## **autoProcess Implementation Process**

### Why?

### Code reviews are a necessary part of the software development process, designed to reduce technical debt and ensure consistency across your codebase. Everyone who writes code makes mistakes, but it is important to catch them before they reach production and start to cause damage.

### We make lot of promises during presales related to code quality standard we follow, but we always fail to deliver those promises due to lack of ownership, accountability, commitment, and lack of standard code quality process. Outlining a standardized system will help solve this.

Right now there is no standardized way to do code quality check and everyone follow their own coding standards. Due to that we always fail to deliver level of quality that we want to deliver.

Standard code quality process will help everyone follow same coding standards. This way every Team Lead/manager/mentor/Developer will maintain a quality standard for their projects.

Most importantly, lack of code quality standard can create bad impression and we end up losing good clients.

Examples where this can be used -

* Presales process so that sales people can understand what we follow and they can make promises accordingly.
* During development, so that developer can make sure that he follow defined code quality process.
* Mentors, so that they can validate code quality against defined standards.
* Team Lead, So that they can find out gaps in code quality standards and arrange training accordingly.
* Project manager, So that they can make sure that team follow defined coding standards.

### What are the goals and outcomes?

The goal here is to make sure that everyone follow same standard code quality process at Simform without applying their own standards.

### Code Review

Code Review is an integral process of software development that helps identify bugs and defects before the testing phase. Code review is often overlooked as an ongoing practice during the development phase, but countless studies show it's the most effective quality assurance strategy. Not having a set process in place means you don't actually know if your code reviews are effective or are even happening.-+

Regardless if it is your first or 100th time doing a code review, there may be certain questions that you need to ask yourself before actually doing it.

1. **Developer approval**: As developers it’s important to validate code quality yourself before generating pull request for code review. You need to make sure you followed defined coding standards so that reviewer don’t need to spend lot of time to point out silly mistakes. As reviewer you need to take conformation from developer that they reviewed code quality before generating pull request.
2. **Static Code Analysis:** We use <https://www.codacy.com> [Login with email : **developer@simform.com** and password : **Simform.321456**] to do static code analysis, developer is responsible to connect their repository with codacy and developer needs to make sure that they check codacy code review result(Style, code duplications, security). It’s expected that each file rated with A grade in codacy. Also developer need to put codacy configuration file(.codacy.yml) in root of the project as per this guideline. <https://support.codacy.com/hc/en-us/articles/360005097654-Ignore-files-from-Codacy-analysis> Check static code analysis result after each git push and fix if it find any issues.
3. **Dynamic code analysis:** Each and every code repository must follow git flow model. Only code reviewer/PM/TL are allowed to merge code in development branches. Developer always needs to generate git pull request if they want to merge code with development branch. Code reviewer needs to follow code quality checklist to review code quality and reject pull request if they find any issues.

1

### Code Quality Checklist

Preconditions

**P1: *Static code analysis***

Code must be verified by codacy and there should not be any open issues, code duplication, and style, security issues or any warnings. Make sure project has ‘A’ grade rating or its must be verified by appropriate linting tool.

**P2: Read me file**

Read me file must be available in root of the code repository (Need to add link for read me template).

**P3: libraries/framework verification**

There should not any deprecated library used within project and package manager should not reference any specific version of library. Always chose library who has high number of stars in github.

**P4: Warnings**

There should not be any warnings in the project before generating PR.

**P5: Environments**

Environment file should be available within project to target different environment in the project, it should contain list of all credentials required to execute project in particular environment. Project should not contain any static variables anywhere in the project.

**P6: Code Push frequency**

Code must be pushed to code repo without fail every day and must be committed multiple times within a day.

**P7: Logging and Monitoring**

<https://aws.amazon.com/xray/>, CloudWatch, Crashlytics Integration, [https://sentry.io](https://sentry.io/)

**P8: Do not allow any generated code in repository**

Code repository should not contact any generated code like build folder, package folder(like node modules, pods, frameworks etc).

General

**G1: *Multiple Languages in One Source File***

Today’s modern programming environments make it possible to put many different languages into a single source file. For example, a Java source file might contain snippets of XML, HTML, YAML, Javadoc, English, JavaScript, and so on. For another example, in addition to HTML a JSP file might contain Java, a tag library syntax, English comments, Javadocs, XML, JavaScript, and so forth. This is confusing at best and carelessly sloppy at worst.

