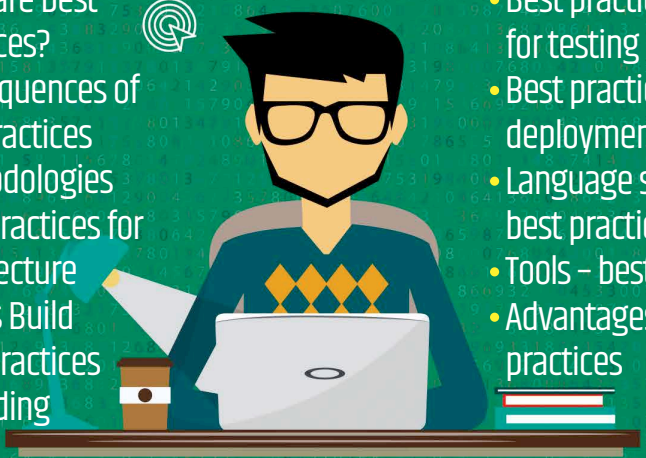
VOLUME 12 | ISSUE 07 | JULY 2017

digit FastTrack
YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

To Coding Better

- What are best practices?
- Consequences of bad practices
- Methodologies
- Best practices for architecture
- Buy vs Build
- Best practices for coding
- Best practices for testing
- Best practices for deployment
- Language specific best practices
- Tools – best practices
- Advantages of best practices



A 9 9 Media Publication

Free With Digit July 2017

www.digit.in/forum

Join the forum to express your views and resolve your differences in a more civilised way.

digit.in FORUM

Post your queries and get instant answers to all your technology related questions

One of the most active **online technology forums** not only in India but world-wide

JOIN NOW

digit.in

CODING BETTER

powered by

digit

YOUR TECHNOLOGY NAVIGATOR

CHAPTERS

CODING BETTER
JULY 2017

CREDITS
The people behind this book

EDITORIAL
Executive Editor
Robert Sovereign-Smith

Managing Editor
Siddharth Parwatay

Assistant Technical Editor
Mithun Mohandas

Copy editing
Abhijit Dey
Arnab Mukherjee

Writers
Arunava Roy Choudhury
Bhoamik Garg
Poulami Dey
Rohan Shrivastava

DESIGN
Sr. Art Director
Anil VK

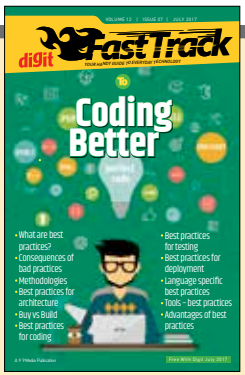
Visualiser
Baiju NV

digit.in

July 2017
Free with Digit. If you have paid to buy this Fast Track from any source other than 9.9 Mediaworx Pvt. Ltd., please write to editor@digit.in with details

Custom publishing
If you want us to create a customised Fast Track for you in order to demystify technology for your community, employees or students contact editor@digit.in

e-mail



COVER DESIGN: PETERSON PJ

Unleash the better coder in you

If you believe in this mantra and want to write code in any language in the most optimal manner - then this book is perfect for you. There's a lot more to coding than just button mashing (albeit, intelligent, specialised button mashing) to get things done. You might be new to programming or have just ventured beyond what's taught in classrooms. In that case, shed your perceptions, developed over years under the infleunce of popular culture, that writing good code is all about spending hours at a terminal simply splurging out line after line of Java, PHP or whichever langauge you chose.

And in case you are an experienced developer yet inexperienced in the concept of best practices of software development, it is a wonder that you've still survived in the world of programming. Unless you're working as a lone wolf, it is of paramount importance to follow certain guidelines at each and every stage of software development right from the inception of the idea to the final product itself.

To begin with, you can't just go out there and say, for instance, "I want a travel website" or "I want a translation software". You have to specify (or understand) exactly what your project needs to achieve, in quantifiable, discreet terms that, in turn, can be explained to other developers and testers in your team.

There are methodologies that you can use to your advantage and even they come with their own set of best practices. And when it comes down to the design, a simple bad architectural choice could throw the whole project into jeopardy down the line. For instance, did you know that a 46 ¢ computer chip had almost caused the third world war?

One of the most important concepts, when it comes to best practices of software development, is that they don't end with software development. Testing, or the stage which developers have the strongest love/hate relationship with, has some of the most stringent guidelines that are ignored the most often.

And once everything is done, and the program needs to be deployed, don't be bringing out the wine and preparing the feast just yet - deployment has its own share of guidelines and optimisations which could make or break the entire project.

Be it Java or Python, Github or Soureforge, Eclipse or Netbeans, each and every tool that has enabled you to achieve your goal in software development so far also needs to be handled with care. Each of these, due to their specialised abilities, can be optimised greatly if you know the best way to approach their usage.

And in the end, it is up to you to decide which best practices are relevant to your project. It may or may not be necessary for you to follow a particular testing process, or use a particular development methodology. Once you've figured that out and went down the right rabbit hole, you'll a better coder.

WHAT ARE BEST PRACTICES?

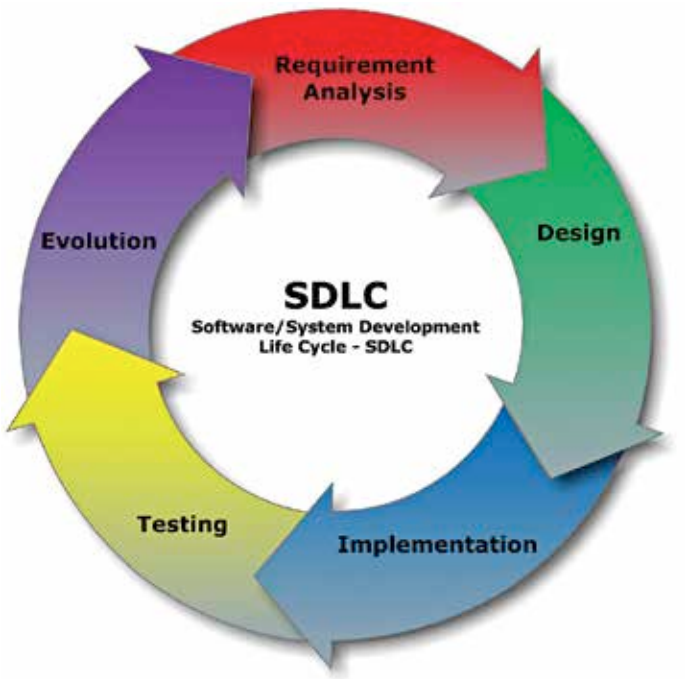
Why are they talked about so much? Are they just a bunch of academic hogwash or do they have any real implications?

There are various phases that are considered as best practices that you can follow in order to successfully develop a software product. These phases or steps are collectively known as the software development life cycle.

Life cycle or software development process is a structure imposed on the development of a software product. Multiple models have been developed

for such processes, each representing a different approach for different types of requirements and environment conditions.Software life cycle models describe phases of the software development cycle and the order in which those phases are executed. Each phase produces deliverables required by the next or other phase. Requirements are conceptualized into designs. Designs give birth to code. After coding and development the testing verifies the deliverable of the implementation phase against the original requirements. The Software testing team follow a Testing Life Cycle (STLC) which is similar to the development cycle followed by the development/project team. There are six phases in the Software development life cycle model:

1. Requirement gathering and analysis
2. Design/Architecture
3. Implementation or coding
4. Testing
5. Deployment
6. Maintenance



The SDLC is essential to every software project

Requirement gathering and planning

Project planning involves a series of steps or activities that need to be done. These activities include estimation of time, effort, and resources required and risks associated with the project.

Identification of requirements

This is the first phase of the software life cycle. Business requirements are gathered in this phase. Extracting the requirements of a desired software product is the first and the most important task in creating it. Customers usually believe they know what the software is to do, but the requirements may not always be clear or easy to comprehend. It requires skill and experience in software engineering to recognize incomplete, ambiguous or contradictory requirements.

This phase is the main focus of the project managers and stakeholders. It prevents obstacles that arise in the project such as changes in projects or organization's objectives, non-availability of resources, and so on. Proper project planning leads to optimal utilization of resources and usage of the allotted time for a project.

TASKS OF INDIVIDUALS INVOLVED IN SOFTWARE PROJECT

Senior Management	Project Management Team
• Involved in approving the project, employing personnel, and providing resources required for development of project. • Actively participating in project planning to ensure that it accomplishes the business objectives. • Tries to avoid conflicts among the team members. • Taking appropriate measures to avoid unwanted risks that may affect the project.	• Involved in implementing the process and procedure for completing the project. • Major task to monitor and manage project activities. • Crucial part in making plans for budget and resource allocation. • Helps in resource distribution, project management, issue resolution, and so on. • Understands project objectives for development and finds ways to accomplish the objectives. • Appropriate time and effort to achieve the optimal results. • Helps in evaluating tools and methods involved in project development.

The managers and the stake holders involved as well as the users conduct multiple meetings together in order to iron out details like: Who is the end user of the system? How are they going to use the system? What purpose is the system serving? What are the input parameters and expected output?

Cost estimation

The cost estimation includes the cost of hardware, network connections, and the cost required for the maintenance of hardware components. Along with the estimation of monetary cost, it is necessary to estimate the cost that is in the form of time spent as well as the effort of all the individuals involved in the project.

Risk analysis

Risk analysis involves identifying the risks that might occur and hamper the project schedule or the quality of the software being developed and defining steps and plans to mitigate these risks. Effective risk management makes it easier to cope with problems and helps to avoid sudden surprises that might harm the project.

Risks can be broadly classified as project risks, product risks and business risks. Project risks are those that affect the timeline of the project. For example, the sudden unavailability of resources. Product risks are those that affect the quality of the project. For example, failure of a component to perform as expected. Business risks are those that affect the organisation developing the product. For example, changes in the company goals or a competitor introducing a new product.

Risk analysis is one of the most crucial phases of SDLC

Finally, a Requirement Specification document or a Software Requirement Specification (SRS) is created which serves as a guideline for the next phase. The testing team follows the Software Testing Life Cycle and starts the Test Planning phase after the requirements analysis is completed.

Design

The main architecture of the system is designed during this phase. The primary objective is to convert the business requirements laid out in the previous phase into a robust and feasible system architecture.

The hardware and software requirements are specified in this phase and the overall product architecture is finalised. These design specifications serve as the input for the next phase of the software development life cycle.

The software/technical architects use the Software Requirements Specification or SRS as a reference to create the best and the most suitable architecture for the product. Usually, multiple approaches or solutions are proposed and thoroughly scrutinized before making a decision. These approaches are documented into what is called the Design Document Specification or DDS.

A design approach defines all the architectural modules of the product along with its data flow representation with the external and third party modules (if required). The internal design of all the modules of the proposed architecture should be clearly defined with the tiniest of the details in DDS.

The architectural design determines how the software will achieve its goal

All the stakeholders and the other necessary people involved review the DDS by taking various factors into consideration, like the risks involved, the design modularity, feasibility, robustness, the cost of development, budget availability, time constraints etc. After rigorous brain storming, they choose the best design and that approach is selected for the product. Approval and sign-off is required by developer, customer, business analyst and tester (all stakeholders including inheritors). Focus shifts to the next process step only after approval.

Implementation / Coding

This is the implementation phase of the Software Development Life Cycle (SDLC). This phase is where the actual development or coding of the software is done. The development team starts its work by reviewing and understanding the design and SRS document. This is the most important phase for the software developers and the longest phase of the software development life cycle.

In this stage of the Software Development Life Cycle the product is developed. The code is programming as per DDS during this stage. If the design is clearly laid out in a crisp, clean, detailed and organized manner, the code generation can be accomplished without much hassle.

In case there are ambiguities in the software requirements, or if they are unclear or if the software development team is unable to understand user requirements correctly and further clarification is required, the user must be contacted and queried. Additionally, the software development team also returns the requirements that are understood by them.

Developers must follow the coding standards and guidelines defined by their organization. Numerous technologies are available for software development, each of which specialise in different scenarios. Therefore choosing the best one is important. Based on the requirements and the system design, appropriate technologies are chosen.

There might be cases when there is a need to bring a change in the hardware or software specifications. However, these changes are implemented only after the approval of the user. When the software code is completely written, it is compiled along with the other required files. Code inspection and reviews are conducted after the compilation. These methods are used to correct and verify errors in the software code. Software testing is carried out to detect and correct errors in each module of the software code. After the software code is tested, the software is delivered to the user along with the relevant code files, header files, and documentation files.

Testing

Testing is the final phase of the SDLC before the software is delivered to the client. The tester's objective is to find bugs and defects within the system as well as verifying whether the application behaves as expected and according to what was documented in the requirements analysis phase.

Before testing can begin, the project team develops a test plan, which includes the types of testing you'll be using, resources for testing, how the software will be tested, who should be the testers during each phase etc. During this phase all types of functional testing are done. For example, unit testing, integration testing, systems testing, acceptance testing etc. In addition, non-functional testing is also done. Testers can either use a test script to execute each test and verify the results, or use manual testing which is more of an experience based approach.

If and when a defect is found, testers inform the developers about the details of the issue and if it is a valid defect, developers will fix and create a new build of the code which needs to be tested again. This cycle is repeated until all requirements have been tested and all the defects have been fixed and the software is ready to be shipped.

Deployment

After multiple stages of testing the product is delivered / deployed to the client for their use. As soon as the product is given to the customers they will first do the beta testing. The new build is tested again if any changes are required or if any bugs are caught, then they will report it to the development/engineering team. Once those changes are made or the bugs are fixed then the final deployment will happen.

Deployment also involves some testing on client side

Once the product is ready for deployment it is released formally on production setups. Mostly product deployment happens in stages as per the business strategy of that organization which involves multiple stages of testing. The product may first be released in a limited

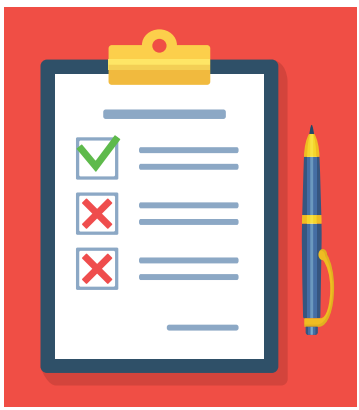
segment and tested in the real business environment (Known as UAT – User acceptance testing).

Then based on the feedback from customers, the product/software may be released as it is or with enhancements as per the market's requirement. After the product is released in the market, its maintenance is done for the existing customer base.

Maintenance

Once the customers start using the developed software, the occasional defects and bugs that arise, need to be solved from time to time. Periodically fixing bugs/defects on a production setup is known as maintenance.

There are various models devised which are followed during the software development cycle. These models are known as Software Development Process Models. Each model is suited for different requirements and has its own advantages and disadvantages. It is up to the organisation to decide which models are suited for their needs.

