D10X : RULES OF SOFTWARE DESIGN AND CODING

August 2021

REMEMBER BAD DESIGN AND BAD CODE/ IMPLEMENTATION CAN KILL YOUR PROJECT/ START-UP

PREPARATION (STARTING POINT)

1

First Think of the solution(s) to the problem than the technology stack.

2

Understand and document who is your customer (i.e. who is paying for your application)

3

Understand and document who is user (i.e. who will be using your application)

4

Understand the scale (number of users, size of data, etc)

5

Think about how your application will be deployed.

- Think multiple possible alternatives.
- For each alternative, document 'why' it

CHOOSING A TECH STACK

1

Articulate why you chose a particular Tech Stack for specific project/ problem statement.

2

If your reason is **'everyone is using it'**, THEN You are not a Technical Architect

3

Avoid adapting to fashion and prefer reason.

Tech Stack Must include:

- Programming Language(s)
- Third party libraries and tools
- Cloud Hosting provider
- Deployment tools like docker, Kubernetes etc
- Communication protocols (gRPC, http/REST, GraphQL, etc etc)
- Databases (Relational, Object, Document, Graph Database, NoSQL etc etc)

CONSIDERATIONS FOR CHOOSING THE STACK

Team

- Team experience
- Their ability to learn and adapt
- Availability and cost of Trained manpower

Fitness for purpose

- Compiled vs interpretated
- Requirements of performance
- Requirements of portability
- Code Readability and maintainability

Maturity of technology

Cost (of licensing, hosting etc)

D10X

DESIGN FOR FAIL FAST

See the 'Rules of Coding' to Understand how to *Fail Fast* during coding.

Understand the concept of "Fail Fast"

Design your architecture, modules, code, tests and CI/CD pipeline to *Fail Fast*

TOP/DOWN DESIGN FOR ANY MEDIUM AND LARGE SCALE PROJECT

Break down your design *"top down"* in

- Processes to be deployed
- Modules/Packages/DLLs/ Libs inside the processes,
- Classes inside modules
- Member functions in a classes

Make sure that expected module dependencies are clearly defined.

Create a 'layer cake' diagram of module dependencies.

- If you are not able to define the 'layer cake' diagram, then you don't have a good design

DESIGN FOR OBSERVABILITY/ TELEMETRY FROM DAY 1

“Observability” enables you understand what is happening in production

Think about Telemetry in 2 categories:

- Troubleshooting
- Business Metrics/adoption (feature usage)

Do not rely on print statements for logging

Use a good logging framework from Day 1

- Typically Log4xx variant for your programming language stack
- Define Logging level

Architect to Integrate APM tools and crash logging tools

Enable 'crash' logging in production

- For Web, You should get an email if the application crashes in production. The mail must contain the call stack of the crash
- For Desktop, create a crash log file. Ask user to email the crash log to you.
- For Desktop, save the file and reopen it on restarting the application

USE OF THIRD PARTY PACKAGES/LIBRARIES

Be careful while using NPM or PyPI or NuGet while pulling a package. It may pull lot of dependencies along with package

- A single NPM package may pull in hundreds MBs of dependencies.

Carefully check licenses of 3rd Party dependencies

- For opensource, Prefer projects with Apache/Mozilla/BSD/MIT licenses.
- If you are developing commercial applications, avoid GPL licenses.

Check the history of opensource project before depending on it. It is possible that development is slowed or stopped.

Avoid 'freemium' packages and opensource products owned by companies. (e.g. MySQL, MongoDB)

Prefer opensource packages from foundations like Apache Foundation, Linux Foundation, Cloud Native Foundation etc.

MINIMIZE DEPENDENCIES

- Remove the dependencies to the extent possible when you are developing (use mock APIs, mock data)
- Target for loosely coupled systems (i.e. minimize tendencies)
- Make sure your dependencies are 'ACYCLIC'. (i.e. no circular dependencies)
- Cyclic dependencies are cause of major problems in software maintainability and negatively impact your project schedule
- Purpose of microservices is to keep dependencies low.

