Software Construction and Development

Defensive programming

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

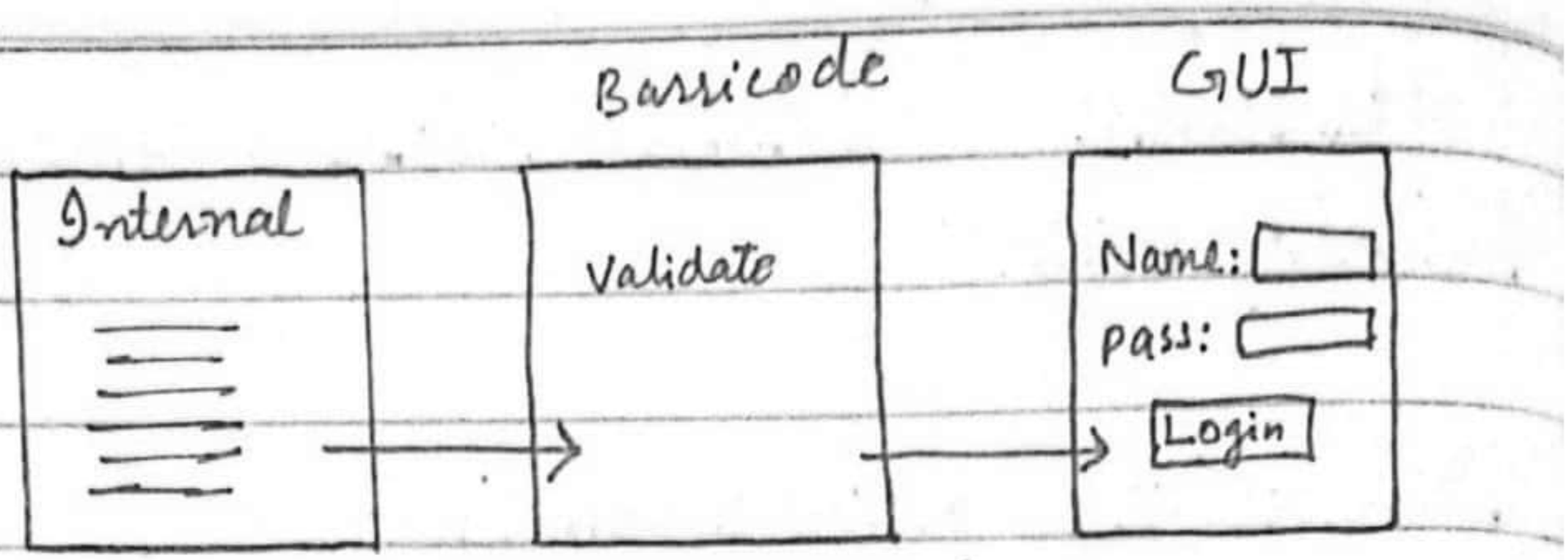
Rule 2: Never trust users

→ check the value of all data from external resource.

→ check all the values of routine input parameters.

Decide how to handle bad data.

possible inputs	Barricode validate class	Internal input class
GUI		
CLI		
Real time external process		expect values
others		



Note: Bassicode class check that the data is valid. if data is valid then it calls the Internal class. Its known as defensive programming.

Assertions (cross-check type)

It is a logical formula inserted at some point in the program. (we do testing)

Approches:

These are two approches in the assertion.

(i) Forward Reasoning — Data Driven

- intuitive (having the ability to know understand things without any proof)
- easily understandable

(iii) Disadvantage:- excess conditions

(ii) Backward Reasoning — Goal Driven

Use of Assertions:

i. An input / output falls within expected rang
ii. A file is open / closed as expected
iii. Assay index out of bound
iv. objects / Assays initiat
v. A containes is empty / full as expected
vi. verify pre-conditions / post-conditions
vii. Use assertions to verify the conditions that should never occur.
viii. Avoid putting executable code into assertion

Code Review

systematic Inspection (check) of a softwase

Quality Assusance

Reviews → personal Review
→ pees review
→ management Review

Audits → SQA testing team

What is peer programming?

Two person work on a project or task.

peer Review:-

After you finish past of a programming you represent / explain your source code

to another programmer.

→ offline version of pair programming

• mean's one person complete the code part and other person just check the mistakes.

→ common practice

Advantages:

i_ collaboration makes a program better quality and stability.

ii_ catch most bugs

iii_ catch design flaws easily

iv_ More than one person has seen the code.

v_ Forces code authors to articulate the decisions and participate in the discovery of flaws.

vi_ Allow juniors to learn from seniors experience without covering the code

vii_ Accountability authors and reviewers

viii_ Assesment of performance (Non-pure

Who should Review?

i_ other developer

ii_ other developes from team

iii- or group of other developers.

Focus in peer Reviews:-

i- Erros prone code

ii- previously discovered problem type

iii- security

iv- standard check list

Distinguish Reviews types / Audits type

i- purpose

ii- Level of independence

iii- tools techniques

iv- roles

v- Activity

Audits types:

i- product Assurance SQA

ii- process Assurance SQA

Management Review:

The main parameters of management reviews are project cost, schedule, scope (which features you are providing) and quality.