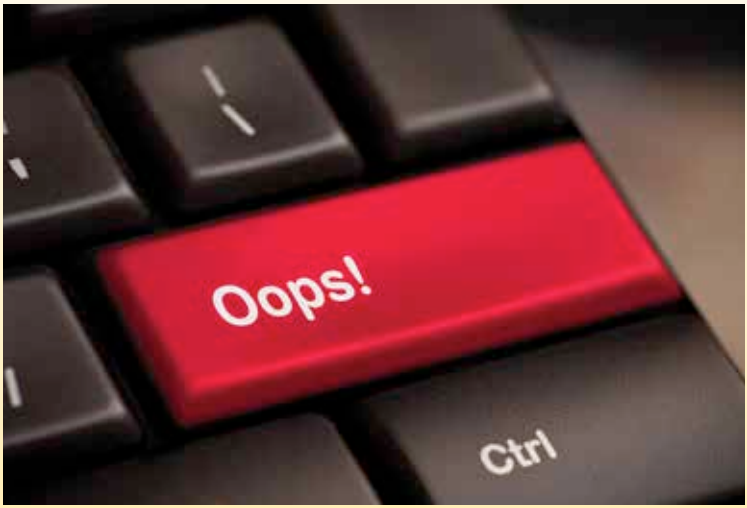


User Acceptance Testing decides the fate of your project

Most popular Process Models are as follows:

- Waterfall Model
- Prototyping
- Incremental Model
- Iterative Model
- Spiral Model
- Rapid application development
- Agile development
- V-Model
- Big Bang Model

So what are the best practices in each of these stages? Before knowing that, let's move on to understanding what happens when these are not followed with some of the best (or worst) examples of what happens in their absence.



CASE STUDIES

When software is badly written, it can end up causing unforeseen and often disastrous outcomes

The general perception towards the practice of software design and development is that it is a useful but extraneous skill not as mission critical as say the defence forces or as vital to the economy as the farming industry. The truth however is that software permeates almost every sector and affects the lives of most people in general. It is an integral part of the backbone of many industries and as such their reliability and performance becomes critical for them.

For an average software engineer today, a mistake in design or a bad coding practice will usually result in a rebuke, an increment loss or at the most a termination. However, in the past sometimes the effects of these mistakes have been very significant.

Let us take a look at some of the biggest 'software development horrors' that have brought researchers, corporations and sometimes even countries to their knees.

Code can crash and burn

Mars orbiter crashed by metrics (1999):

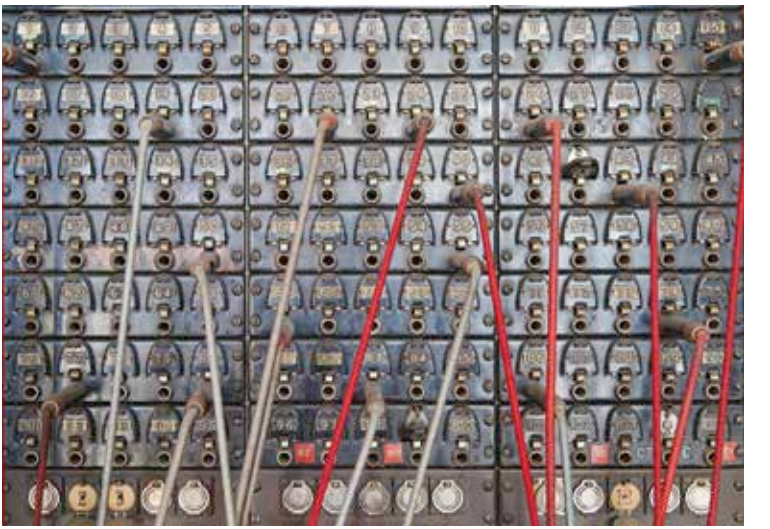
In September of 1999 the mars climate orbiter crashed due to a technical snag on its ground-based navigation software. The Mars Climate Orbiter (MCO) was to observe Martian weather systems and relay communications. The spacecraft was to fire its engines to enter orbit at an altitude of 145 km. Instead, the spacecraft entered the atmosphere at 57 km, and the friction caused it to burn up. This was no computer failure, but a human using the wrong units. NASA conventionally uses metric units, but USA-based engineers often use imperial. To put the orbiter in the proper orbit, a calculation for its momentum (mass x velocity) was needed.

Instead of the expected metric Newton-seconds, an engineer in the organization used imperial pound-seconds. The resultant calculation was 4.45 times bigger than it should have been. This caused the orbiter to enter at a much lower orbit than expected and the additional friction at the lower orbit was enough to compromise the structure of the craft and fry its systems. This project had cost NASA a cumulative of over $100 million.

AT&T communication down across USA (1990):

At 2:25pm on Monday, 15th January 1990, network managers at AT&T's Network Operations Center in Bedminster, N.J. began noticing an alarming number of red warning signals from various parts of their world-wide network.

Working backwards through the data, technicians identified the problem. The New York switch had performed a routine self-test that indicated it was nearing its load limits. As standard procedure, the switch performed a four-second maintenance reset and sent a message over the signalling network that it would take no more calls until further notice. After reset, the New York switch began to distribute the signals that had backed up during the time it was off-line.



Maybe calls should be forwarded like this

Across the country, another switch received a message that a call from New York was on it's way, and began to update it's records to show the New York switch back on line. A second message from the New York switch then arrived, less than ten milliseconds after the first. Because the first message had not yet been handled, the second message should have been saved until later. A software defect then caused the second message to be written over crucial communications information. Software in the receiving switch detected the overwrite and immediately activated a backup link while it reset itself, but another pair of closely timed messages triggered the same response in the backup processor, causing it to shut down also. The problem repeated iteratively throughout the 114 switches in the network, blocking over 50 million calls in the nine hours it took to stabilise the system.

AT&T alone lost more than $60 million in unconnected calls, of course the net loss due to the indirect business disruption from the terminated calls was significantly larger.

Passenger plane shot down by USS Vincennes(1988):

On July 3, 1988, 290 passengers and crew of Iran Air Flight 655 were travelling over an active conflict zone. However, sometime in the morning, the United States Navy guided missile cruiser Vincennes fired two missiles at

the plane, destroying the defenceless target and its civilian occupants with horrific precision.

While conspiracy stories blame the then active Bush administration for carrying out what they call a 'cold blooded massacre' as a show of aggression, one thing is for certain, the captain of USS Vincennes was given faulty and misleading information as a result of very poor UI design for the Aegis software.

Not only was the craft ascending (and not descending, as it was reported to be) but the craft's Identification, Friend or Foe (or IFF) reading, designed to distinguish between civilian and military craft, was reported to be Mode II (military) and not Mode III (civilian, as it actually was).

This mistake cost the lives of all the passengers and crew on board the Iranian flight and was the cause of much public humiliation for the United States on the world stage.

Scud missile hits American barracks undetected by US air defences(1991):

Twenty-eight Americans were killed on February 25, 1991 when a Scud missile hit the Army barracks in Dhahran, Saudi Arabia. The Patriot defence system had failed to track and intercept the missile.

The Patriot missile saves time as a fixed point register that had a length of 24 bits. Since the internal clock of the system is measured every one-tenth of a second, 1/10 expressed in a 24 bit fixed point register is

Faulty software can lead to terrible consequences

0.0001100110011001100110011 (the exact value of the representation 0.0001100110011001100110011 of 1/10 in the 24-fixed point register however, is 209715/2097152) . As we see, this is not an exact representation of 1/10. It would take infinite numbers of bits to represent 1/10 exactly. So, the error in the representation is (1/10-209715/2097152) which is approximately 9.5E-8 seconds.

On the day of the attack, the battery on the Patriot missile was functioning for 100 consecutive hours, hence causing an error of 9.5E-8x10x60x60x100=0.34 seconds (10 clock cycles in a second, 60 seconds in a minute, 60 minutes in an hour).

The shift of the defence gates calculated due to the error of 0.342 seconds was calculated as 687m. For the Patriot missile defence system, the target is considered out of range if the shift is more than 137m. Hence the incoming missile was considered out of range and was not targeted resulting in the death of 28 Americans in the barracks of Saudi Arabia.

NORAD reports U.S. was under missile attack(1980):

On June 3, 1980, at about two-thirty in the morning, computers at the National Military Command Center, at the headquarters of the North American Air Defense Command (NORAD) issued an urgent warning: the Soviet Union had just launched a nuclear attack on the United States.

Software could have ended the cold war in a rather catastrophic manner

As per protocol General William Odom called Zbigniew Brzezinski, the then president's national-security adviser to inform him that two hundred and twenty missiles launched from Soviet submarines were heading toward the United States. A retaliatory strike would have to be ordered at the earliest, Washington might be destroyed within minutes. Odom called back minutes later with a correction: twenty-two hundred Soviet missiles had been launched.

U.S. Air Force ballistic-missile crews prepared their launch, bombers ran to their planes, fighter planes took off to search the skies, and the Federal Aviation Administration prepared to order all commercial airliners to land.

Brzezinski decided not to wake up his wife, preferring that she die in her sleep. As he prepared to call Carter and recommend an American nuclear counterattack, the phone rang for a third time. General Odom apologised – it was a false alarm.

An investigation on the issue later found that a defective computer chip in a communications device at NORAD headquarters had generated the incorrect warning. The chip was worth forty-six cents.

Man deletes entire company with single command(2016):

A web hosting service provider had claimed on a forum to have accidentally run a command on his server that deleted all his data as well his client's data from it. This happened when he ran a script on the machine with the command 'rm -rf' along with a set of variables that were undeclared, like 'rm -rf {foo}/{bar} where foo or bar or both were undeclared. Although when pointed to a particular directory the command is supposed to clear its contents and is often used by server admins to routinely clear data from storage locations, as in this case the program was not specified a location, it proceeded to delete everything on the machine, including the backups from the drives that were mounted to it.

When software shift-deletes your life

Much hype was created around his post on the forum where people mostly advised him to declare bankruptcy or

start looking for a lawyer since he had effectively 'killed' his company. Some mentioned submitting the drives to data recovery experts although chances of success in such cases were very low. Still others were skeptical regarding the authenticity of the entire incident partly due to the mind bogglingly mismanaged and poorly designed architecture, criticizing the user for being careless enough to allow something like this to happen.

The incident if true, would no doubt have cost the provider a large sum of money to either restore his hard drives, or when his clients sued him for destroying their data.

Death by ambulance (1992):

The London Ambulance Service catered to 6.8 million people living in a 600 square mile area in the year 1992. It had 318 emergency ambulances, of which on an average 212 were in service at any given time. The service received anywhere, between 2000 and 2500 calls daily, 60% of which were requests for emergency services.

When SOS calls are answered by software

The morning of October 26, 1992 saw a new software being deployed to automate the entire process of handling calls and dispatching ambulances to the requested locations. Just a few hours later, however, problems began to arise. The software was unable to keep track of the ambulances in the system. It began sending multiple units to some locations and no units to other locations. The system began to generate a great quantity of exception messages on the terminals. The problem was compounded when people called back more because the ambulances they called had did not arrive. As more and more data was entered into the system, it became increasingly clogged. The next day, the LAS switched back to a part-manual system, and later had to shut down the computer system completely when it quit working altogether eight days later.

Because of the size of the area serviced by the LAS, many people were directly affected by the failure. There were as many as 46 deaths that would have been avoided had the requested ambulance arrived on time.

At the time the system went live, there were 81 known issues with the software and no load-tests had been run. There were no provisions for a backup system either. While the gap of 10 months between the time dispatchers were first trained to use the software and when it was deployed played its role in the disaster, the software had three primary flaws that immediately caused the failure:

Edge case data

The software system did not function well when given invalid or incomplete data regarding the positions and statuses of ambulances.

Interface issues

The deployed system had a wide variety of errors in different parts of the user interface. For example, parts of the MDT terminal screens had black spots, thus preventing ambulance operators from getting all information needed.

Memory Leak

The root cause of the main breakdown of the system, however, was a memory leak in a small portion of code. This defect retained memory that held information on the file server even after it was no longer needed. As with any memory leak, the memory eventually filled up and caused the system to fail.

Why did these things happen?

In each of the incidents above, one interesting fact to note is how a majority of them were not really an issue of coding quality or machine fault but how they were all to some degree or another caused by poor design or bad coding standards or even abstract and vague requirements.

While delivery of a finished application in due time is crucial, sacrificing design time or ignoring planning practices can ultimately result in much higher losses than expected. As is evident from the case studies above, the simplest of errors either in development, planning or design can compound into unforeseen losses.

The essence of this chapter thus, is to impress upon the reader the importance of the processes that go into planning the development of an application, designing it to the clearest granularity of detail, and only when you have defined both of these clearly should you begin to develop it.

METHODOLOGIES

A number of standard methodologies have proven to be highly beneficial for software development

Software Development Life Cycle and its Methodologies:
Software Development Life cycle (SDLC) is a process using which software is conceptualized, developed and maintained. SDLC includes 5 major phases/ stages, right from gathering the requirement of the client for whom the software has to be developed to the deployment and maintenance of that software.

Following are the 5 stages of SDLC:

1. Requirement Gathering and Analysis
2. Design
3. Develop
4. Testing
5. Deploy & Maintenance

Requirement Gathering and Analysis

This first stage of a software development life cycle focuses on the objective and requirements of the end user for whom the software has to be developed. This is a crucial phase in the life cycle since the software would be developed further basis on the requirements or the information gathered in this phase. This phase includes the analysis of functional and financial feasibility along with identifying the requirements from the stakeholders and defining the same in Software Requirement Specification (SRS). This SRS is then used in the later phases.

Design

Based on the requirements documented in the Software Requirement Specification (SRS) document, an architectural design is proposed for the software. This design is then documented in a design document. This phase includes identifying the hardware and software system based on the requirement and designing the system architecture. Creating UML (unified modelling language) diagrams such as use cases, class diagrams, sequence diagrams and activity diagrams are a part of this phase.

Develop

This is a phase where the software is actually developed and built and is usually the longest phase in the software development life cycle. In this phase, the actual code is written and is verified against the requirements by performing unit testing.

Testing

In this phase, the system is tested as a whole and the bugs/defects found in the system are reported, tracked and fixed. Different types of testing are performed until the system reaches the desired quality standards.

Deploy & Maintenance

Once the system is thoroughly tested in the testing phase, it is ready to go live. In this phase, the system is ready to be delivered, installed and put into use for the end user.

This phase also includes correction of the defects/errors that were not caught in the earlier phases along with the complete maintenance of the system which includes improving and ultimately enhancing the system.

Methodologies of Software development life cycle

There are various software development life cycle models or methodologies defined and designed to follow the software development process. Each of these methodologies splits the development work into above mentioned software development phases with an intent of allowing better management and planning. Each methodology follows a series of steps unique to its type to ensure success in the process of software development. Basically, a software development methodology is a framework that is used to structure, plan, and control the process of developing an information system.