The ideal is for a source file to contain one, and only one, language. Realistically, we will probably have to use more than one. But we should take pains to minimize both the number and extent of extra languages in our source files.

**G2: *Obvious Behavior Is Unimplemented***

Following “The Principle of Least Surprise”any function or class should implement the behaviors that another programmer could reasonably expect. For example, consider a function that translates the name of a day to an Enum that represents the day.

Day day = DayDate.StringToDay(String dayName);

We would expect the string "Monday" to be translated to Day.MONDAY. We would also expect the common abbreviations to be translated, and we would expect the function to ignore case.

When an obvious behavior is not implemented, readers and users of the code can no longer depend on their intuition about function names. They lose their trust in the original author and must fall back on reading the details of the code.

**G3: *Incorrect Behavior at the Boundaries***

It seems obvious to say that code should behave correctly. The problem is that we seldom realize just how complicated correct behavior is. Developers often write functions that they think will work, and then trust their intuition rather than going to the effort to prove that their code works in all the corner and boundary cases.

There is no replacement for due diligence. Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm. *Don’t rely on your intuition*. Look for every boundary condition and write a test for it.

**G4: *Overridden Safeties***

Chernobyl melted down because the plant manager overrode each of the safety mechanisms one by one. The safeties were making it inconvenient to run an experiment. The result was that the experiment did not get run, and the world saw it’s first major civilian nuclear catastrophe.

It is risky to override safeties. Exerting manual control over serialVersionUID may be necessary, but it is always risky. Turning off certain compiler warnings (or all warnings!) may help you get the build to succeed, but at the risk of endless debugging sessions. Turning off failing tests and telling yourself you’ll get them to pass later is as bad as pretending your credit cards are free

money.

**G5: *Duplication***

This is one of the most important rules in code review, and you should take it very seriously.

Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright. By folding the duplication into such an abstraction, you increase the vocabulary of the language of your design.

The most obvious form of duplication is when you have clumps of identical code that look like some programmers went wild with the mouse, pasting the same code over and over again. These should be replaced with simple methods.

A more subtle form is the switch/case or if/else chain that appears again and again in various modules, always testing for the same set of conditions. These should be replaced with polymorphism.

I think the point has been made. Find and eliminate duplication wherever you can.

**G6: *Code at Wrong Level of Abstraction***

It is important to create abstractions that separate higher level general concepts from lower level detailed concepts. Sometimes we do this by creating abstract classes to hold the higher level concepts and derivatives to hold the lower level concepts. When we do this, we need to make sure that the separation is complete. We want *all* the lower level concepts to be in the derivatives and *all* the higher level concepts to be in the base class.

For example, constants, variables, or utility functions that pertain on ly to the detailed implementation should not be present in the base class. The base class should know nothing about them.

This rule also pertains to source files, components, and modules. Good software design requires that we separate concepts at different levels and place them in different containers. Sometimes these containers are base classes or derivatives and sometimes they are source files, modules, or components. Whatever the case may be, the separation needs to be complete. We don’t want lower and higher level concepts mixed together. Consider the following code:

public interface Stack {

Object pop() throws EmptyException;

void push(Object o) throws FullException;

double percentFull();

class EmptyException extends Exception {}

class FullException extends Exception {}

}

The percentFull function is at the wrong level of abstraction. Although there are many implementations of Stack where the concept of *fullness* is reasonable, there are other implementations that simply *could not know* how full they are. So the function would be better placed in a derivative interface such as BoundedStack.

Perhaps you are thinking that the implementation could just return zero if the stack were boundless. The problem with that is that no stack is truly boundless. You cannot really prevent an OutOfMemoryException by checking for stack.percentFull() < 50.0. Implementing the function to return 0 would be telling a lie.

The point is that you cannot lie or fake your way out of a misplaced abstraction. Isolating abstractions is one of the hardest things that software developers do, and there is no quick fix when you get it wrong.

**G7: *Too Much Information***

Well-defined modules have very small interfaces that allow you to do a lot with a little. Poorly defined modules have wide and deep interfaces that force you to use many different gestures to get simple things done. A well-defined interface does not offer very many functions to depend upon, so coupling is low. A poorly defined interface provides lots of functions that you must call, so coupling is high.

Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better.

Hide your data. Hide your utility functions. Hide your constants and your temporaries. Don’t create classes with lots of methods or lots of instance variables. Don’t create lots of protected variables and functions for your subclasses. Concentrate on keeping interfaces very tight and very small. Help keep coupling low by limiting information.

**G8: *Dead Code***

Dead code is code that isn’t executed. You find it in the body of an if statement that checks for a condition that can’t happen. You find it in the catch block of a try that never throws. You find it in little utility methods that are never called or switch/case conditions that never occur.

The problem with dead code is that after a while it starts to smell. The older it is, the stronger and sourer the odor becomes. This is because dead code is not completely updated when designs change. It still *compiles*, but it does not follow newer conventions or rules. It was written at a time when the system was *different*. When you find dead code, do the right thing. Give it a decent burial. Delete it from the system.

**G9: *Vertical Separation***

Variables and function should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope. We don’t want local variables declared hundreds of lines distant from their usages.

Private functions should be defined just below their first usage. Private functions belong to the scope of the whole class, but we’d still like to limit the vertical distance between the invocations and definitions. Finding a private function should just be a matter of scanning downward from the first usage.

**G10: *Inconsistency***

If you do something a certain way, do all similar things in the same way. This goes back to the principle of least surprise. Be careful with the conventions you choose, and once chosen, be careful to continue to follow them.

If within a particular function you use a variable named response to hold an HttpServletResponse, then use the same variable name consistently in the other functions that use HttpServletResponse objects. If you name a method processVerificationRequest, then use a similar name, such as processDeletionRequest, for the methods that process other kinds of requests.

Simple consistency like this, when reliably applied, can make code much easier to read and modify.

**G11: *Clutter***

Of what use is a default constructor with no implementation? All it serves to do is clutter up the code with meaningless artifacts. Variables that aren’t used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source files clean, well organized, and free of clutter.

**G12: *Artificial Coupling***

Things that don’t depend upon each other should not be artificially coupled. For example, general enums should not be contained within more specific classes because this forces the whole application to know about these more specific classes. The same goes for general purpose static functions being declared in specific classes.

In general an artificial coupling is a coupling between two modules that serves no direct purpose. It is a result of putting a variable, constant, or function in a temporarily convenient, though inappropriate, location. This is lazy and careless.

Take the time to figure out where functions, constants, and variables ought to be declared. Don’t just toss them in the most convenient place at hand and then leave them there.

**G13: *Obscured Intent***

We want code to be as expressive as possible. Run-on expressions, Hungarian notation, and magic numbers all obscure the author’s intent. For example, here is the overTimePay function as it might have appeared:

public int m\_otCalc() { return iThsWkd \* iThsRte +

(int) Math.round(0.5 \* iThsRte \*

Math.max(0, iThsWkd - 400)

);

}

Small and dense as this might appear, it’s also virtually impenetrable. It is worth taking the time to make the intent of our code visible to our readers.

**G14: *Misplaced Responsibility***

One of the most important decisions a software developer can make is where to put code. For example, where should the PI constant go? Should it be in the Math class? Perhaps it belongs in the Trigonometry class? Or maybe in the Circle class?

The principle of least surprise comes into play here. Code should be placed where a reader would naturally expect it to be. The PI constant should go where the trig functions are declared. The OVERTIME\_RATE constant should be declared in the HourlyPayCalculator class.

Sometimes we get “clever” about where to put certain functionality. We’ll put it in a function that’s convenient for us, but not necessarily intuitive to the reader. For example, perhaps we need to print a report with the total of hours that an employee worked. We could sum up those hours in the code that prints the report, or we could try to keep a running total in the code that accepts time cards.

One way to make this decision is to look at the names of the functions. Let’s say that our report module has a function named getTotalHours. Let’s also say that the module that accepts time cards has a saveTimeCard function. Which of these two functions, by it’s name, implies that it calculates the total? The answer should be obvious.

Clearly, there are sometimes performance reasons why the total should be calculated as time cards are accepted rather than when the report is printed. That’s fine, but the names of the functions ought to reflect this. For example, there should be a computeRunningTotalOfHours function in the timecard module.