It evaluate decisions about:

→ corrective actions

→ changes in the allocation of resources

→ changes to the scope of the proj

Personal Review:

Review his/her own code to enh
and ensure quality.

Ensure that your code is fol
standards of team / technology.

Review before peer review, manag
review or audit.

Refactoring:

Improving a piece of softwar
internal structure without altering its
external behaviour.

Each part of your code ha
these parts or purpose.

- Execute functionality
- Allow change and easy to maint
(Not tight coupling)
- Communicate well to developer
who read it.

→ Commenting is important

→ developer's who are working with
or not should understand the code

Note: if nothing would happen you should change the code.

Types

(i) Low Level Refactoring:

i. Naming

→ use discriptive names for variables and function rathes than dummy variables.

Like: A, B, X, Y, Z,....

→ Avoid using magic constants

Magic: anti pattern of using numbers for constant name.

Example:

Const double X = 3.1514

Pi (π) used.

(ii) proceduse

- Extract code into method extract.
- common function into method
- Inlining an opesation / proceduse.
- changing operation signature (overloading)
- This technique expose significant optimization oppostunities.

Inlining: compiles copies the code from function definition directly into code

of calling function rather than creating a separate set of instru in a memory.

Re ordering / Re orders:

→ split one operation into several me
→ to improve cohesion and readability.
→ put the semantically related stat neas each other physically within you program.

High Level Refactoring

Significance:

More important than Low level refac toring.

. Improve overall structure of your pr

principles:

1. Exchange obscure Language idioms w safer alternatives.

e.g: If you can write an "if" statemen in one single line, some other develo may not be familias. Use coding st that has wide familiasity and is g in terms of seadability use switch, b continue, return statements.

1. Avoid loop controls 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:
   → Algorithms selection (optimal)
   → Data structure
   → Source code
   → Build / Deployment.
4. Refactor to design patterns
5. Use polymorphism to replace conditionals.

Note:

Compare to low level refactoring high level of refactoring is not well supported by tools

God class:

A class that try to do everything in the system
→ Difficult to read
→ Difficult to maintain

5. Use polymorphism to replace condi
6. Introduce emmeration
7. Convert primitive type of a clas
8. encapsulate collections.

How to refactor a god class?

→ Identify / categorize related attributes a operations

→ from class diagram

→ put togethes the related items

→ find natural home of operations in related class.

2 → Remove all transient association

Associated classes should be accessed through proper classes rather than direct relation.

Transient (Property of any element system that is temporary)

Assumptions:

Add a new feature to a co that is not well designed.

→ Assume you have a plenty o time.

→ write unit test that verify external code's behavior correctness

→ Low level Refactoring

→ High level Refactoring

→ Add new features

Cost of Refactoring

→ usually developer's don't want to refactor

→ Management don't want it

→ Time

Benefits of Refactoring

→ 500% ROI (Return Over Investment)

→ Code is more conductive to rapid development.

→ Programming mosale (well structured)

→

When to Refactor?

Best practice: continously as a past of the development process

It is hard to refactor your software Late in the project

Reason:

Later in the projects a lot features are added and changes affect/ impact huge part/ features the software.

Reasons to Refactor:

1. Duplicated Code
2. A long routine (improve system introducing modelarity.
3. Long & deep nested Loops
4. poor cohesion (a class has more th one respons.
5. Inconsistent level of abstraction
6. Too many parameters
7. Tight coupling
8. Related Items are not organized
9. A routine uses more features/attrib of other classes than it own attrib feature.
10. Inheritance heirarchies are not m in parallel.
11. primitive data type in overload

12. Global variables
13. Improper / No comments
14. Sub classes do not fully use the parent's class
15. public data members
16. poor names
17. Middle class / Middle man is not doing anything.

Tramp data

18. passing data to other routines without any usage / modification)

Levels of Refactoring

1. Data level refactoring
2. Statement level
3. Routine / Function
4. class Implementation
5. class Interface
6. System

Deployment: → Shouldn't occur with back plans.

→ this stage occurs at the end of active development of any piece of Software.

→ It is more of an event than st

→ current Technology wave: Automated dep ment cloud technologies.

1. Azure
2. Amazon

→ Must have a plan for backup & recovery.

→ Deployment include planned step. pr areas & plans to recover.

Deployment plan Concerns

→ physical Environment

→ Hardware

→ Documentation

→ Training

→ DB related activities

→ 3rd party software

→ Software executable

Deployment Focus

→ Deliever Software

→ Revest on Failure

Rollback:

Reversal of actions completed dusi during a development with the intent to revest a system back to its previous working state.

Reasons of Rollback

→ Determine your point of no return before deployment.

→ Installation does not go as expected

→ problems could take longer to fix than installation window.

→ Keep production system alive.

Software Evolution

Various experts have asserted that most of the cost of software owner ship arise after delivering software i.e: at maintenance.