Following are the methodologies commonly used in Software Development Life cycle:

- Agile Software Development
- Crystal Methods
- Dynamic Systems Development Model (DSDM)
- Extreme Programming (XP)
- Feature Driven Development (FDD)
- Joint Application Development (JAD)
- Lean Development (LD)
- Rapid Application Development (RAD)
- Rational Unified Process (RUP)
- Scrum
- Spiral
- Systems Development Life Cycle (SDLC)
- Waterfall (a.k.a. Traditional)

Agile Software Development

Agile methodology focuses on iterative development, which ensures delivering a software system to the client quickly by developing multiple iterations of the entire system and adding more features with each iteration.

In traditional software development methodologies like Waterfall methodology, the client may not be able to see the project until its completion which may take several months or years. When it comes to projects which are non-agile, a huge amount of time is allocated for requirement gathering, design, development and testing before finally delivering the product or the software to the customer. Whereas, Agile projects have iterations which are delivered in shorter duration during which the requested features are developed and delivered. Agile projects can have a single or multiple iterations and may deliver the complete system at the end of the final iteration.

Agile is a very popular methodology followed in the Industry

In Agile development methodology, the entire project is broken down into iterations which ideally consist of the same duration. At the end of each iteration, a working product is delivered to the client.

In this methodology, instead of spending major time on requirements gathering, the team will decide the basic core features that are required in the product and decide which of these features should be developed in the first iteration and handed over to the customer. The remaining features are covered in the next or upcoming iteration basis the priority and the requirement of that feature. At the end of each iteration, the team would deliver a working system which would consist of the features which were finalised for that iteration, this iteration is then incrementally enhanced and updated.

In the Agile approach, the entire system is incrementally developed and released in iterations. In this approach, the customer can interact and work with the iterations released and provide feedback on it. It also allows the development team to take up changes easily and make corrections in the existing code if needed.

Agile methodology gives more significance to collaboration with the team and with the customers, responding to change and timely delivery of working software.

Advantages of Agile Methodology

- In Agile Methodology, the delivery of software is incessant.

- The customers are satisfied with each iteration, which is ideally a working feature of the system is delivered to the client.
- The customers can actually work with and have a look at the feature which was developed as per their expectations.
- If the customer has any feedback or modification request to be done in the system, it can be done in the current release of the product.
- In this methodology, frequent interactions are required between the business people and the developers.
- In this methodology, attention is paid to the design of the product developed.
- Changes in the requirements are accepted and developed even in the later stages of the development.

Disadvantages of Agile Methodology:

- In Agile development methodology, documentation is less since only the major requirements are identified initially for the first iteration and thus not all requirements are documented.
- At times, in Agile methodology the requirement is not very clear (if not documented in detail) hence it's difficult to predict the expected result.
- In Agile methodology, at the beginning of the software development life cycle it's difficult to determine or evaluate the actual effort required for a few projects.
- In Agile methodology, projects may have to face some unknown risks which can affect the development of the software.

Agile methodologies

There are multiple Agile methodologies, a few of them are described below:

1. Kanban
2. Dynamic System Development method
3. Scrum

Kanban

Kanban is a popular framework used by software teams who practice Agile software development. It is a method for visualizing the amount and flow of work, in order to balance demand with available capacity and spot bottlenecks. Work items are visualized to give participants a view of the progress and process, from start to finish. Team members take up work as capacity permits, rather than work being pushed into the process when requested. It is a process designed to help teams work together more effectively.

Kanban is based on three basic principles:

1. Visualize what you do today (the workload that you have): Viewing all items in context of each other can be helpful.
2. Limit the amount of work in progress: This helps in balancing the flow-based approach so teams don't start and commit too much work at once, making it difficult to complete or some all of them in time.
3. Enhance flow: When some work is completed, the next important thing from the backlog is pulled into play.

   One can apply Kanban principles to any process they are already run-

Kanban board

ning. In Kanban, work is organized on a Kanban board. The board has columns which depict states that every work item passes through – from left to right. Work items are pulled along through the in progress, testing, ready for release, and released columns.

A basic kanban board has a three-step workflow: To Do, In Progress, and Done. However, depending on a team's size, structure, and objectives, the workflow can be mapped to meet the unique process of any particular team.

For every column (state) on the Kanban board "Work In Progress"-Limit (WIP Limit) should be defined. The WIP limit depicts how much work items are allowed to be in a certain state at any given point in time. If a state reaches its pre-defined WIP limit, no new work can enter that state. The whole team has to help clear the filled up state first. Work items trapped in a state will build highly visible clusters on the Kanban board. These

clusters make bottlenecks in the progress visible – one can simply look at the Kanban Board to see where the process needs improvements. Making the need for improvement visible challenges the team to change the way they work to avoid such bottlenecks in the future. That's how WIP limit act as change agent in Kanban.

Advantages of Kanban:

Kanban is one of the most popular software development methodologies adopted by agile teams. It offers several supplemental advantages to task orchestrating and throughput for teams of all sizes.

1. Shortened cycle times:

Cycle time is a key metric for kanban teams. Cycle time is the duration it takes for a unit of work to peregrinate through the team's workflow–from the moment work commences to the moment it ships. By optimising the cycle time, the team can confidently forecast the distribution of future work.

2. Planning Flexibility:

A kanban team is only fixated on the work that's actively in progress. Once the team consummates a work item, they pluck the next work item off the top of the backlog. The product owner is in liberty to reprioritise work in the backlog without disrupting the team, because any transmutations outside the current work items don't impact the team. As long as the product owner keeps the most paramount work items on top of the backlog, the development team is assured they are distributing maximum value back to the business.

3. Visual metrics:

One of the core values is a vigorous fixate on continually ameliorating team efficiency and efficacy with every iteration of work. Charts provide a visual mechanism for teams to ascertain what all they're supposed to amend. When the team can visually perceive data, it's more facile to spot bottlenecks in the process (and abstract them).

4. Fewer bottlenecks:

Multitasking kills efficiency. The more work items in flight at any given time, the more context switching, which obstructs their path to completion. That's why a key tenant of kanban is to inhibit the amount of work in progress

(WIP). Work-in-progress limits highlight bottlenecks and backups in the team's process due to lack of focus, people, or adeptness sets.

5. Continuous Delivery:
We know that perpetual integration–the practice of automatically building and testing code incrementally throughout the day–is essential for maintaining quality. Now it's time to meet perpetual distribution (CD). CD is the practice of relinquishing work to customers frequently–even daily or hourly. Kanban and CD complement each other because both techniques fixate on the just-in-time (and one-at-a-time) distribution of value. Kanban teams fixate on optimising the flow of work out to customers.

Disadvantages of Kanban

1. Requires constant Board Monitoring:
While many teams may consider this a benefit, it cannot be denied that a kanban board must be constantly monitored, to ascertain that the cards do not become archaic, thus causing more harm than good.

2. Possible bottlenecks:
If your team doesn't plan for and deal with blocked cards well, it's probably going to be "stuck" for a long time. While this could also occur utilizing other methodologies, kanban presents a particular hazard since it fixates on cards makes scheduling and milestoning rather arduous.

3. Potential for complexity:
Since kanban is such a flexible methodology, it's entirely possible for a board to be engendered that is perilously over-engineered and intricate.

Dynamic Systems Development Method (DSDM)

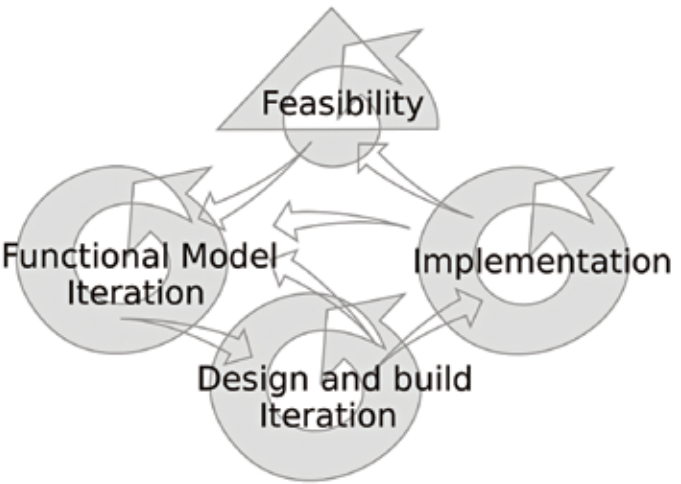
DSDM is an agile software development methodology. It is an iterative, incremental approach that is largely predicated on the Rapid Application Development (RAD) methodology.

The method provides a four-phase framework consisting of:

- Feasibility and business study
- Functional model / prototype iteration
- Design and build iteration
- Implementation

DSDM relies on several different activities and techniques within each phase predicated on these principles:

- Self-managed and empowered teams must have the ascendancy to make time sensitive and critical project-level decisions.
- Best projects are evolved through direct and co-located collaboration between the developers and the users.
- Design and development is incremental and evolutionary in nature and is largely driven by conventional, iterative utiliser feedback.
- Frequent and incremental distribution of working software is valued over infrequent distribution of impeccably working software.
- Working software deliverables are defined as systems that address the critical, current business needs versus systems that address less critical future needs.
- Perpetual integration and quality assurance testing is conducted in-line, throughout the project lifecycle.
- All changes introduced during development must be reversible.
- Overtness and transparency is inspirited through customary communication and collaboration amongst all project stakeholders.



DSDM Development Process

Scrum

In Scrum methodology, each project is built in a series of iterations which are of fixed lengths known as 'Sprints'. These sprints give the develop-

ment teams a framework for shipping the software in regular modulation. The milestones defined, which is usually the end of each sprint comes frequently, thereby giving continuous inspiration and energizes everyone. Small iterations also boost up the significance of good estimation and fast & continuous feedback from tests.

Scrum is a popular Agile concept

The following 4 ceremonies bring structure to each sprint in a Scrum:

1. Sprint planning: In this phase, the entire team plans on what can be completed in the coming sprint.
2. Daily Stand-up: This is usually a meeting held by the team members so that everyone is in sync. It is also called as daily scrum.
3. Sprint demo: An information sharing meeting held to showcase what they been shipped in that sprint.
4. Sprint retrospective: This meeting is held to review what went well and what went wrong in the previous sprint to make the next sprint better.

Advantages of Scrum

1. Allows for Rapid prototyping:

A maximum of only one month can be devoted to any particular Sprint Goal.

With this timeline, scrum sanctions for rapid coding and development of ideas or components that may be experimental or may even fail, without worrying about potential downsides.

2. Encourages continuous productivity:
With the help of daily scrums, an insight about the progress of the work assigned to all team members can be achieved and the required feedback or improvements can be suggested wherever required to keep the development on track.

3. Keep customers in loop:
Since Scrum follows an Agile framework, the customers are involved in every iteration and can assess the progress and provide continuous feedbacks throughout the development life cycle.

Disadvantages of Scrum:

1. Abundance of Meetings:
Given the adoption of an agile methodology, Scrum presents ample amount of meetings which invariably end up taking too much time from the entire development cycle.

2. Requires Lenient Leadership:
Successful implementation of scrum requires that managers and leadership are able to trust the development team and give them complete freedom to work independently. A proper scrum emphasizes the importance of separating the management of development teams and that of scrum masters of product owners.

3. Potential difficulty with estimation:
This is a similar disadvantage as that of the Agile methodology. Since scrum follows agile framework, it can often obfuscate the genuine time and monetary costs of a project (or even aspects of verbally expressed project), often until a few months down the line.

Rapid Application Development (RAD)

RAD is a software development methodology which focuses on rapid prototyping and iterative delivery. By utilising a rapid application development method, designers and developers can aggressively utilise cognizance and

revelations gleaned during the development process itself to shape the design and or alter the software direction entirely.

Rapid application development generally follows a process which includes four basic steps:

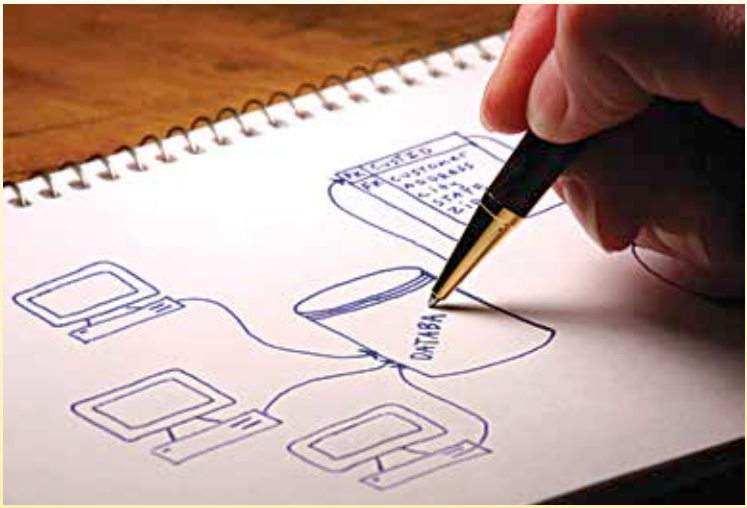
1. Planning requirements: During this initial stage designers, developers, and users come to a rough acquiescent on project scope and application requisites, so that future stages with prototyping can commence.
2. Utilizer Design: Utilizer feedback is amassed with heftily ponderous accentuation on determining the system architecture. This sanctions initial modeling and prototypes to be engendered. This step is reiterated as often as compulsory as the project evolves.
3. Rapid Construction: Once rudimental utilizer and system design has commenced, the construction phase is where most of the authentic application coding, testing, and integration takes place.
4. Cutover: The final Cutover (or Transition) stage sanctions the development team time to move components to a live engenderment environment, where any obligatory full-scale testing or team training can take place.

Advantages of RAD

1. Quick application development and delivery.
2. Less number of resources is required.
3. The progress of the project can be easily visualized.
4. Lest testing activity required.
5. The client reviews everything from the very beginning of development, reducing the probability of missing any requirements.
6. It is cost effective and good for small projects.

Disadvantages of RAD:

1. Since the number of resources required is less, the ones hired should be highly skilled.
2. The client's feedback is required in each development phase.
3. Not a good process for long term and big projects.
4. It is difficult to manage and automated code generation is costly.



BEST PRACTICES FOR ARCHITECTURE

It's always smarter to spend more time on structuring from the beginning rather than restructuring later.

The central aspect to perhaps any application, whether it be a simple app or a multi-layered enterprise application is the amount of thought and effort that goes into designing it.

Converting requirements on a sheet of paper into a well-defined set of technical specifications is arguably the most challenging (and fun) part in the entire development cycle.

Needless to say, this stage is as crucial as it gets when it comes to maintaining the quality of an application. Essentially, it can make or break it even without having written a single line of code!

A comprehensive guide to architecting an application (however small it maybe) is out of scope of this tutorial or most of them. It takes years of experience, technological expertise, domain knowledge and innovative thinking to design a very successful application. Thus, it is of no surprise that the software architects of a company hold one of the highest paid and respected positions within the organisation.