**G15: *Inappropriate Static***

Math.max(double a,double b) is a good static method. It does not operate on a single instance; indeed, it would be silly to have to say newMath().max(a,b) or even a.max(b). All the data that max uses comes from its two arguments, and not from any “owning” object. More to the point, there is almost *no chance* that we’d want Math.max to be polymorphic.

Sometimes, however, we write static functions that should not be static. For example, consider:

HourlyPayCalculator.calculatePay(employee, overtimeRate).

Again, this seems like a reasonable static function. It doesn’t operate on any particular object and gets all it’s data from it’s arguments. However, there is a reasonable chance that we’ll want this function to be polymorphic. We may wish to implement several different algorithms for calculating hourly pay, for example, OvertimeHourlyPayCalculator and StraightTimeHourlyPayCalculator. So in this case the function should not be static. It should be a nonstatic member function of Employee.

In general you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you’ll want it to behave polymorphically.

**G16: *Use Explanatory Variables***

One of the more powerful ways to make a program readable is to break the calculations up into intermediate values that are held in variables with meaningful names.

Consider this example from FitNesse:

Matcher match = headerPattern.matcher(line); if(match.find())

{

String key = match.group(1); String value = match.group(2); headers.put(key.toLowerCase(), value);

}

The simple use of explanatory variables makes it clear that the first matched group is the *key,* and the second matched group is the *value*.

It is hard to overdo this. More explanatory variables are generally better than fewer. It is remarkable how an opaque module can suddenly become transparent simply by breaking the calculations up into well-named intermediate values.

**G17: *Function Names Should Say What They Do***

Look at this code:

Date newDate = date.add(5);

Would you expect this to add five days to the date? Or is it weeks, or hours? Is the date instance changed or does the function just return a new Date without changing the old one? *You can’t tell from the call what the function does*.

If the function adds five days to the date and changes the date, then it should be called addDaysTo or increaseByDays. If, on the other hand, the function returns a new date that is five days later but does not change the date instance, it should be called daysLater or daysSince.

If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

**G18: *Understand the Algorithm***

Lots of very funny code is written because people don’t take the time to understand the algorithm. They get something to work by plugging in enough if statements and flags, without really stopping to consider what is really going on.

Programming is often an exploration. You *think* you know the right algorithm for something, but then you wind up fiddling with it, prodding and poking at it, until you get it to “work.” How do you know it “works”? Because it passes the test cases you can think of.

There is nothing wrong with this approach. Indeed, often it is the only way to get a function to do what you think it should. However, it is not sufficient to leave the quotation marks around the word “work.”

Before you consider yourself to be done with a function, make sure you *understand* how it works. It is not good enough that it passes all the tests.

Often the best way to gain this knowledge and understanding is to refactor the function into something that is so clean and expressive that it is *obvious* how it works.

**G19: *Prefer Polymorphism to If/Else or Switch/Case***

This might seem a strange suggestion given the topic of Chapter 6. After all, in that chapter I make the point that switch statements are probably appropriate in the parts of the system where adding new functions is more likely than adding new types.

First, most people use switch statements because it’s the obvious brute force solution, not because it’s the right solution for the situation. So this heuristic is here to remind us to consider polymorphism before using a switch.

Second, the cases where functions are more volatile than types are relatively rare. So *every* switch statement should be suspect.

Use “ONE SWITCH” rule: *There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.*

**G20: *Follow Standard Conventions***

Every team should follow a coding standard based on common industry norms. This coding standard should specify things like where to declare instance variables; how to name classes, methods, and variables; where to put braces; and so on. The team should not need a document to describe these conventions because their code provides the examples.

Everyone on the team should follow these conventions. This means that each team member must be mature enough to realize that it doesn’t matter a whit where you put your braces so long as you all agree on where to put them.

Find standard conventions in department bible document.

**G21: *Replace Magic Numbers with Named Constants***

This is probably one of the oldest rules in software development. In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.

For example, the number 86,400 should be hidden behind the constant

SECONDS\_PER\_DAY. If you are printing 55 lines per page, then the constant 55 should be hidden behind the constant LINES\_PER\_PAGE.

**G22: *Be Precise***

Expecting the first match to be the *only* match to a query is probably naive. Using floating point numbers to represent currency is almost criminal. Avoiding locks and/or transaction management because you don’t think concurrent update is likely is lazy at best. Declaring a variable to be an ArrayList when a List will due is overly constraining. Making all variables protected by default is not constraining enough.

