on a code simultaneously).

Advantage:

- collaboration makes a program better in quality & stability.
- catch most bugs.
- catch design flaws early.
- More than one person has seen the code.
- forces code author to articulate their decision and participate in the

discovery of flaws.
- Allow juniors to learn from seniors. experience without lowering the code quality.
- Accountability: authors and reviewers.
- Assessment of performance (Non-purpose).

⇒ Who should review?
1) Other developer.
2) Other dev from team.
Other dev or group of dev either from team or from outside.

⇒ Where should it conduct?
- In meeting
- On a decided place.

↳ The artifact should be shared before.

Fows
1) Error prone code.
2) previously discovered problem type.

3) Security.

4) Standard Checklist ⇒ Rule or style of writing the code in a company

Distinguish review types & audit types.

1) Purpose.
2) Level of independence.
3) Tools & techniques.
4) Roles.
5) Activity.

Audit Types.

1) Product assurance SQA team
2) Process assurance SQA team.

SCD-17 (21/11/23)

Management Reviews:

The main parameters of management reviews are project costs, schedule, scope and quality.

It evaluates decisions about
- corrective actions.
- changes in the allocation of resources.
- changes to the scope of project.

Personal Reviews.
- review his/her own code to enhance & ensure quality.
- ensure that your code is following standards of team/technology.
- review before peer review & management review.

SCD-18 (22/11/23)

- Banking systems were build on power builder are now being developed on ASP.Net Corr.

- **Refactoring:**
Improving a piece of software's internal structure without altering its external behaviour.

Each part of your code has 3 purposes.

1) Execute functionality.
2) Allow change & easy to maintain.
3) Communicate well to developers who read it.

2 Types of Refactoring:

1) Low level Refactoring.
2)

Low level Refactoring.

1) Naming:

Use descriptive names for variables & functions rather than dummy variables like a, b, a1, a2, etc.

- Avoid using 'Magic' constants.
  ↓
  anti-patter of using numbers for constant name.
- const double PI = 3.1514.

2) Procedure:

- Extract code into method.
- extract common function into method.
- Inlining an operation/procedure.
  - ↳ This technique expose significant optimization opportunities.

* Inlining:

compiler copies the code from function definition directly into the code of calling function rather than creating a separate set of instruction in a memory.

- - changing operation signatures (overloading).

3) Re-ordering:

- split one operation into several methods to improve cohesion & readability.
- put the semantically related statements near each other physically within your program.

SCD-19 (28/11/23)

Code Refactoring.

SCD-20 (29/11/23)

High level Refactoring

Significance:

- More important than low level refactoring.
- Improves overall structure of your problem.

Principles:

1- Exchange obscure language idioms with safer alternatives.

Example:

If you can write an "if" statement in one single line; Some other developers may not be familiar. Use coding style that has wide familarity & is good in term of readability.

Use switch, break, continue, return, instead of loop control variables.

Avoid loop control variables as much as possible.

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

3) Performance Optimization.

Process of modifying a software system to make it work more efficiently and execute more rapidly.

- Design level.
- Algorithm.
- Data Structure.
- Source Code.
- Build/Deployment.

God Class Features

- Difficult to read.
- Difficult to maintain.
- Encapsulate collection.

4) Refactor to design pattern.

5) Use polymorphism to replace condition.

6) Introduce enumeration.

7) Convert premitive type to a class.

Note

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

God Class

A class that try to do everything in the system.

SCD-21 (05/12/23)

Q: How to refactor a God Class?

Step 1:

Identify/categorize related attributes & operations from class diagram.

- put together related items.
- find natural home of operations in related class.

Step 2:

Remove all transient association (transitive association from DBMS) ↓ "property of any element in the system that is temporary."

- Associated class should be accessed through proper class rather than direct relation.

e.g: Library class associated with, catalogue associated with Item, Library should not be associated with Item.

Question: Add a new feature to code that is not well designed.

- Assume you have plenty of time (Although it is not true most of time).
- Write unit tests that verify external code's behaviour, correctness.
- Low level refactoring.
- High level refactoring.

3 Steps to add new feature

↳ After all of these only to add a new feature.

Benifits.