SCALE CHANGES EVERYTHING

⚠ BEWARE OF SCALE ⚠

- Your design must be appropriate for the expected scale in future and its implementation has to be appropriate for current scale. It is really hard to get this balance.
- Design Patterns are good solutions for large scale software design.
- Avoid design patterns for small projects
  - Especially Factories, IoC containers etc
  - Creating interfaces for everything.
- Do not create Interface/ Implementation segregation if you are going to create only one implementation.
- Minimize Mocks in testing.
- Beware of Order of Algorithms.
  - If you data size is in 100s for testing, your O(n2) algorithm will work great during testing but may bomb during productions of millions of data items.

CLASS HAS TO HAVE 'Responsibiliy'

- Document it.
- Document it in source code as well (as Class level comment)
- If you are not able to clearly describe the 'responsibility' of class, then you have design problem.
- Responsibility determines 'what operations' are to be defined on class. (i.e. Member functions).
- If class does not have any member methods and only 'set/get' methods , it is NOT a class.
- Using 'class' keyword does not make your design 'object oriented'

D10X

MINIMIZE CLASSES LIKE XXUTILS,XXMANAGER, XXCONTEXT, ETC

Usually classes named 'xxUtils' are grouping lot of static functions. It is essentially acting like a 'namespace' and not a class.

Usually it is hard to describe the responsibility of these kind classes

‘xxContext’ : Many a times become a ‘Global State’ that is passed around from top to bottom and can cause unexpected side effects

MINIMIZE SET/GET METHODS

- Do not auto generate set/get methods
- NEVER return a private collection (e.g. List) from a 'get' method
- NEVER set a private collection directly i.e. Replace existing collection object with another collection object. (Important for GC languages)
- If you want to allow iterating over a ‘private’ collection, provide a ‘get’ method that returns an ‘iterator’ over the collection
- More ‘set’ methods you have, more code duplication will be in your project.