When you make a decision in your code, make sure you make it *precisely*. Know why you have made it and how you will deal with any exceptions. Don’t be lazy about the precision of your decisions. If you decide to call a function that might return null, make sure you check for null. If you query for what you think is the only record in the database, make sure your code checks to be sure there aren’t others. If you need to deal with currency, use integersand deal with rounding appropriately. If there is the possibility of concurrent update, make sure you implement some kind of locking mechanism.

Ambiguities and imprecision in code are either a result of disagreements or laziness.

In either case they should be eliminated.

**G23: *Structure over Convention***

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance. For example, switch/cases with nicely named enumerations are inferior to base classes with abstract methods. No one is forced to implement the switch/case statement the same way each time; but the base classes do enforce that concrete classes have all abstract methods implemented.

**G24: *Encapsulate Conditionals***

Boolean logic is hard enough to understand without having to see it in the context of an if or while statement. Extract functions that explain the intent of the conditional.

For example: if (shouldBeDeleted(timer))

is preferable to if (timer.hasExpired() && !timer.isRecurrent())

**G25: *Avoid Negative Conditionals***

Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives. For example:

if (buffer.shouldCompact())

is preferable to if (!buffer.shouldNotCompact())

**G26: *Functions Should Do One Thing***

It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than *one thing*, and should be converted into many smaller functions, each of which does *one thing*.

For example:

public void pay() { for (Employee e : employees) { if (e.isPayday()) {

Money pay = e.calculatePay();

e.deliverPay(pay);

}

}

}

This bit of code does three things. It loops over all the employees, checks to see whether each employee ought to be paid, and then pays the employee.

This code would be better written as:

public void pay()

{

for (Employee e : employees)

payIfNecessary(e);

}

private void payIfNecessary(Employee e) { if (e.isPayday()) calculateAndDeliverPay(e); }

private void calculateAndDeliverPay(Employee e) {

Money pay = e.calculatePay();

e.deliverPay(pay); }

Each of these functions does one thing.

**G27: *Encapsulate Boundary Conditions***

Boundary conditions are hard to keep track of. Put the processing for them in one place. Don’t let them leak all over the code. We don’t want swarms of +1s and -1s scattered hither and yon. Consider this simple example from FIT:

if(level + 1 < tags.length)

{

parts = new Parse(body, tags, level + 1, offset + endTag); body = null;

}

Notice that level+1 appears twice. This is a boundary condition that should be encapsulated within a variable named something like nextLevel.

int nextLevel = level + 1;

if(nextLevel < tags.length)

{

parts = new Parse(body, tags, nextLevel, offset + endTag); body = null;

}

**G28: *Keep Configurable Data at High Levels***

If you have a constant such as a default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function.

The configuration constants reside at a very high level and are easy to change. They get passed down to the rest of the application. The lower levels of the application do not own the values of these constants.

Functions

**F1: *Too Many Arguments***

Functions should have a small number of arguments. No argument is best, followed by one, two, and three. More than three is very questionable and should be avoided with prejudice.

**F2: *Output Arguments***

Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on.

**F3: *Flag Arguments***

Boolean arguments loudly declare that the function does more than one thing. They are confusing and should be eliminated.

**F4: *Dead Function***

Methods that are never called should be discarded. Keeping dead code around is wasteful. Don’t be afraid to delete the function. Remember, your source code control system still remembers it.

Comments

**C1: *Inappropriate Information***

It is inappropriate for a comment to hold information better held in a different kind of system such as your source code control system, your issue tracking system, or any other record-keeping system. Change histories, for example, just clutter up source files with volumes of historical and uninteresting text. In general, meta-data such as authors, lastmodified-date, SPR number, and so on should not appear in comments. Comments should be reserved for technical notes about the code and design.

**C2: *Obsolete Comment***

A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete. If you find an obsolete comment, it is best to update it or get rid of it as quickly as possible. Obsolete comments tend to migrate away from the code they once described. They become floating islands of irrelevance and misdirection in the code.

**C3: *Redundant Comment***

A comment is redundant if it describes something that adequately describes itself.

For example: i++; // increment i

Comments should say things that the code cannot say for itself.