Having said that, there are a few quick tips and guidelines that can help us avoid some major fall-throughs when designing an application. Let us take a look at some of these well-known (and some not-so-well-known) practices and how they relate to their life cycle.

Communication is key

The moment you are handed a document of requirements, the clock starts ticking, there is limited time assigned to each phase and you are expected to deliver the design on time. In today's world, especially with startups where there are aggressive timelines, this translates to all-nighters, war-rooms and marathon meetings. However, a subtle and dormant flaw might have already crept in. It is possible (and likely) that while the top-level requirements were clear

Avoid gruelling deadlines by planning and communicating with your team efficiently

and could be easily translated to design, some specific second-tier requirement might have more than one possible interpretation. The architect's perspective also might not align with the delivery team's, causing absolute chaos later on.

This is more relevant in teams that are located at different geographical locations as clear communication becomes more difficult in these cases. There is a very interesting and somewhat underrated law regarding this as postulated by Michael Conway: "Any organisation that designs a system (defined broadly) will produce a design whose structure is a copy of the organisation's communication structure."

This roughly means that your application's design mirrors the way you communicate across teams within the company! Hence, it is imperative to have clear communication across teams within the company, get the requirements clear down to their grainiest level and agree on the acceptance criteria of the required software well ahead of development.

Zoom out

One of the challenges that most architects face when designing an application from ground up is usually the sheer size of the requirements stack and the delivery commitments associated with it.

Often this ends up with "burst coding" where the development team is given a module to work upon immediately once the requirement is available with the idea that a change would be accommodated as required.

While agile methodologies of development have their merit, none of them advocate developing something without a concrete and stable architecture around it. The first step is then to take a look at the big picture, to zoom out, identify the components that are central to the application and figure out the way they can be connected. Only when you have a clear high-level structural definition of the entire application should development begin.

Choice of technology

You have your design board ready, you know the components involved and the way they are intended to work. Now comes the fun part of choosing technologies to fit your requirements. Some projects have spent weeks and months doing Proof of Concepts (PoCs) on a variety of technologies to find out which fit their needs and arrive at a solution. Simultaneously, others have blindly gone for the industry standard while some have even adopted

technologies based on what skill was most common among the dev team that would require minimal learning effort.

There is no single rule in adoption of a technology to fit your needs but questions like:

- What are the functional and non-functional requirements that this technology needs to satisfy?
- What are the selection criteria?

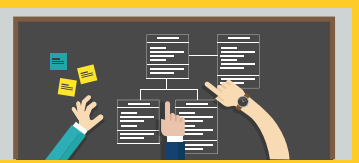
These questions can go a long way in filtering out a few from among a number of available options.

Another trend that is a bit dangerous is how sometimes we blindly follow or try to incorporate whatever technology is "hot" in the industry. While using an industry buzzword might seem like a no-brainer, sometimes based on the situation at hand, the "industry-ranked-best" tool might not be the best fit for your application. An analogy for this would be how swiss knives are known for their all-powerful and capable utility but if you can carry only a small bag (or only a light tool), then a swiss knife, however well rated and recommended, might not be the solution.

At the time of writing of this article, a couple of technology buzzwords that have been dominating the industry are noSQL, Cloud Computing, Machine Learning etc. While each of these technologies are a powerful and formidable tool, it would depend entirely on the use-case whether we are able to benefit from them or not. Long story short, stick to the questionnaire above for your choices, don't get carried away when you hear how Amazon has an incredible face-recognition technology and try to apply it to your accounting software. Unless there is a use-case for it.

NOTE:

A lot of hype has evolved around tools (UML-based or otherwise) that help in this phase of the design. While there are tools offering a variety of features like Modelio or LucidChart, it is the opinion of this author that in most cases (if not all) whiteboarding, sketches and post-its



are the best methods to start the design phase. It can then be moved on to one of the tools for a more expanded and detailed definition.

Choosing a database

If you have ever designed or worked on an application that stores data in a persistent manner then you are most likely familiar with a database of some kind. While choosing a database might seem trivial compared to designing the logic layer sometimes a choice in the database technology can actually affect your application logic.

For example: A company which had chosen to migrate to an Oracle database from a mysql one at some point of time, in order to comply with client guidelines, realised that some of their queries on production were not working on the new database! A quick investigation revealed that Oracle limits the number of positional parameters in a query to 2000, post which it throws an exception! Eventually, they had to rewrite the entire application logic around their new database system.

When choosing a database of some kind, be very clear about the kind of data that you are expecting to store in it now and in the near future. Whether it's relational or has large objects or needs very fast access time. Also, on the percentage of read or write operations, the choice of a database for specific uses (like archiving) might vary.

Choosing language(s) for development

- `python > java ? python : java;`

While the above statement might have drawn more than a few cringes from the readers it remains a valid subject, that once we have our technology stack in place we would need to decide on the language of our choice to code in.

Of course a qualitative analysis on the benefits of choosing a strongly typed or a weakly typed language , an object oriented or a procedural one or one that supports functional programming would be too long and likely cause a small debate war but what we can attempt to do is to ensure that we choose one that has a clear syntax, has good IDE support, a solid collection of libraries with good API support for most if not all of the technologies in our stack.

Another key factor here is the learning curve for the developers using the language, while some on-hand learning is expected, throwing a dev team into a war room with a completely new language and asking them to create a well formed application in a month would most likely not yield expected results.

User Interface(UI) and User Experience(UX)

A few years ago, Amazon (who had then patented its 'one-click-checkout') had a simple buying process. You browsed for items on the website, added them to cart, and then once you clicked on the cart icon, there was a button labelled 'checkout'. Once you click the 'checkout' button, you were presented with a simple form that asked you to login or register. A large number of carts were abandoned at this point,simply because users were not willing to 'register' and were irritated that they had to do so to make a purchase.

Note that Amazon did not ask for additional information on the form, but even then had to suffer such losses. They quickly came up with a 'continue' button and a message saying: "You do not need to create an account to make purchases on our site. Simply click "Continue" to proceed to checkout. To make your future purchases even faster, you can create an account during checkout." This little change resulted in a 45% increase in completed purchases, and for the first year, the site saw an additional $300M in revenue.

UI/UX, while usually designed as an afterthought, is definitely one of the biggest deal breakers to any application that faces end customers on a regular basis. It is then perhaps of paramount importance to ensure that a smooth and fluid UI coupled with an amazing experience is always present in the application to retain users.

Even exception handling and error pages can be path breaking and wow the customers in a way that keeps them coming for more (Google jump game for no internet connection page).

A beautiful and engaging UI usually has a clean and fluidic nature, with compatibility across all major browsers (yes, even Internet Explorer) and is also capable of restructuring the content across to automatically fit the smaller screen on a tablet or mobile.

Don't implement then architect (a lot)

Once you have your choice of technology stack in place and your dev team is ready with their choice of programming language, development for the application can begin. However, it is possible that at a later stage of iteration you realise that the stack of technologies, while perfect for their use, do not cooperate with each other very well.

For example, a database of choice might handle asynchronous calls well while the UI end technology might assume to make synchronous calls to fetch data to display to the user. In such a case, the application tier is expected to miraculously come up with a "bridge" between these two.

Implementing the code before architecting can be a bad idea in the long run

While in this particular case, it is possible to do so and trivial, sometimes such changes might not sync with the goals of the application tier whose primary function is to hold application logic and business flows.

Such cases are often seen to increase over time as the application grows, however, care should be taken while architecting the application tier.

We must ensure to not do a lot of coordination activities across our technology stack rather than concentrating on the business logic.

In some critical cases, perhaps an additional layer commonly known as an orchestration layer might be designed to act as a 'gate' across the technologies but the additional effort for it should also be taken into account.

Scalability from scratch

The application you design today for a certain volume of work would most likely end up facing a higher amount of it over a period of time. Thus scal-

As your system scales to higher numbers, it becomes difficult to maintain it

ability is a necessity for it. One way of doing this is to modify the application as and when required to ensure we are able to handle just enough work. This however requires constant development and maintenance effort and might also end up degrading the quality of both the code and the design. Scalability of an application should be handled right from the start, as a non-functional requirement, it is often less prioritised than say, a new shiny app but once the volume of work starts exceeding the application's capabilities, it needs to be scaled immediately.

The correct way of ensuring that an application is scalable is to build it right into the design. Some popular ways of doing this are:

Decreasing the processing time

It's elementary really, if each process of work takes less time to complete, then more amount of it can be done sooner. When designing an application we often strive to achieve perfectly abstracted and modular components. While this is advantageous for a fair number of reasons, sometimes it can be unnecessary and affect performance. Abstractions and data transformations all take up precious processing time, so we must be careful not to overuse them.

Partitioning work

Another way of getting more things done is to partition the work into inde-

pendent (or nearly independent) tasks that can then be done in a parallel manner to ensure faster completion. I/O is one such area where partitioning has helped a great deal in ensuring faster communication software for all our mobile devices.

Concurrency

Working on a large number of jobs simultaneously also helps getting more amount of work done faster. Concurrency is of incredible use when doing API calls, or handling user requests (WhatsApp handles two million concurrent text data requests simultaneously and 74 million requests per second).

This, however, can be tricky as when coded incorrectly can very easily corrupt system behaviour as well as persisted data. Locks and monitors should therefore be used as and when necessary but care should be taken to ensure that they are only in use for the least possible time duration.

Extensive automation testing

When you start off with an MVP and are testing everything manually, you can be sure to have release cycles of a reasonable time frame, but once the feature set and the codebase grows over time, manual testing is no longer a viable option. Often at this point, architects design a secondary supporting automation test suite only to realise that their architecture does not support automated tests (at least not effectively). Right from the start, automation should be a design facet for all the components of the application. Each component should ideally have an independent set of test cases that are exhaustive of all of its components. The entire application should have another set of test cases for regression tests.

A well designed automation suite can reduce bugs in the code or even detect if the design of the architecture is compromised.

Conclusion

These simple tips, although evident and obvious, are often ignored or compromised upon for the sake of time constraints or other concerns. Granted these are only guidelines and to follow them to the letter would perhaps increase the design time for an app, sometimes unacceptably so. However, they do help reduce both time and effort in the long run and when followed with enthusiasm they can actually help somebody design a great application that is both scalable and robust.

BUY vs BUILD

Should you spend money on buying a solution or rather build it on your own? We explore both the sides.

Today, innovation in technology is progressing exponentially. So, IT groups must make their deliverables up-to-date and compatible with the latest technologies in order to stay ahead of the competition. Now, for doing so, the most common dilemma IT companies face is to whether buy a software or build it. Many times, organisations have made the wrong decision as they failed to consider all the factors (or they did not have a clear idea) to be considered. In the past, decision making was relatively easier since older enterprise applications had an average of 75 percent of the features that organisations needed. Currently, with the introduction of new technologies like cloud computing, the tables are turning. Before deciding whether to go with an in-house custom-built

or commercial off-the-shelf product, the developers must have a clear idea about the requirements and objective. After analysing the following points, only then can the final evaluation be made.

Time complexity

Time plays a crucial role while selecting any one of the solutions. Consider a project which has a duration of one year. Now if more than a month is spent in developing a supporting feature it's not justified. Instead, an existing off-the-shelf solution that does the job well, will be considered.

Size of the developer community and in-house skills

Whenever an organisation considers building a custom software at first, it should consider two things. One is the size of its developer community and the other is the in-house skill it posesses. The organisation should have enough experts capable of building, maintaining, and supporting the solution. Lower number of skilled professionals is one of the primary reasons behind the significant reduction of ROI of custom solutions.

Cost analysis

Once the requirements get cleared, and there is an estimation of resource requirement, a cost analysis can be run. Most of the people have a perception that an in-house solution will always be cheaper as they don't need to pay to a third-party. This is a misconception. Cost associated in replacing technology, extending functionality or rebuilding and reengineering of poorly designed systems, sometimes exceeds the cost of buying that product. Many times, there is a lack of skilled personnel. Then they are required to conduct training sessions or recruit people with the necessary skills and experience. Hence, the cost increases. On the other hand, if the organisation is buying a ready-made product, the price might come out to be justifiable. Once the product is bought, in-house support cost should also be considered which is usually overlooked. Both the long term and short term support cost should be considered.

ROI potential

ROI or Return Of Investment is the most important factor that needs to be addressed. While running all sorts of analysis, an estimated ROI should be presented. It should also specify how the ROI will be measured and whether the product will make enough money to justify its cost. In most of

the cases, the answer makes it easier to make a final decision. For example, an organisation might not have skilled employees and proper infrastructure for building custom business solutions and off-the-shelf products. In most cases, the initial cost of implementing an enterprise solution is significantly much higher than an in-house custom solution. Enterprise solutions usually provide best ROI over the long run because of the following reasons:

- The product vendor makes the product compatible with new technologies
- The vendor itself must rectify all design problems
- If there is any architectural change, all the migration should be carried out by the vendors
- Quality of these products are higher than custom products as they are already tested and used by some other organisations.

Enterprise solutions provide the best ROI in long term

Availability of the product vendor

At the end, organisations should research the market place. If they decide to buy a software, they need to check if there is any product available that matches with their requirements. If the required product is not available in the market, then they have only one option and that is to hire skilled people and build it. Again, sometimes there are many vendors for a single product, offering different prices. In such cases, the final decision needs to be made wisely because quality cannot be compromised. Both the in-house and off-the shelf products have their own pros and cons, and we will explore a few of them.

Advantages

Build

If the organisations decide to build the software on their own, they will have total control over the development and features. All the required and unique features they need, they can build. The organisation will have complete ownership of the software code. In the future, they can reuse or modify it as per their needs. With custom solutions, they will have direct access to the application system experts.

Building the product on their own, they no longer need to be dependent on a third-party vendor. They'll have the freedom of selecting any technology they wish to incorporate which will be easily integrated into their existing infrastructure. The control of the product roadmap will be in the hand of the organisation itself. They will always have the freedom to decide when and what features need to be added to the product. Finally, the end product will be unique where none of the competitors will have a similar application.

Buy

The most important advantage of off-the-shelf products is that they are finished products. Whenever an organisation needs it, they can get it. In a more real scenario, there could be an urgent requirement in case they can't afford the time to build the application. Getting these off-the-shelf products becomes an the ideal solution at such times. Flexibility and adaptability of these products are generally higher in comparison with custom solutions as these products must compete in the market. Number of bugs is minimal as these products have already been used and tested by various other organisations. Another advantage is these off-the-shelf products are associated with a one-time cost. Sometimes they turn out to be much lower than custom built products on long term. The product vendor provides training and expert support, reducing any kind of burden from the organisation. The product vendors are solely responsible for continuous functional enhancement of the product and the organisation doesn't need to supply its resources for this.