Types:

Types of Software Maintenance:

1 - Corrective Maintenance:

This encompasses fixing bug features

2 - Adaptive Maintenance:

This include softwase adap to changing needs.

3 - Perfective Maintenance:

This cater's improved softw in terms of performance and ma ability.

4 - Perventive Maintenance:

This type of maintenance d with improved software by fixing before they activate.

Manny Lehman ⇒ Father of soft Evolution

S - Type = Static

E - Type = Evolutionary (Real world systems)

Law's of Software Evolution

1. Law of continuing change
2. Law of increasing complexity
3. Law of Self regulation

E-type system evalutionary process in self regulatory with distribution of product & process

parameters: Size, time b/w releases num. ber of reported

4. Law of conservation of organizational stability

→ The average incremental growth rate of e-type systems tends to remain cons. tant over time or decline over time

→ Mastery of the system decreases

5. Law of conservation of fasmilarity
6. Law of contining growth
7. Law of declining quality
8. Law of feedback system

Average activity sate in an E-type process tends to remain constant over system lifetime os segments of that lifetime.

Legacy Systems:

Outdated computing software and/or hardware that is still use.

Challenges:

i_ Mission critical

ii_ Not equipped to delieve m recvices.

iii_ Do not upgrade at the spe scale of users expectation.

Worst Case Scenario:

→ Legacy system can't connect t new systems

→ Legacy system does not supp. IOT, mobile, cloud application.

→ Real time data ko b handle krty.

→ For community with Legacy sy use/Write API.

Source code and Layout & Style

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand" Martin Fowler

Layout

→ It does not affect execution speed and memosy consumption

→ It affects how easy is it to understand the code, review and revision after months.

→ It also affects other developers readability, understanding and modification in your absence.

Fundamentals

1 - Logically organized
Proper use of White Space (indentation, new lines)

```
for(int i; i<size, i++)
  Statement A;
  Statement B;
  Statement C;
```

```
for(int i, i<size, i++){
    Statement A;
    Statement B;
    Statement C;
}
```

2. Consistent

Throughout the program use the same style.

Formatting results in:

→ Maintainable code

→ Improve Readability

Code Complete . Chapter 31 (self Read) . Layout & style

Fundamental theorm of Formatting:

Good visual layout shows the logical structure of a program

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

Techniques:-

proper use of whitespaces.

(a) Grouping

(b) Blank lines

(c) Indentation

proper use of pranthesis.

Style:

A lock of related / similar code use "login" and "end". It is class by looking at the code that particular lock of code state or end.

Control Structure Layout

(a) Avoid unintend login-end pairs.

e.g:

```
for (initial cond, final cond, each step)
{
    statement A;
    statement B;
}
```

(b) Avoid double indentation with login and end

e.g:

```
for(___,___,___){
    {
    }
}
```

(c) Use blank lines b/w pasagraphs (blocks of related code)

(d) format single statement blocks consistently.

e.g:

```
if (exp)
    statement A;

if (exp){
    statement A;
}

if (exp)
{
    statement A;
}

if (exp) statement A;
```

(e) for complicated expressions put separate expressions on separate lines.

```
if ( exp A &&
     exp B ||
     exp C ){
   statement A ;
   }
```

(f) Avoid Goto's (It makes program hard to format)

(g) No endline for case statements (expectional)

✗ not recommented

```
switch (exp){
caseA: statement;
       break;
caseB: statement;
       break;
caseC: statement;
       break;
default statements;
       breaks
}
```

✓ recommented

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

Individual Statements Layout

(a.) Statement length

outdated rule: 80 chasacter ma

now-a-days: 90 chàsacter usua

(b) Use spaces for clarity & read

_ spaces in logical expression

_ spaces in assay references

_ spaces in parameters

(c) Formatting continuation lines.

(i) → Make incomplete statements obvio

(not-recommented) recommented

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

(ii) keep closely related elements do

(iii) Indent routine call continu

lines the standard amount.

(iv) Make it easy to find the end

a continution assignment statements

(v) Indent contral-statements / assign

Continution line the statement sho

amount.

(i) Don't align right sides 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.
- ~~declare~~ Order declaration sensibly.
- In C++, put asterik * of pointers with variable name.

Comments Layout

→ indent a comment with its corresponding code.

→ set off each comment with at least a line.

Routines Layout

→ Use blanks to separate parts of routine

→ Use standard indentation for routine argument.

Classes Layout:

If a file has more than one
identify each class clearly
→ put one class in one file
→ File name should be related
class
→ Separate routines within a file
→ Sequence routine alphabetically.

SWEBOK chp 6 (Section 2)

Configuration ⇒ set up

Software Configuration Management
→ SQA k qareeb taseen ho.
→ knows all the phases of S
development life cycle

Configuration: Functional and phys
characteristic of hardware & softwa
mentioned in technical documentati
achieved in a product.
planning for SCM
Software Release Management
Identification package and deliver

elements of a product

Release:

→ Executable program

→ Documentation

→ Release Note

→ Configuration Data

Concerns:

→ When to issue a release

→ psoduct delivery items

→ Version release notes.

→ track distribution of product to customers

→ disgital verification