**C4: *Poorly Written Comment***

A comment worth writing is worth writing well. If you are going to write a comment, take the time to make sure it is the best comment you can write. Choose your words carefully. Use correct grammar and punctuation. Don’t ramble. Don’t state the obvious. Be brief.

**C5: *Commented-Out Code***

It makes me crazy to see stretches of code that are commented out. Who knows how old it is? Who knows whether or not it’s meaningful? Yet no one will delete it because everyone assumes someone else needs it or has plans for it.

That code sits there and rots, getting less and less relevant with every passing day. It calls functions that no longer exist. It uses variables whose names have changed. It follows conventions that are long obsolete. It pollutes the modules that contain it and distracts the people who try to read it. Commented-out code is an *abomination*.

When you see commented-out code, *delete it!* Don’t worry, the source code control system still remembers it. If anyone really needs it, he or she can go back and check out a previous version. Don’t suffer commented-out code to survive.

Environment

**E1: *Build Requires More Than One Step***

Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control. You should not need a sequence of arcane commands or context dependent scripts in order to build the individual elements. You should not have to search near and far for all the various little extra JARs, XML files, and other artifacts that the system requires. You *should* be able to check out the system with one simple command and then issue one other simple command to build it.

**E2: *Tests Require More Than One Step***

You should be able to run *all* the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE. In the worst case you should be able to issue a single simple command in a shell. Being able to run all the tests is so fundamental and so important that it should be quick, easy, and obvious to do.

Tests

**T1: Insufficient Tests**

How many tests should be in a test suite? Unfortunately, the metric many programmers use

is “That seems like enough.” A test suite should test everything that could possibly break.

The tests are insufficient so long as there are conditions that have not been explored by the

tests or calculations that have not been validated.

**T2: Use a Coverage Tool!**

Coverage tools reports gaps in your testing strategy. They make it easy to find modules,

classes, and functions that are insufficiently tested. Most IDEs give you a visual indication,

marking lines that are covered in green and those that are uncovered in red. This makes it

quick and easy to find if or catch statements whose bodies haven’t been checked.

**T3: Don’t Skip Trivial Tests**

They are easy to write and their documentary value is higher than the cost to produce

them.

**T4: An Ignored Test Is a Question about an Ambiguity**

Sometimes we are uncertain about a behavioral detail because the requirements are

unclear. We can express our question about the requirements as a test that is commented

out, or as a test that annotated with @Ignore. Which you choose depends upon whether the

ambiguity is about something that would compile or not.

**T5: Test Boundary Conditions**

Take special care to test boundary conditions. We often get the middle of an algorithm

right but misjudge the boundaries.

**T6: Exhaustively Test Near Bugs**

Bugs tend to congregate. When you find a bug in a function, it is wise to do an exhaustive

test of that function. You’ll probably find that the bug was not alone.

**7: Patterns of Failure Are Revealing**

Sometimes you can diagnose a problem by finding patterns in the way the test cases fail.

This is another argument for making the test cases as complete as possible. Complete test

cases, ordered in a reasonable way, expose patterns.

As a simple example, suppose you noticed that all tests with an input larger than five

characters failed? Or what if any test that passed a negative number into the second argument

of a function failed? Sometimes just seeing the pattern of red and green on the test

report is enough to spark the “Aha!” that leads to the solution

**T8: Test Coverage Patterns Can Be Revealing**

Looking at the code that is or is not executed by the passing tests gives clues to why the

failing tests fail.

**T9: Tests Should Be Fast**

A slow test is a test that won’t get run. When things get tight, it’s the slow tests that will be dropped from the suite.

### Architecture Review

Architecture reviews are an effective way of ensuring design quality and addressing architectural concerns. We need to perform architecture review three times during life cycle of the project.

1. Before staring development
2. After stage deployment
3. After production deployment

We need to review following critical things during architecture review:

**A1: Performance**

The time required to respond/UI lag/Long running processes/queries, or the number of events processed in some interval of time.

Things to verify:

1. Caching – Using a local copy of data to reduce access time for mobile apps. Use cache server to store frequently used data in-memory for backends.
2. On demand data loading.
3. Identify long running thread/processes/queries.
4. Reduce for loops as much as possible
5. Load static content (JS/CSS/fonts etc) from CDN. And have a way to version them so that previously cached files aren’t served with new build/version.
6. Check memory and battery usage for mobile applications.
7. Deployment - Adequate size of VM, Load Balancing, separate application and database VM, usage of Media storage service and CDN etc are basic and must for most project.