Disadvantages

Build

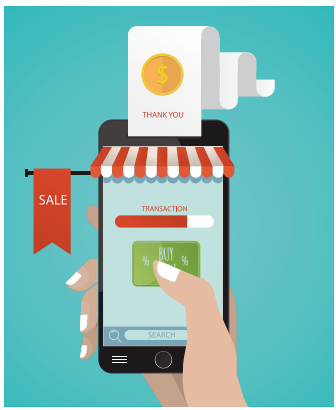
Building an in-house software is a lengthy process. The organisation has to invest a lot of resources in terms of developer, hardware, and tools. Time is required for conducting requirement analysis before the actual development can be started. If there is an urgent requirement then an in-house solutions isn't a solution. Some training has to be conducted by the organisation itself and eventually engage developers for providing full-time support. Hence, this could result into a resource crunch. It is challenging to stay updated when business demands rapid changes. Finally, the biggest disadvantage is that development of custom software is a trial and error method. Stability of these kind of products is lower than off-the-shelf products. Development of custom software is generally requirement driven. Hence, these products aren't much flexible.

Buy

The main disadvantage of off-the-shelf products is organisations can only have those features that they offer. They cannot modify those products according to their requirements. Sometimes, vendors provide custsomisation options while asking for additional money. These products might also have many extra features which aren't required by the organisation. These extra features make the products heavy.

A lot of integration is required when moving to an off-the-shelf product as the product may not be developed using the existing technologies. Also, some training will be required to provide basic knowledge about the product to the employees.

Whenever there is any technical or design issue, organisations have to rely on product vendors. Such services might be charged additionally by the vendors. Above all, organisations have to wait for the support that ends up wasting time while slowing down operation.



Third-party vendors might demand additional money for extra features

Final thoughts

At the end, it can be concluded that none of the above solutions are perfect. Both have a considerable share in pros and cons over the other. Making a decision on whether to buy enterprise software or build it themselves, depends upon several factors. Everyone should run an analysis based on the nature of the organisation, complexity of the problem, time, resources and the amount of money they wish to spend. The most important thing is both the options should be fully explored before taking a final decision.

BEST PRACTICES FOR CODING

Apart from avoiding those notorious semi-colon errors, structuring your code is quite a healthy practice.

Comments

Comments can be very helpful if it is well written and strategically placed. However sometimes, dogmatic comments can make a module very clustered. All comments are not good. Like nothing is more harmful than misleading comments in code. We use comments to convey information that we failed to express via our code. If programming languages were completely expres-

sive, we wouldn't need comments at all. But that is not the world that we live in. Plain code just isn't expressive enough to explain itself. Sometime s, code handle very complex logics that forces us to use comments. Every time you find the need to write a comment, stop and check if there is a better way to express yourself through the code.

The primary reason to avoid comments is because overtime they become very misleading. Code changes, it is refactored and evolved. However, comments aren't. If it is tough to maintain code, it is a lot tougher to maintain comments. Programmers can't realistically maintain them. Often, they get separated from the code they were meant to explain.

Very often, we write code that we know is messy and disorganised. Rather than write comments to explain the code, the best practice is to structure and improve the code.

Good comments

Some comments are beneficial and necessary. For example,

- **Legal comments:**

Comments that we are forced to write for legal reasons so as to follow our corporate coding standards.

eg.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

- **Informative comments**

It is sometimes a good practice to write comments that convey basic information.

eg.

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher =
Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

In this case the comment lets us know that the regular expression is intended to match a time and date that were formatted with the SimpleDateFormat.format function using the specified format string.

- **Clarification**

Comments may be used to explain a complex expression or express the return value of a statement in a simpler way.

eg.

```
assertTrue(x.compareTo(x) == 0);              // x==x
assertTrue(x.compareTo(y) != 0);              // x!=y
assertTrue(xy.compareTo(xy) == 0); // xy==xy
assertTrue(x.compareTo(y) == -1);    // x<y
assertTrue(xx.compareTo(xy) == -1);           // xx<xy
assertTrue(yx.compareTo(yy) == -1); // yx<yy
assertTrue(yy.compareTo(yx) == 1); // yy>yx
```

- **TODO comments**

It is sometimes reasonable to leave "To do" notes in the form of //TODO comments. In the following case, the TODO comment explains why the function has a degenerate implementation and what that function's future should be.

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout
model
protected VersionInfo makeVersion() throws Exception
{
return null;
}
```

- **Javadocs**

While writing public APIs you should definitely write good javadocs. They are extremely helpful

eg. Javadocs for the Java Standard Library

Bad comments

Unfortunately, the comments that many programmers write fall in this category.

- **Ambiguous / unclear comments**

Writing a comment just for the sake of writing one, leads to unclear and ambiguous comments. If you do decide to write a comment, spend time and make sure you write it in the simplest and clearest way.

- **Redundant comments**

Comments are meant to express what we failed to express via code. Their purpose is not to translate code to english.

eg.

```
// Utility method that returns when this.closed is true. Throws an exception if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis) throws Exception {
                if(!closed) {
                        wait(timeoutMillis);
                        if(!closed)
                                        throw new Exception("MockResponseSender could not be closed");
                }
}
```

Commenting helps others understand your code, even yourself while revisiting for errors

- **Mandatory comments**

It is unnecessary to mandate that every function should have a java-doc. This leads to a cluttered and clumsy code base. eg.

```
/**
*   @param id The id of the item
*   @param cost The cost of item
*   @param quantity The quantity of item
*   /
```

```
public void addItem(String id, double cost, int quantity){
            Item item = new Item(id,cost,quantity);
            item.add();
}
```

- **Commented out code**

It is a terrible practice to comment out code and leave it there.

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.
getByteCount());
//  InputStream resultsStream = formatter.getResult-
Stream();
//  StreamReader reader = new
StreamReader(resultsStream);
```

Others, who see this commented code will never know whether it is commented for a reason or if it's okay to delete it. Thus, they will leave it be.

Naming conventions:

Naming is an integral part of software development. We name our variables, our methods, our classes and packages. We also name our directories, our jar files. Naming things is a constant process, therefore we need to do it properly. There are simple guidelines that need to be followed in order to write good names.

- **content revealing names**

The name of a function or a variable should give you all the necessary information about them.

It should tell you what they do and why they exist. It is a good practice to choose good names and also to change names if you find better ones.

If a name requires a comment, then it is not a good name. It needs to be self explanatory.

eg. `float r = 5.0f;   // rate of change`

the variable r does not explain anything, the name should specify what the variable does.

eg. `float rateOfChange = 5.0f;`

The power of following naming conventions :

```
1. public List<int[]> getThem(){
            List<int[]> list1 = new ArrayList<int[]>();
```

```
                for (int[] x : theList)
                        if(x[0] == 4)
                                        list1.add(x);
                return list1;
}

2. public List<int[]> getFlaggedCells(){
                List<int[]> flaggedCells = new
ArrayList<int[]>();
                for(int[] cell : gameBoard)
                        if(cell[STATUS_VALUE] == FLAGGED)
                                        flaggedCells.
add(cell);
                return flaggedCells;
}
```

Programs 1 and 2 both perform the same function but their readability varies vastly. Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit.

- **Misleading names**

Avoid misleading names that create confusion. Do not name a group of elements "itemList" if it isn't a List. The word List has a certain meaning. Instead, name the elements itemGroup or simple items.

Avoid using names which differ in very small ways. It is tough to spot the difference between a XYZControllerForEfficientHandlingOfStrings in one module and XYZControllerForEfficientStorageOfStrings in another.

- **Interfaces and implementations**

These are sometimes a special case for encodings. For example, say you are building an ABSTRACT FACTORY for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? ShapeFactory and ShapeFactory? I prefer to leave interfaces unadorned. The preceding like I, so common in today's legacy wads, is a distraction at best and too much information at worst. I don't want my users knowing that I'm handing them an interface. I just want them to know that it's a ShapeFactory. Implementation is encoded. Calling it as ShapeFactoryImp, or even the hidden CShapeFactory, is preferable for encoding the interface.

- **Class names**

Class names should not be verbs. A class name should be a noun or a noun phrase like Item, Department or Employee. Avoid words like Processor or Info.

- **Method names**

Methods in code should have verb phrase names like deletePage and post-Payment or save. Mutators, accessors and predicates should be named for their value and prefixed with get, set, which is in accordance to the javabean standard.

eg.

```
string name = employee.getName();
table.setName("Cost");
if (document.isValid())
```

- **Pick one word per concept**

Pick one word for one abstract concept and stick with it. For instance, it's confusing to have fetch, retrieve, and get as equivalent methods of different classes.

It's confusing to have a manager, a driver and a controller in the same code base. What is the essential difference between a DeviceManager and a Protocol- Controller? Why are both not controllers or both not managers? Are they both Drivers really? The name leads you to expect two objects that have a different type as well as having different classes.

Variables with unclear context.

```
private void printGuessStatistics(char candidate, int
count) {
String number;
String verb;
String pluralModifier; if (count == 0) {
number = "no";
verb = "are";
pluralModifier = "s";
}
else if (count == 1) { number = "1";
verb = "is"; pluralModifier = "";
```

```
•
• }  else {
• number = Integer.toString(count); verb = “are”;
• pluralModifier = “s”;
• }
• String guessMessage = String.format(
• “There %s %s %s%s”, verb, number, candidate, pluralModi-
fier );
• print(guessMessage);
• }
```

Variables that have a context.

```
• public class GuessStatisticsMessage { private String
number;
• private String verb;
• private String pluralModifier;
• public String make(char candidate, int count) { createPlu
ralDependentMessageParts(count); return String.format(
• “There %s %s %s%s”,
• verb, number, candidate, pluralModifier );
• }
• private void createPluralDependentMessageParts(int count)
{ if (count == 0) {
• thereAreNoLetters();
•       }  else if (count == 1) {
• thereIsOneLetter();
•
•       }  else {
• thereAreManyLetters(count); }
• }
• private void thereAreManyLetters(int count) { number =
Integer.toString(count);
• verb = “are”;
• pluralModifier = “s”;
• }
• private void thereIsOneLetter() { number = “1”;
• verb = “is”;
• pluralModifier = “”;
• }
• private void thereAreNoLetters() { number = “no”;
• verb = “are”;
```

```
• pluralModifier = "s";
• } }
```

Documentation

Irrespective of the complexity of code, any good code should be well documented. Documentation is imperative to the creation of good long-lasting software. Effective documentation helps in building, updating, and using the software.

There are two kinds of documentation, internal and external. Internal documentation is usually in the form of comments. Comments should be terse when possible and clearly show what the purpose of the code is. Here are some good practices for commenting programs.

a. Have standard commenting practices
b. When modifying code, always keep the commenting around it up to date.
c. At the beginning of every module, it is helpful to provide standard, proper comments, indicating the routine's purpose, limitations, and assumptions. A boilerplate comment should be a brief introduction to understand why the routine exists and what it can do.
d. If your comments explaining a function are only a few words long, ask yourself if you can better name the function and avoid the comment itself.
e. Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.
f. Use comments to explain the intent of the code.

Format

Why is formatting important? Because proper formatting makes your code stand out. It appears modular, consistent and logical.

Well-structured and formatted code looks clean and is easier to read, for you as well as for any developer that works on your code in the future. A readable code is one which can be easily maintained.

There are simple steps that can be followed in order to achieve consistent formatting.

a. Try to have a standard size for an indent, such as same spaces, and use it consistently. Align sections of code properly using the prescribed indentation.
b. Use a monospace font when publishing hard-copy versions of the source code
c. Apart for constants, which can be best expressed in all uppercase char-

acters with underscores, use mixed case instead of underscores to make names easier to read.

d. Use a standard bracket structure for code. There is usually 2 styles called normal brackets and Egyptian brackets

Normal

```
doSomething()
{
        Code
}
```

Egyptian

```
doSomething(){
        Code
}
```

e. Follow a maximum line length for comments and code to avoid scrolling the source code window and to allow proper presentation.

f. Use spaces before and after operators, doing so, does not alter the intent of the code

g. Proper indentation should be used. Without indenting, code becomes difficult to follow, such as:

```
If … Then
If … Then
…
Else
…
End If
Else
…
End If
```

However with indentation it is much easier to read. eg.

```
If … Then
        If … Then
                …
        Else
                …
        End If
```

Rather than a line of text, indentation makes the code more readable

```
Else
    …
End If
```

h. When writing SQL statements, use all uppercase for keywords and mixed case for database elements, such as tables, columns, and views.
i. Divide source code logically between physical files.
j. Break large, complex sections of code into smaller, comprehensible modules.

Version control

One of the best and easiest ways to collaborate in software development is to use version control. Learning how to effectively use version control from scratch before beginning a project will still end up saving more time than working on a project without it. Some advantages of using it are: -

a. With a version control, everybody on the team is able to work absolutely freely - on any file at any time. All the changes can be merged with common version using Version Control System. There's no question where the latest version of a file or the whole project is. It's in a common, central place: your version control system.
b. You can have multiple versions of the same project. Each version will be well documented and at any time if you decide to use a previous version, it is easy to switch.
c. Similarly, there exist different versions for each file. If you ever find yourself stuck or if you made changes that you don't feel are necessary, you can rollback to any of the previously saved versions.
d. Whenever your save your modified code, your VCS requires you to provide a short description of what was changed. Additionally (if it's a code / text file), you can clearly see what has been changed in the file's content. This helps you understand how your project evolved between versions.
e. Distributed VCS is a type of VCS where every collaborator has a copy of the entire repository, complete with the entire history of the project. In the event that a teammate's copy or the central server's copy is wiped, all you have to do is send your copy, since it's the complete repository

Design patterns

In simple terms, a Design Pattern is a general purpose abstraction of a problem, which can be applied to a specific solution. Why reinvent the wheel?

In fact there can be use cases where following design patterns automatically achieve certain non functional goals in the requirement.

There are many reasons to use design patterns when applicable in programs, such as: -

a. Design Patterns give a software developer, an array of tried and tested solutions to common problems, thus reducing the technical risk to the project by not having to deploy a new and untested design.
b. They are language neutral and thus can be applied to any language that supports object-orientation.
c. They aid communication by the very fact that they are well documented and can be researched if that is not the case.
d. They have a proven track record as they are already widely used and thus reduce the technical risk to the project.
e. They are highly flexible and scalable and can be used in practically any type of domain or application.

Design Patterns, despite their initial learning curve, are a very worthwhile investment. They will enable you to implement tried and tested solutions to problems, thus saving time and effort during the implementation stage of the software development lifecycle. By using well understood and documented solutions, the final product will have a much higher degree of comprehension. If the solution is easier to comprehend, then by extension, it will also be easier to maintain.

Code respositories are essential for version control

These are just some of the many practices and principles for good programming. Writing simple, readable and easily extendable code goes a long way in effectiveness, often as important than the actual solution itself. There is seldom code written by one person. d

BEST PRACTICES FOR TESTING

The stage where all the hard work is put to the test - the practices to be followed are of paramount importance

Testing methodologies

Testing methodologies are approaches and strategies that are used to test a product to ascertain if it is fit for deployment.