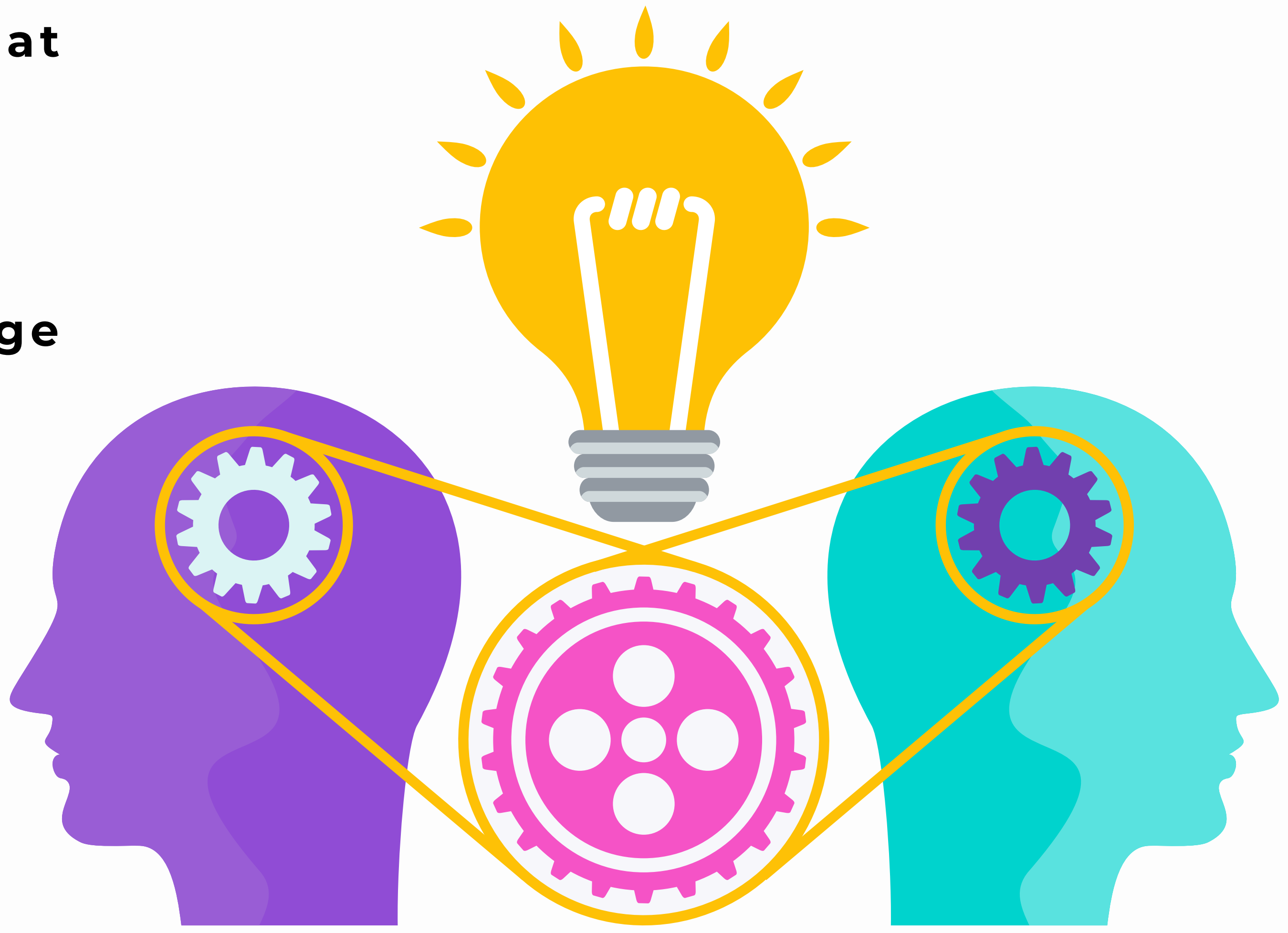
MEMORY MANAGEMENT FOR YOUR LANGUAGE

Understand how memory management works behind scenes for the programming language that you are using.

Memory management has huge impact on program performance

Understand new/delete

Understand malloc/free/calloc/realloc



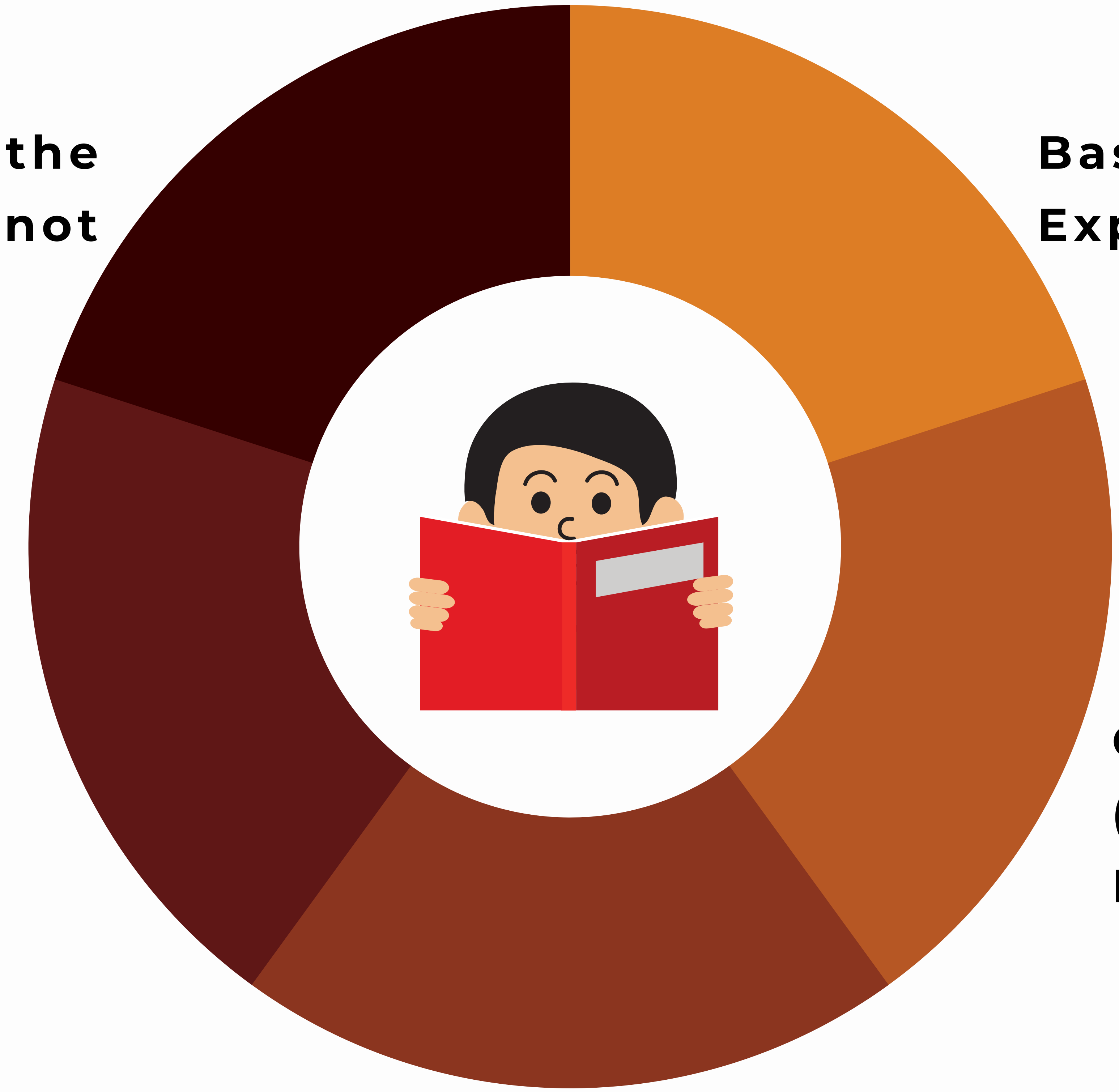
Understand how RAIC (Resource Allocation Is Construction) works for your language (e.g. IDisposable in .NET, using keywords, ‘with’ keyword in python)