**A2: Reliability**

The ability of the system to keep operating over time in the context of application and system errors and in situations of unexpected or incorrect usage.

Things to verify:

1. Microservices architecture must be followed for APIs.
2. All deployment and third party services/component/API related configuration (URLs, Keys, IPs etc) should be well organized and managed for stage/production deployment.
3. Containerization of applications using Kubernetes and docker. Containers can be defined for different components of the application (Ie. Backend, Caching server, Queue processor) and deployed seamlessly using CI/CD on cloud services which supports containers (Ie. Beanstalk).

**A3: Security**

Measure of the system's ability to resist unauthorized attempts at usage and denial of service.

Things to check:

1. Authorisation - How is the control access to information in the system organized once a user is identified and authenticated?
2. Authentication - Identification and representation of end-user in your system and verification if he is telling the truth?
3. OWASP- Follow as much OWASP guideline and checklist as possible. - <https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf>
4. Audits – Do full security audit for important projects at the time of beta testing.
5. Deployment access – Follow coud best practices for securing access to cloud deployment. Ie. For AWS always create separate user/identity in IAM for every service (Ie. Separate user for production S3 and stage S3, Separate user for Beanstalk, separate for VM, separate user for RDS). Never use master identity within app to access all services.

**A4: Modifiability**

Ability to make changes to a system quickly and cost effectively.

Things to verify:

1. Independence of interface from implementation – API should be developed by looking at the user interface and APIs should generic as much as possible.
2. Separation of concern – This strategy separates data and function that address different concerns. Since the concerns are separate, we can modify one concern independently of another. Isolating common function is another example of a separation strategy. Better use of database views, functions can be helpful to separate logic.
3. Configurability – Keep as much variable as possible in a configuration file/db, which can be changed at runtime without requiring redeployment. Avoid hard-coded setting and names (connection strings, queue names, URLs etc).

**A5: Usability**

How easy is it to use the program?

Things to verify:

1. Designer usability audit: Have UI/UX team do proper usability audit to assess the implementation of the UI design original given by them.

**A6: Ease of Deployment**

How quickly can you deploy the system?

Things to verify:

1. CI/CD - Automate deployment for faster, error-free and fault-tolerant deployment with quicker rollback ability Ability to roll back to previous version without any conflict (db etc) is a must. Manual deployment configuration errors (Wrong ID, db name, URLs, Credentials) should not happen at all..

**A7: Scalability**

Support continuous growth to meet user demand and business complexity. It must be possible to extend the minimum hardware configuration needed for the application with additional hardware to support increased workloads.

Things to verify:

1. Database replication – If database is being bottleneck, use multi database and replication.
2. Serverless – Use serverless architecture for some of the high traffic APIs or services
3. Load balancing – Use multiple instances for application code. Such deployment must be containerized.

**A8: Debug-ability /Monitoring**

Preparing application for easy and efficient debugging. Registration of abnormal behavior. Real-time monitoring.

Things to verify:

1. Logging – Always have a proper functional/logical logging and exception/error logging. So that when issues occur in production you have logs data to analyze and troubleshoot rather than wasting days in reproducing similar situation.
2. Monitoring – Enable deployment resources monitoring.
3. application performance management

Tools:

<https://aws.amazon.com/xray/>, CloudWatch, Crashlytics Integration, [https://sentry.io](https://sentry.io/)

**A9: Development Productivity**

Cost and time saving mechanism to aid development of applications based on the software architecture. The developers should be able to learn the architecture concept and how to implement it easily. Extending the development team with new developers should not cost much effort in instruction, etc. A standardized way of working using templates and coding standards could help raise both the learning curve and quality.

Things to verify:

1. Framework needs to be followed as per bible document.
2. Templates/code generation should follow as per standard department code template.
3. Best practices Coding checklist & standards should follow as per bible document of department.

References

1. <https://www.investigatii.md/uploads/resurse/Clean_Code.pdf>
2. <http://ptgmedia.pearsoncmg.com/images/9780137081073/samplepages/0137081073.pdf>
   1. <https://refactoring.guru/>
3. <https://packagequality.com/>