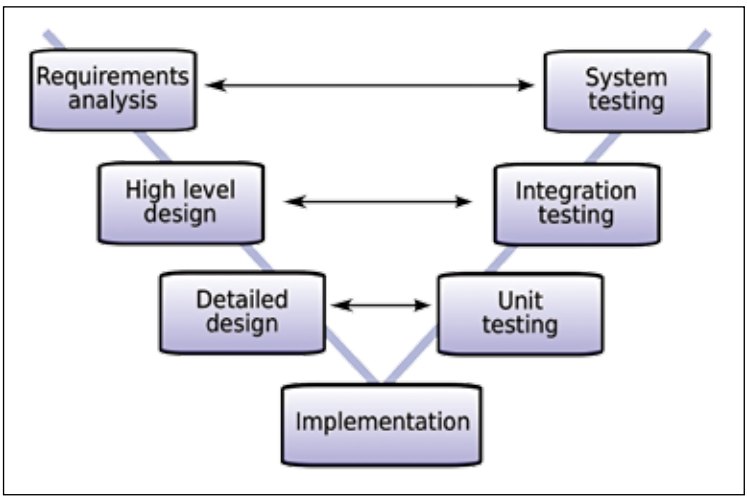
It involves making sure that the product works in accordance with its specifications, and has no undesirable side effects when utilized in ways outside of its design parameters.

Software testing methodologies include everything right from unit

testing which is testing each individual component or module, integration testing an entire system and other specialized forms for security and performance testing.

As software applications get more and more complex and associated with multiple other systems and devices, the process of testing them and ensuring a high degree of quality calls for a robust testing methodology.

Test methodologies include Functional and Non-functional testing to validate the applications under test.



The V model - the relation between the initial stages and the testing stages in software development

Functional testing involves ensuring that the core features of the application (i.e. the business requisites) have expected behavior. Some of the ways this is done are:

- Unit testing.
- Integration testing
- System testing and acceptance testing.

Unit Testing

Unit testing is testing of an individual software module or component which makes up an application or software system. These tests are conventionally written by the developers of the module and in a test-driven-development methodology (such as Nimble, Scrum or XP).

Integration Testing

The objective of integration testing is to ensure that the components, while individually functioning correctly continue to do so when they are combined with each other to achieve the end requirement.

System Testing

This test is carried out by interfacing the hardware and software components of the entire system (that have been already unit tested and integration tested), and then testing it holistically.

Acceptance Testing

This test involves making sure that the product developed has met all the requirements and operates as desired by the end user.

Non-Functional testing

Another facet of testing is Non-functional testing, where the non-functional aspects are tested. These aspects are not directly related to the features of the program but might cause it to behave in an abnormal manner.

Non-functional testing involves testing performance, vulnerability to hacks and compatibility on a variety of environments.

Some of the techniques of testing non-functional requirements are:

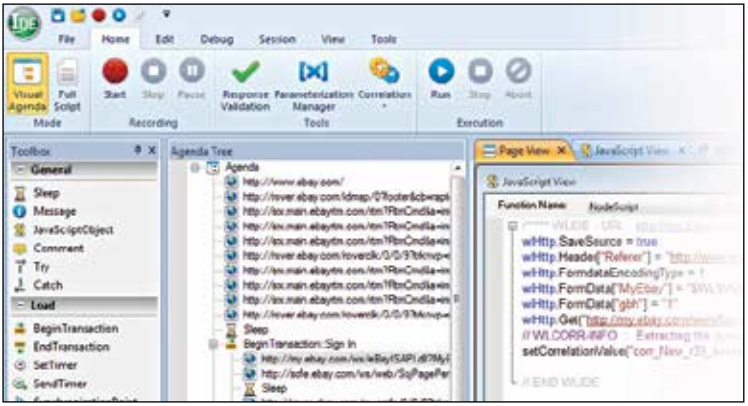
- Performance testing
- Security testing
- Usability testing
- Compatibility testing

Performance, Load, Stress Testing

There are several variants of performance testing in most testing methodologies, for example: performance testing is quantifying how a system behaves under an increasing load (both numbers of users and data volumes), load testing is verifying that the system can operate at the required replication times when subjected to its expected load, and stress testing is finding the failure point(s) in the system.

Below are some of the most widely used performance testing implements for quantifying web application performance and load stress capacity.

- WebLOAD
- LoadView
- Apache JMeter



WebLoad UI

Security, Vulnerability Testing

Application security is something that requires to be designed and developed at the same time as the desired functionality. Security testing tests for confidentiality, integrity, authentication, availability, and non-repudiation.

Usability Testing

This type of testing covers the ease with which a user can access the product forms the main testing point. Usability testing visually examines five aspects - learnability, efficiency, gratification, memorability, and errors.

Compatibility Testing

The compatibility part of a testing methodology tests that the product or application is compatible with all the designated operating systems, hardware platforms, web browsers, mobile devices, and other platforms.

Test Case

A test case is a set of conditions or variables using which a tester determines if the software or the application developed works properly as per the requirements and achieves the original objective that was required of it from the get go.

Structure of a good test case

While the template for a test case varies across organizations a good example of one such template would be one with most or all of the following fields:

1. Test suite ID: The ID of the suite to which a test case belongs.
2. Test Case ID
3. Test case summary: A summary or an objective of the test case.
4. Related requirement: ID of the requirement belonging to the test case.
5. Prerequisites: Conditions that must be fulfilled prior to execution.
6. Test Procedure: Step by step procedure to execute the test.
7. Test Data: Data or links to it, which are used while executing a test.
8. Expected result
9. Actual Result: To be filled post the execution of the test.
10. Status: Basic status is Pass or Fail.
11. Remarks: Any comments on the test case or after test execution.
12. Created By: Name of the author who created the test case.
13. Date of creation
14. Executed By: The person who executed the test case.
15. Date of execution
16. Test environment: The environment name in for execution

Writing good test cases

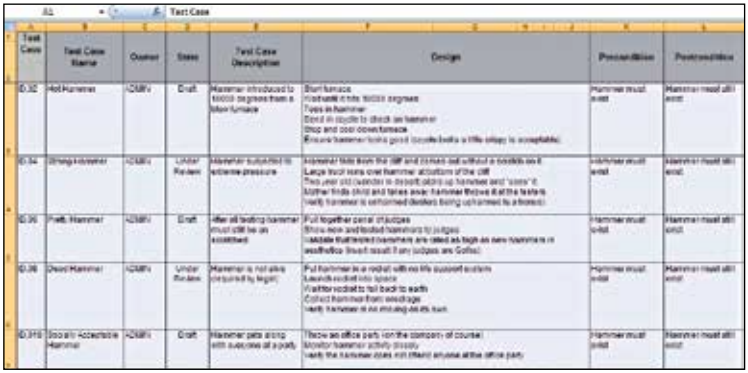
- Endeavor to make your test cases 'atomic'. Write test cases such that you test only one thing at a time. Do not overlap.
- Make sure that all scenarios are captured, including positive and negative.
- Do not use complicated jargons while writing a test case.
- Use the correct name (of fields, forms, etc.) keep it constant throughout.
- The test case should be accurate and to the point, and also reusable.
- Do not include unnecessary steps or words in the test case definition.
- The test case should be clearly traceable from the initial requirements.

Do's and Don't in Software Testing

Any testing process will commence with the planning of a test (Test Plan), building a strategy (How To Test), Preparation of test cases (What To Test), Execution of test cases (Testing) and culminate in reporting the results (Defects).

Do's:

- Ascertain that the testing activities are in sync with the test plan.
- Arrange training for technical areas where you might need assistance.
- Follow the standard testing strategies as identified in the test plan.
- Prefer to have a testing requirements document. This should ideally contain the details of the relinquishment that was made to QA for testing.

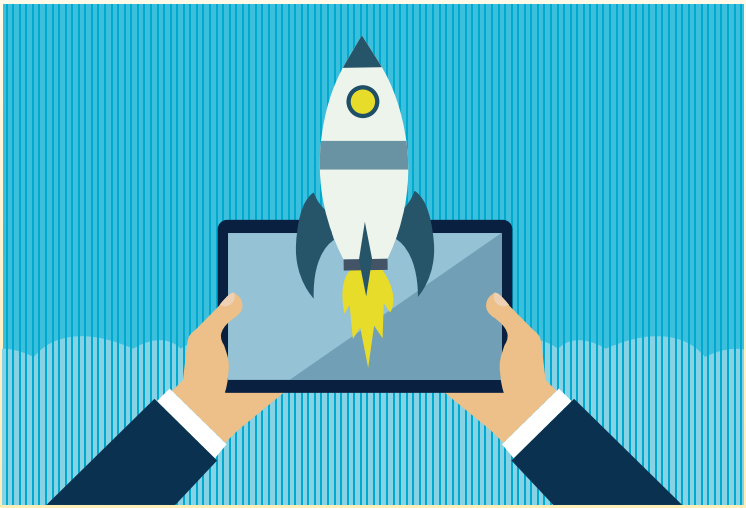


Test Case

- Some of the data to request are version of code to be tested, features that are a component of the requirement, features that are not a component of the requirement, incipient features and functionalities integrated
- Stick to the ingress and exit criteria for all the testing activities.
- Update the test results for the test cases as and when you execute them.
- Report the defects found in defect tracking implement.
- Take the code from the configuration management (as identified in the orchestration) for build and installation.
- Ascertain that the code is version controlled for every requirement.
- Relegate defects as P1/P2/P3/P4 or Critical/High/Medium/ Low.
- Perform a lucidity testing as and when the relinquishment is made.

Don'ts:

- Do not update the test cases while executing them. Track the changes and update based on an written reference (SRS or functional designation etc).
- Do not track defects in multiple places.
- Do not spend time test features that are not in the current release.
- Do not fail to document a defect even if it is identified as of low priority.
- Do not make assumptions while verifying the fine-tuned defects.
- Do not update the test cases in a hurry without running them on the theory that it worked in previous releases.
- Do not fixate on negative scenarios only, which are going to consume lots of time but in reality will be least utilized by end user.



BEST PRACTICES FOR DEPLOYMENT

Once your project is developed, it needs to be deployed to the live environment - and there are some rules to follow.

Deployment is an integral part of project planning and development. Basically it involves creation of release artifacts, which comprise of release documents and some installation documents. The deployment team installs, configures or deploys new version of code on production setup. Some bugs/defects can be found on production server which are suspected to be software level defects. They can be deployment issues, critical defects in application etc. which can lead

to overspending on project, project overrun, loss of reputation or failure to deliver product on time. The most inevitable part of software development is deployment. Automation can be used or implemented throughout the entire lifecycle. A process which needs no human intervention can be automated to avoid expensive resources doing repetitive and mechanical tasks so that end goal can be achieved quickly, accurately and consistently. Development workflow should include deployments as critical part. Workflow usually includes at least 3 environments:

1. Development
2. Staging
3. Production

Developers works on their own separate branch for developing any feature or to fix any bug. Minor fixes can be directly fixed on stable development branch (trunk). Once the feature is implemented, the code is merged into the staging branch so that it can be deployed on staging server for testing and quality assurance. After testing phase, feature code is merged into development branch. Then on releasing new feature the feature branch is then merged to the production so that new feature can be deployed on production servers.

Testing is also a crucial part of deployment lifecycle

Development Environment:

Every developer has their own local setup for working on fixes or features. Some companies have development environments configured for automatic build deployment for every commit or push or whenever required. This

can be treated as an advantage as it reduces time to deploy new build on the server. Every small change must be deployed after it is committed or pushed. If the change was made by mistake then an older version is restored in the repository.

Staging Environment:

Features considered fairly stable are merged into staging branch which can be automatically deployed on staging environment as required. Various types of testing is performed to ensure stability of the software. It is very helpful to have separate branch known as staging to have staging environment associated to it.

It helps developers to have multiple branches with different features that can be deployed to the same server simultaneously, simply by syncing/merging code with the local branch that is required on staging environment.

Production Environment:

After successful completion of testing phase, new code base can be deployed on production environment. It should be merged into a stable/properly working branch if the feature(s) was implemented in some other branch. Incomplete features, verbose logging or debugger breakpoints can be identified by taking diff between the production and development branch. Once the diff review is completed, new version of code can be deployed on the production environment. After successful deployment on production server, proper automation process should be implemented to check if all the features of the fix are working properly.

Rollback Strategies:

Sometimes things break at the time of deployment on any server. So rollback is required for maintaining business flow. Issues in rollback will wreak more havoc than the issue due to which rollback is required. If defect is related to code then files can be restored from the backup location. Usually a release introduces some changes or new database structure that is incompatible with the old release. In this case, the application will break after rollback. Proper database backup is restored for proper functioning of application. Backup policy should be implemented to ensure recovery of file to a last known working stage. Normal deployment mechanism is used for rollback also to avoid any cluttered production boxes. Maintaining database and configuration management helps to rollback application smoothly.

For weekly, or eventually daily updates, automated deployment is the way to go

Automated deployment reduces risk of failures and delay. Automation helps in CICD i.e Continuous Integration and Continuous Delivery with no or minimal deployment. It forms two copies of database, one can be used for deployment and other as backup incase of failure.

As lucrative as CICD sounds, achieving it takes a great deal of effort and technological expertise, which is really fed towards a non-functional requirement that in most cases analysts and / or clients do not fully understand. Thus automated deployment remains a low priority during the development of an application especially in its initial stages. In many cases it is observed that overtime the application achieves such critical dependency or market volume that improvements and features need to be continuously streamed as minor or major releases. An example of this would be the Facebook app which sends out regular updates on bug fixes and even new features. Without an automated deployment mechanism in place, achieving this continuous stream of releases is a very difficult task, as usually releases are bulky and very procedural in nature. Thus, if at some point in the future we see our application graduating to frequent releases in the period of weeks or even days then it might be worth spending the extra effort to establish an automated deployment model right at the beginning.

Automated deployment process:

Getting started on ADP helps reduce time and costs and offers improved testability of software application with improved auditing. To manage

complexity of deploying new version of code on production environment is a difficult task which involves multiple processes and configurations.

Automation Advantages:

1. Repeatability
2. Time Saving
3. Quick Feedback
4. Change management

Practices of Automated Deployment:

Version is Bible

Most deployment processes include a built-in safety net, preventing out-of-process, locally generated artifacts from entering the production environment. Every change a developer makes must be committed to a source control repository which is the only source for the build process.

This insistence on having a single source of truth—and a reliable one—creates a stable foundation for all development processes. However, the processes are only as strong as their weakest link. That's why the first golden rule for effective continuous delivery is to version-control everything. The process of versioning should not only be limited to code but include configuration, scripts, databases, website html and even documentation.

Build Binaries Once

The final build may be a single file or a complex build with several different possible deployment versions, each build version should be created in exactly the same way and result in unique artifacts. These artifacts should then be only copied and not rebuilt again when moving between environments.

This one-time-only compiling greatly reduces the risk of untracked differences due to various deployment environments, third-party libraries or different compilation contexts or configurations that will result in an unstable release going out.

We should save the compilation phase output (the binaries) to a binary repository, from which the deployment process can retrieve the relevant artifacts.