1) Costs 500% ROI when refactoring (Even if der or management do not want to do it).
- Time

2) Conductive to rapid dev.
3) Programming morale : well structured.
4) Programmers prefer to work in "clean-house"
   → Refering to code i.e., Programmers will prefer to work with well structured, well written code.

When to refactor?

Best Practice: Continuously as part of development process.

It is hard to refactor software late in the project.

Why? ⇒ Later in the project, a lot of features are added and changes affect layout huge part/features of software.

Reasons to Refactor?

1) Duplicated Code.
2) long routine (improve system by introducing modularity).
3) long & deep nested loops.
4) Poor cohesion/class has >1 responsibility.
5) Inconsistent level of abstraction.
6) Too many parameters.
7) Tight Coupling.
8) Related items are not organized.
9) A routine uses more features/attributes of the classes than its own attributes/features.

10) Inheritance hierarchies are not modified in parallel ⇒ One child doing more work.

11) Primitive data type is overloaded ⇒ e.g. currency (create class rather than int).

12) Global variables.

13) Improper / No comments.

14) Sub classes do not fully use parent class.

15) Public data structures.

16) Poor names.

17) Middle class / middle man isn't doing anything [Tramp data] e.g. only passing parameters / calling functions of other classes.

18) Passing data to other routines without modification / usage.

⇒ tramp class passing tramp data.

Refactoring:

(principles are separate (high/low))

1) Data level.
2) Statement level
3) Routine / function level.
4) Class implementation level.
5) Class interface level.
6) System.

Deployment:

- This stage occurs at the end of the active development of any piece of software.
- More of an event than a stage.
- current tech wave : Automated deployment
  1) Azure 2) Amazon
  ↓ Cloud Technologies ↙
- → Must have backup/recovery plan.
- → Deployment includes planned steps, problem areas, and plans to recover
- → Deployment should not be initiated without rollback plans.

Deployment Plan Concerns:

- Physical Environment.
- Hardware.
- Documentation.
- Training.
- DB related activities.
- 3rd party software.
- Software executable.

Deployment flows:

- Deliver software
- Revert on failure.

Rollback:

Reversal of actions completed during a deployment, with the intent to revert a system back to its previous working state.

Reasons for Rollback

⇒ (Determine the point of no return before deployment)

- Installation did not go as expected.
- Problems could take longer to fix than installation window.
- Keep production system alive.

SCD-22 (06/12/23)

Software Evolution:

Various experts have asserted that most of the cost of S/W ownership arise after delivering software i.e., at maintenance.

Types of maintenance:

1) Corrective : encompasses fixing bugs/features
2) Adaptive : Software adaptation to changing needs.
3) **Perfective:** This caters improved software in terms of performance.
4) Preventive : Deals with improved software by fixing bugs before they activate.

Manny Lehman "Father of S/W evolution"
S-type → Static E-type → Evolutionary.
[Real-world systems]

S/W evolution laws ⇒ On slides.
(All for e-type).

1) of continuous change ⇒ must change system with needs.
2) of increasing complexity ⇒ (more complex) = less efficient (must be extensible)
3) of self regulation ⇒ Distribution of products and processes (time for each release ≈ same).
(also size, no of reported errors)

4) of conservation of organizational stability ⇒ the average activity rate of e-type process tends to remain constant over lifetime/ segment of the lifetime.

⇒ The avg increment growth rate of e-type systems tends to remain constant over time or decline over time.

⇒ Mastery of sys decrease.

5) Conservation of familarity ⇒ All dev (also users) must grow if system grows.

6) Continued growth ⇒ must also maintain user satisfaction with s/w growth.

7) Declining quality ⇒ quality declines if appropriate amount of work not done.

8) Feedback Systems ⇒ Many sources, levels of feedback & they must be focused on.

Legacy:

Out dated computing software and/or hardware that is still in use.

Challenges:

1) Mission Critical.

2) Not equipped to deliver new services.

3) Do not upgrade at the speed and scale of user expectations.

ot, mobile,
d app.

Legacy systems cannot connect to current/new systems.

- Legacy system do not handle real-time data.
- Well designed API to communicate with legacy systems.