Understand how GC (Garbage Collection) works for your language.

Understand smart pointers (auto_ptr, shared_ptr, unique_ptretc)

How variable 'scoping' works (local, global, module, static etc)

LEARN



DevOps – Understand the business benefits and not just 'buzzword'

Basics of UX (User Experience Design)

Design Thinking

Concepts of "internet" (TCP/IP, DNS, HTTP Requests/Response etc)

Concepts of CSS and Website design

D10X

RULES OF CODING

MOST IMPORTANT

- Read the ‘Rules of Design first' and follow it.
- Good Code is not a substitute for Bad Design.
- Quality is the fastest way to get your code into production.
- Own every line that you write.
  Do not copy/paste without understanding

WRITING FUNCTIONS

Max function size is 25 lines (including empty lines)

- Entire function must be visible without doing a Page Up/Down

Max 3 explicit parameters to function

If you are copy/pasting in the same file/class, you are lazy. Convert the copy/pasted code into a private method.

If you are copy/pasting across files/classes, you have a serious design issue.

NAMING FILES/CLASSES/FUNCTIONS

- Understand the vocab of domain. In application layer prefer domain based names instead of generic ones.
- All file names must be small case.
- File names must not contain any special characters. File name must not contain any white space characters.
- Name must match the responsibility
- Spend some time thinking the appropriate name for the class/function/variables. Don’t use the first name that pops in your mind.
- Use 'intention' revealing names.

USE ASSERTS

Remember Code Level Assert and Unit Test 'Asserts' are DIFFERENT.

Asserts are the way to implement FAIL FAST in code

Asserts improve the code readability.

Asserts improve the testability

Asserts improve the debuggability

Even mission critical code like "Mars Rover" uses asserts.

D10X

READABILITY AND UNDERSTANDABILITY

- Write code for *'ease of reading'* and not for *'ease of writing'*.
- Assume that the algorithm will change. Write a code in such a way that it is easy to change the algorithm later.
- Do not violate 'Law of Demeter'. Easier if you don't have set/get methods
- "Adapt" – adapt to existing coding and design style when working on large scale projects.
- Comments should describe things that are not obvious from code.
- Do not add redundant comments.

SIMPLIFY

S Simplify the code. Engineers/teams have a tendency for overengineering

R Relentlessly, focus to reduce the lines of code. Larger code is not only difficult to maintain but leads to more issues/defects.

M Make it work, Make it right, Refactor it to simplify.

P Prefer readymade library for low level code. For example, Do not write your own low level collections (e.g. use STL in C++)

ACKNOWLEDGEMENTS

THESE RULES ARE ADAPTED FROM
"THINKING CRAFTSMAN RULES OF SOFTWARE DEVELOPMENT"
BY NITIN BHIDE (CO-FOUNDER OF D10X)

Mayank Dikshit
Ashish Sarode

Thank you, for reviewing the draft
and suggesting changes and improvements.

NITIN@D10X.IO