Smoke Test

Smoke testing is a very rapid way to make sure that the most crucial functions of a program are working as expected. This is non-exhaustive software testing that does not provide the same level of depth as full test suites but

the upside is that it can be run frequently and quickly, often in a matter of minutes (rather than hours or days). This allows for a much quicker turnaround time on potentially time-consuming and critical issues.

Mimic Production

Each new deployment stage should be made into an environment that mimics the actual final production environment as closely as possible. This includes infrastructure, operating system, databases, patches, network, firewalls and configuration.

By validating new software changes in this kind of realistic pre-production environment, mismatches and last-minute surprises can be effectively eliminated and applications can be released safely to production with a higher assurance of success.

Dont forget the Database

Continuous delivery best practices would never be complete without advising that the database be subjected to the same basic quality protocols that ensure secure and reliable source code, task, configuration, build and deployment management.

However, many times this is neglected because the database is rather difficult to manage or incrementally grows by doing something as easy as copying files from one location to another. Yet, the database is actually a repository of the most valued and irreplaceable asset — our data — and preserving it accurately is imperative to continuous delivery.

Although the database brings with it several unique challenges, specialized database tools such as enforced database source control (for all environments), a database build automation tool along with database release and verification processes can ensure a stable resource in your DevOps chain.

Considering these continuous delivery best practices will allow you to create a deployment pipeline with increased productivity, faster time to market, reduced risk and increased quality.

Like all other stages mentioned before, deployment is an important part of running an application but the unique position of this stage (in effect the last one) also ensures that all the deliverables generated in the previous stages would only be delivered if the deployment stage completes successfully. Having a robust, fault tolerant and rapid deployment process is therefore paramount to achieve our final goal in software development: To get the application running in the production environment.

LANGUAGE SPECIFIC BEST PRACTICES

There are good habits specific to programming languages. We share some of the most popular ones here.

Best Practices in Java

Today Java is one of the most popular programming languages over the world. Main reason behind its popularity is its platform independent nature. Like one can write code, build it, compile it and can give it to another person with a different platform to run it. Java is also open source, and it has simple syntaxes making it easy to understand and easy to learn. Millions of developers choose Java as the programming language to build their career on.

Java is an object oriented language

There are some best practices that one can follow while writing code in Java.

Avoid creating unnecessary object: In Java, everything revolves around objects. But creation of object is the most expensive operations in terms of memory utilization as well performance. So, all the objects should be created or initialized only when it is actually required.

Instance field should be private: Fields declared within the classes are known as Instance fields. These instance fields of a class should always be private. And they should always be accessed within the class itself only. Instead, if they are declared as public, anyone outside that class can access these fields or modify the value. Hence the program is no longer secure and anybody can inject a bug. Best approach is to declare instance fields as private and add *getter* methods to access these.

Limit the scope of the local variable: A local variable is declared locally within a method or block of code. A local variable is of great use but most of the times the programmer unknowingly inserts bugs by reusing local variable many times. So, scope of a local variable should be minimized, that is, it should be declared just before its use. That makes the code more readable and less error prone. Also, local variable must be initialized while declaring. If not possible at least null value should be assigned.

Use primitive values instead of wrapper class: In Java, there are primitive values int, float, double and then there are wrapper class for them. It is

better to use primitive values as they are not nullable. This prevents many bugs. More over primitive values are quite faster than wrapper classes. Also, these wrapper classes are immutable hence they don't have setter method. Every time they need to be modified, a new object will be created. This will create unnecessary object resulting in memory wastage.

Use String wisely: Like wrapper classes, string is also immutable in Java. In any program string needs to be handled with utmost care as misuse of it directly affect the performance of the program. Whenever string object is needed, it is a better idea to instantiate the string directly rather than using an object of string.

Prefer Interfaces over Abstract class: Methods or in simple words functionality can be extended from classes. But Java does not support multiple inheritance by extending multiple class. So, it is better to use interfaces as it is possible to implements multiple interfaces. Again, it is easier to change the implementation of an existing class and add implementation of another interface rather than changing the full hierarchy of the class. But an interface should be used only when there is a clear idea which methods the interface will contain.

Return empty collection instead of null: All the methods return some value unless they are declared as void. The most important thing that needs to be taken care of here is that return type should never be null. The methods should at least return empty value. This will in turn save a lot of checking for null values.

Avoid Null Pointer Exception: Null pointer exceptions are quite common in Java. If a method is called on a null reference of an object this exception is thrown. Whenever it is thrown the program quits, which is unacceptable. For avoiding such scenarios first those object references that may contain null references need to be identified. And then you must put some null check so that they can be eliminated.

Avoid method (T...) signature: In Java var args method can be declared that accepts an Object... arguments. Here T can always be inferred as Objects. These methods cannot be overloaded type safely. So, it is better not to use it.

Proper resource handling: In Java, we can connect easily with external entities like Database and external file systems. But the common mistake most of the developers make is that they forget to close these connections or streams. If a database connection is not closed in a timely fashion, all the connections in the connection pool will be exhausted. Then no other thread will be able to connect to Database. Similarly, after accessing the external file

system, *close* method of the stream must be called. It will close the stream and release any system resources associated with it. If these resources will not be released, then it may cause memory leak.

Dilemma while choosing among Collection: One of the most common dilemma developers face is while deciding which data structure they need. All the developers need to do is understand the requirements thoroughly.

Typical Java code has quite a few best practices associated with it

Selection is solely depending on the requirements. For example, people often get confused whether they need to choose a list or a set. If they have to deal with duplicate values or if insertion order of the data is important, then they must go with list. But if they have only unique values and they need to make comparison between data sets then they must go with set.

Proper exception Handling: Most developers do not handle the exceptions thrown in a proper way. It is always better write custom exception handling according to the program. Also, don't put any code in finally block which itself may throw some exception. If it is unavoidable then the exception should be logged. Logging and throwing of exception must not be done together and java.lang.Exception should not be thrown directly.

Try to use Standard Library: It is better to use the standard libraries available for Java as they are already tested used by other people. It increases the performance as well as reduces the chance of adding bugs in the code.

Best Practices in Python

Python is a widely-used scripting language. It emphasizes the code readability and has a syntax that allows programmers to express the concept in fewer lines than other languages. Hence its popularity among developers is increasing day by day. While using Python for developments people should adhere to some basic rules.

Structure of the Repository: Most developers don't pay much attention to the structure of the repository. They just set up a code repository and some version control. Which is not right, as it is a crucial part of the architecture. Again, the layout is not the everything but it has its own importance. Whenever someone new lands on the project it becomes easier for them to go through the project.

In Python, according to Kenneth Reitz, the product owner of Python at Heroku, the following components should exist in the repository:

- README.rst
- LICENSE
- setup.py
- requirements.txt
- sample/__init__.py
- sample/core.py
- sample/helpers.py
- docs/conf.py
- docs/index.rst
- tests/test_basic.py
- tests/test_advanced.py

An organised repository increases understandability of the project

- License: It is next most important thing after the source code. This file should contain license data and also the copyright claims. There are websites like choosealicense.com that help you choose one.
- README: There should be a basic README that will contain project description, outline of the basic functionality and the purpose of project.
- The Actual Module: This is the heart of the project. If the project contains a single file, then it should be placed in the root directory.
- Setup.py: This is the setup script which describes module distribution of the project to the distutils so that things operate correctly on the modules.
- Requirements.txt: This is the optional file can be used to specify development modules and other dependencies required for running the project properly.

- Documentation: A well-structure documentation is the key component of any development project.
- Test: It is best to have 100% code coverage in any test plan.

Structure of Code: Structure of the code is the key component of a project. Some common mistakes made by developers are

- Multiple circular dependencies
- Strong code coupling
- Use of global state or context that can be modified by anyone
- Multiple nested if else clause or nested loops.
- Ravioli code

The only suggestion for the developer is to avoid stated scenarios.

Modules: Use of modules provides the main abstraction layer in Python. This layer separates code with similar data and functionality. All the variables, functions, classes confined within the module can be called through the module's namespace. This is one of the main advantage of using Python.

Packages: Developers should make use of the Python's straightforward packaging system. Packaging is a system where module mechanism is extended to a directory structure.

Avoid pure Object-oriented programming: Python is not a strict object oriented programming language like Java. It does not use those specific mechanisms of object oriented programming like class inheritance. In some scenarios, there may be a need of sticking some state and some functionality together. In short both the function oriented and object oriented can be useful depending on the scenarios.

Decorators: A decorator function works like a wrapper of a method or function. Decorated function simply replaces the original function. In Python functions are like first-class objects. So, by using @decorator tag a function can be easily declared as a decorator function. This mechanism can be used to separate external logic from the core logic of a function. For example, it can be used for memorizing or caching.

Context Managers: In Python, there is concept known as context manager. A context manager provides some extra information to a specific action. All this information takes the form of calling upon context initiation using *with* statement and calling upon completion of the code inside the *with* block. For example, the most well-known use of a context manager is to open a file. There are two ways for implementing this functionality

- using a class
- using a generator

Python emphasizes on code readability and an optimised syntax

As these two approaches appear almost same it becomes difficult to decide which one to follow. If some significant amount of logic is being used, then the class approach should be used. Otherwise function approach may be better for simpler situations.

Dynamic typing: Python is called dynamically typed language, that means types of variables used in Python depends on the value they contain. They do not have a specific type. In statically-typed languages, variables are a segment of the computer's memory. But in dynamically typed languages like Python they are more like 'tags' or 'names' that points to the objects. So, a single variable can be assigned multiple types of value. This feature may be one of the strengths or may turn to a weakness depending how it is utilised by the developers. Often, it leads to complexities and makes the code hard-to-debug. Some advice to avoid this issue:

- Same variable name should not be used for different things.
- Short functions or methods should be used for reducing the chance of using the same name for two different things.
- Different names should be used for things having a different type even though they are related.

Mutable and immutable types: Python has two kinds of built-in types – mutable and immutable. Mutable allows content modification. For example, lists are mutable. Immutable types have no method for modifying their

content. For example, suppose a variable is set a value. If someone tries to modify it, it will not replace but will create a new value and set it to another variable. Developers should make use of these variable wisely. String are also immutable in Python like Java.

Best Practices in PHP

PHP is a server side scripting language. PHP code can be embedded into HTML. It generally creates dynamic web page content or dynamic images used on websites. PHP also can be used for command-line scripting and graphical user interface (GUI) applications at client side. PHP is compatible with most of the web servers. Many popular website uses PHP. There are various reasons behind its popularity – it is open source, easy to learn, easy to use. Here are some PHP specific guidelines developers should follow.

Always use <?php?>: Many developer uses short tags like "<?" or "<%". All these tags are either deprecated or not official. It is better to write the proper tag rather than make the code incompatible with future version of PHP. Sometimes due to this, conflicts arise with XML parsers.

Indentation, White Spacing and Line Length: PHP developer should specifically consider these three. Some of the guidelines for these are:

- An indent should be of 4 spaces
- Tab should not be used.
- L line length should be less than 80 characters to make the code easily readable and debuggable.

Single Quoted vs. Doubles Quoted Strings: Developers need to understand the difference between the single quoted strings and the double quoted strings. Single quotes should be used if there is a simple string to display. If there are variables and special characters in the string, then they need to use double quotes. Then only those strings will be parsed by the PHP interpreter. Generally, it takes more execution time than simple strings.

Should not use functions inside loops: People often make the mistake of using functions inside loops. If there is a function inside the loop the function will be executed every time the loop will be executed. It will affect the performance of the program to a great extent. To avoid this, store the value returned by the function in a separate variable before the loop.

Using Single Quotes around Array Indexes: There is a difference between $array['quotes'] and $array[quotes]. PHP has a unique feature, that is it can use unquoted strings as an array. And the constant should be defined beforehand, otherwise a warning will be displaced.

Understanding Strings in a Better Way: PHP developer should have a proper knowledge of string manipulation. Because it is one of the error prone module can inject wide range of bugs.

Turning on Error Reporting for Development Purposes: There some errors which don't stop the application but inject some potential bugs. For handling this it is best to use error_reporting(), a function provided by PHP. It reduces the chance of reworking as the errors can be detected while developing the code. E_NOTICE, E_WARNING, E_PARSE are some levels of error reporting available in PHP. Developer can also use E_ALL for all kind of errors. This error_reporting() function should be disabled after development is done.

Should not put phpinfo() in the Root Directory: phpinfo is one of the unique features PHP has. If developers drop a file containing <? php phpinfo(); ?> somewhere onto the sever they can instantly learn almost everything about the server environment. One of the common mistakes made by the developers is to place this file in the Webroot of the server. This can open many security loopholes. So, it is best avoided.

Befriend the PHP Manual: It is always good to follow the PHP manual. The PHP manual has comments following each article and thorough description of every module. It is not only the best guide for beginners, it also has answers for almost every question PHP developers ask.

Proper knowledge of the PHP manual can save a lot of time

Use an IDE: Some of the well-known IDE's for PHP are NetBeans, phpDesigner, phpStorm, PHP Introduction and Resources.

Use a PHP Framework: A lot can be learned about PHP by experimenting with PHP frameworks. It helps PHP developers to grow themselves. Most of the PHP frameworks are designed on the basis of Model-View Controller architecture. Some of the best PHP frameworks are CakePHP, CodeIgniter, Zend, Symfony.

Use Objects (or OOP): Objects are the key points of Object-oriented programming (OOP). Some key features of OOP are

- Abstract data types and information hiding
- Inheritance
- Polymorphism

All these can be done in PHP. Developer should try to achieve this for breaking their code into separate sections based on the logic and reducing redundancy.

Should handle all possible scenarios: If some application has user interaction, then it is taking some input from the user. In such a situation, a developer should cover all the cases including the negative scenarios. For example, if a form in PHP is taking username and password from user, user may put wrong data or wrong type of data. For that, a validation procedure should be there in the code.

All these set of informal rules are learned over time for improving software quality. It is solely depending on the developers how they will make use of these guidelines. It has been consistently observed that when someone follows these guidelines properly the numbers of bugs found in that code reduces significantly. Also, it makes the code easy-to-debug. Finally, we can conclude that superior coding techniques and programming practices along with the technical knowledge are the key to success in this world of software development.

TOOL SPECIFIC BEST PRACTICES

There are many tools and processes out there that can amplify your software development in many ways

Throughout the software development process we use and rely on a number of tools to eliminate or reduce effort in doing a wide variety of tasks which ultimately increase the net productivity by allowing us to focus on the application itself.

These tools are including but not limited to UML design tools, Issue/User Stories database, source code repository,Integrated Development Environments (IDEs), database browsers, deployment automation tools

or even word processing and data processing applications. The sole objective of using these is to reduce time and effort in doing non-functional tasks and concentrate on the core application. A lot of these tools are heavily misused or not used to their full potential primarily because we do not very easily see them directly as important to our application. This is a significant mistake as often on closer and more detailed inspection it is observed that had the supporting tools of the application been used in its intended manner the resulting error might have been easily detected or not have occurred in the first place.

Dos and Dont's for tools used in Software Development

UML Design tool

As obvious from its name, these tools are mostly used in the design phase to form the structure of the entire application using Unified Modelling Language to depict all its individual components, cluster them into modules, define the relationships between these modules clearly and expose all of its APIs in a neat and readable manner.

Be Concise

Large diagrams that contain tons of elements actually tend to convey less information than smaller focussed diagrams. When reading a large diagram your intended audience will be distracted and not know what to focus on, and since there's too much on there, they will quickly give up trying altogether

A minimalist UML diagram is what you should put on your design review sheet. As a simple rule of thumb one can check if they are able to print the diagram on a single A4 sheet while keeping things readable.

Never Cross

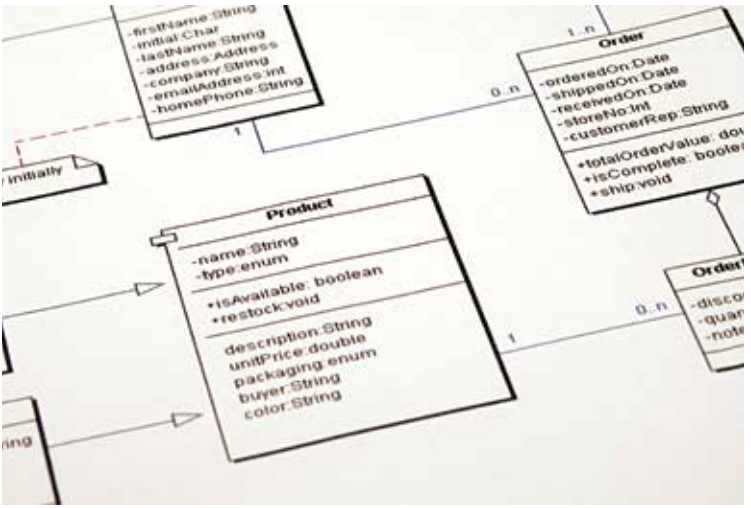
Always try to avoid two lines crossing each other in your diagram. Uncrossing lines in a diagram makes it a lot more readable and understandable. More importantly, If you are unable to uncross the lines on the diagram then it is possible that:

a) there's too much content on the diagram.
b) there is some kind of design flaw in your model.

Parents Up

While drawing hierarchies on a diagram make sure to place the parent elements higher than the child elements so the arrows always point upwards.

This rule is pretty intuitive for most UML modellers, as it mimics the natural age relationship from parent to child. In case the diagram has multiple elements all descending from the same parent, it is recommendable to show the hierarchy in a vertical tree style.



The hierarchy between parent and child elements has to be clearly indicated

Issues trackers and User stories databases

The bible for most project managers and requirement analysis teams, issue trackers and user story trackers are basically a compendium of the list of undergoing development efforts including their scope and requirement criteria as well as a list of open issues that need to be fixed. When used effectively, these are very powerful tools that can contribute a lot to the development cycle by streamlining the process of requirement analysis to design as well as planning right at the beginning itself. That is a significant reason behind their popularity.

Using these tools project managers may plan the roadmap of the development of an application, assign work modules to respective teams, keep track of the progress made on those and quickly raise and notify about detected issues or requirement changes that need to be handled.

Some of the popular tools that do this are JIRA, Trello, Asana and GitHub Issues. Some guidelines around the effective and optimal use of these tools are mentioned below:

Minimal creation forms

When creating an issue or a story on the tracker board, care should be taken to not make too many fields as mandatory or spread the parameters across the form. This might take up valuable user time and discourage creation of more of them. If any field is semi important, or makes sense at a later stage in the workflow, they can be made mandatory for that transition and not during the creation of the issue. Doing this will definitely increase the number of documents that are created on the tracker board and ensure that most of these are logged in the tool.

Automate things!

As with all things good, automation is key to maintaining a healthy state of the tracker data. Periodically cleaning, archiving data based on key events using a REST API or otherwise will not only result in a cleaner dashboard but ensure that most of these data are also automatically saved at a different location and can be easily retrieved when needed.

User stories are a popular method of tracking progress on projects

Source code repositories

Source code repositories as their names suggest are tools that store and version the source code of the application in a clear manner and also offer a number of features to mark, copy or revert to each version of the code.

A source code repository is essential to most applications that have a sizable codebase and have a number of team members working collaboratively on that codebase.

These repositories also come with features to ensure that two or more changes that do not match with each other are not allowed in, thus preventing the accidental overwriting of one developer's code by another. Source code repositories also often feature integration modules with IDEs and deployment tools to allow seamless inflow and outflow of code. While writing to a source code repository might seem simple, there are a few standard practices to follow. Some examples of popular source code repositories are Git, Subversion, Mercurial etc.

If it's not in source control, it doesn't exist

There is a common mantra that says: "The only measure of progress is working code in source control". As established earlier, source repositories are the one true source of code truth and unless the code you write is not part of it – it simply doesn't exist.

Commit early, commit often and don't spare the horses

Extending the previous point, the surest way to avoid "ghost code" – that which only you can see on your local machine – is to get it into the repository as early and as often as possible. The early and often approach achieves a few other things as well, which can make a significant difference:

1. Every committed revision gives you a rollback marker. If anything goes wrong, are you rolling back one hour worth of changes or one week?
2. The risk of a merge conflict nightmare increases dramatically with time. When you've not committed code for days and you suddenly realise you've got 50 conflicts with other people's changes, it won't be cool.

When you work this way, your commit history starts to resemble a regular pattern of multiple commits each work day. Of course it's not always going to be the same but the overall objective is achieved. However, when in an entire project there are entire days or even multiple days where nothing is happening, it might be reason to get worried. Often development is happening in a very "do-it-all" sort of way or absolutely nothing of value is happening at all because developers are stuck somewhere. Either way, something is wrong and source control can be a very useful indicator to let you know.

Always check your changes

Committing code into source control is easy but do too much of it carelessly and what you end up with is changes and files being committed without care!

One doesn't even need to be told how popular Github is

Remember the axe-murderer when writing commit messages

There's an old adage that says :"Write every commit message like the next person who reads it is an axe-wielding maniac who knows where you live". If I was that maniac and I'm delving through reams of your code trying to track down a bug and all I can understand from your commit message is "updated some codes", I will find you.

The whole point of commit messages is to explain why you committed the code. Maybe something was broken in the jdbc pool. Maybe the customer didn't like the layout of the page. Maybe you're just tweaking the build configuration to optimise performance. Whatever it is, there's a reason for it and you need to log this in while committing the code.

Nothing but code belongs in source control

A easy way to say this would be: "Nothing that is automatically generated as a result of building your project should be in source control." It may break things as your local settings , credentials etc might get overwritten on all other peoples machines and this will not work until they revert those changes back and then recompile. And then once they do recompile and recommit, they start sending the files in opposite direction and this time you're the one who has the problem.

Dependencies need a home too

This might be the last of the guidelines but it's a really, really impor-

tant one. Whenever an application has external dependencies which are required for it to successfully build and run, they need to be part of the source control! Otherwise those dependencies would be missing from the project source of all other members and their builds and runs will fail. This can also break deployment automations and test suites. Usually a dependency management tool like maven or gradle is used as an easier alternative to committing all dependencies to project source repo.

Integrated Development Environments (IDEs)

The bread and butter (or the knife really) of almost every developer involved in designing an application, Integrated Development Environments or IDEs as they are commonly known are powerful tools that act as a centralized one stop solution for coding requirements. They feature a language specific editor, function listing, project structure navigation, allow programs to run within their scope, as well as support multiple plug-ins to connect to source code repositories, databases or even issue tracker software. While the choice of a good repository would depend on the kind of architecture the application has, the choice of programming language used as well as the developer's preference there are some common tips around the best use of these tools to achieve better results.

Use Integrations

The objective of using an IDE first and foremost is the ease with which it allows the seamless integration of different components of the development, compilation, testing and debugging cycle. Using and leveraging these integrations is then the primary guideline while using IDEs yet surprisingly a large number of developers tend to only use an IDE to write and compile code (sometimes even compilation is done externally) while testing and debugging are done externally via other tools or command line. This actually defeats the purpose of using the IDE. We should always strive to setup and use the most number of possible integrations from within the IDE itself. These may include source code repository integration, debugging from an embedded server or virtual machine etc.

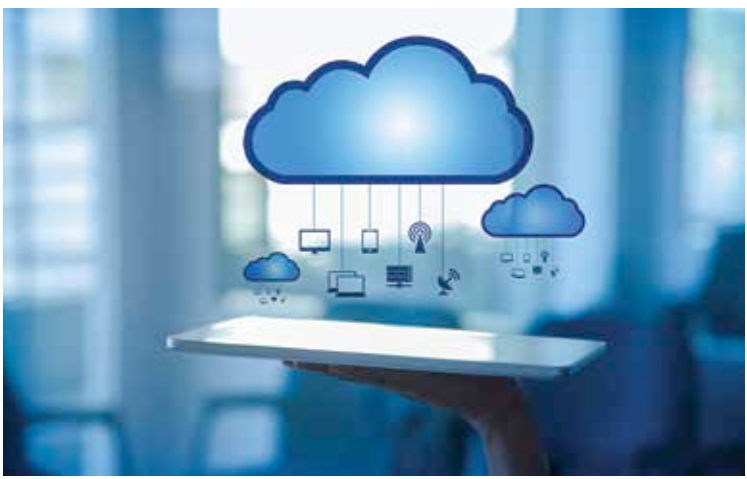
Save actions on editors

Ever had an unnecessary import dangling in your java file? These unnecessary imports slow down the application. Often when we are working on a function we import some dependency that we later realize we do not

need but fail to remove it, would it not be great if it could be magically removed? Well it can! Simply use save actions on the editor of your language to specify removal of imports that are unused and every time you save your source code file, the IDE will smartly detect and remove all unused imports. How about indentation? Tabbing out the if else blocks is not only tiresome but unnecessary. A simple save action setting will automatically style the entire source file according to the required specification of the style xml specified.

Don't jump to the cloud

While cloud based IDEs are all the rage today, with multiple IDE vendors launching their own variants of a cloud based IDE (Eclipse Che, Cloud9 etc) the standalone IDE is not yet a fossil. There are distinct advantages to using a local IDE over a cloud one. While the cloud one indeed offers



Cloud based IDEs aren't as great as they sound

great features like increased accessibility, real time file sharing and cloud based debugging and testing the biggest issue is perhaps the network lag involved, when running large scale applications the network lag can become significant (based on connection quality) and hinder the rapid development of the application.

Whereas on a local IDE the application would run much faster and even allow custom settings and offline development that would not be available in a cloud based IDE. Finally, some company guidelines strictly restrict the

transfer of proprietary company code from within their physical premises, which would not be possible with a cloud based IDE.

Deployment Automation Tools

Deployment automation tools like Jenkins or Ansible help in complete automation of the process of deployment of the application once it has been successfully built. It involves moving the release artefact(s) across environments along with configuration changes, database schema changes, firewalls, proxies and all documentation.

Using deployment tools greatly reduce our efforts during application releases but care should be taken to ensure strict adherence to standards and guidelines without which a lot of issues can crop up in the final stage of the software development process.

Some of the best practices of using deployment automation tools are:

Configuration management

We should aim to use a Configuration management database along with our deployment tool to ensure that the configuration data is correctly persisted, audited and versioned every time there is a change in them. This will help in quickly reverting to a 'happy form' state when something goes wrong in our deployment and we want to revert the application to a previous working state.

Use specific infrastructure

While it is a good practice to always deploy code in production like environment to test against realistic cases, sometimes special use cases like regression tests require infrastructure with higher capabilities. The deployment tool must be configured and profiled in a way that the deployment is done on specific machines based on the use cases and these builds are monitored for performance.

CASE STUDY: SOLUTIONS

We provide our own solutions to the case studies presented in Chapter 2

Now that we have looked at some of the best practices in the areas of design, coding, testing and deployment it might be interesting to look back at our case studies from Chapter 2 to theorize how things might have gone differently (or not) had the teams responsible followed these practices.

Please note that this is only an academic exercise and nowhere do we claim to solve or state with certainty how any of these serious incidents could've been averted. The objective of this Chapter is to take our newfound understanding of some of the best practices in the industry and extrapolate on the impact that they might have had on the cases discussed earlier.

The below section is strictly a point of view based on the facts of the case, feel free to disagree or counter the opinion (in fact, that can be a rather interesting conversation) and write to us on the same.

Case 1: Mars orbiter crash

The first case that we mentioned was the crash of the $100 million mars climate orbiter (MCO) due to a simple confusion of units.

The MCO as you will recall had set orbit at 57 kms from the surface of mars instead of the intended much higher 145 kms and was destroyed due to the increased atmospheric friction.

Good documentation could've averted this crisis

This was the result of engineers using imperial units instead of standard ones.

The question here is why did the engineers choose to use different metrics for the calculation? Turns out they were US-based engineers who were usually using imperial units for measurement.

Would this still have happened if the requirement document clearly mentioned which units to follow in the first phase itself? Would it have been avoided if the requirements thus captured were converted into a set of test cases that specifically checked for this condition? A simple practice of having the requirements of the project clearly and objectively mentioned

in the requirement analysis phase which can then be transformed into a highly granular design model might have averted the situation altogether.

The additional effort in granular and exhaustive definition of requirements is thus clearly worth the effort if one is trying to do something as monumental as check the weather on Mars. A comprehensive suite of test cases that cover all edge scenarios might also have detected this flaw before it ever reached production.

Case 2: AT&T communications go down

While at first glance the AT&T fiasco might look like a technical snag or a quality issue, it is also a result of design flaw within the application. The race situation that caused the self-propagating disaster resulting in the shutdown of half of the switches in USA should have been identified as a concurrency point right at the design phase, which could then ideally have been handled in a desired manner using locks.

Designing an application structure to granular detail, including the technical stack and interaction diagram is key to developing it.

As mentioned in the best practices for design chapter, a good and effective design diagram can automatically detect issues like synchronization or unnecessary components.

These can then be handled via workarounds of design without writing a single line of code yet!

Case 3: Passenger plane shot down by USS Vincennes

Perhaps the most unfortunate of all the cases, a civilian plane was shot down by the US Navy in 1988 mainly due to a poorly designed User Interface. The UI of the Aegis system was too cluttered and misleading to deduce information rapidly and hence prone to misinterpretation as seen in this case. A clear and well designed UI is very critical in retrieving and delivering information to users. Often low prioritized, the quality of the UI component specifically for customer facing applications can literally make or break them.

While we see here that the information collected by the Aegis system was 100% correct, representing it in a poor manner resulted in the operator misunderstanding the content, which in turn cascaded into this disaster.

USS Vincennes

A well design and thoughtful UI would have easily solved this issue. In fact one report says that one of the engineers in the Aegis software system even suggested a few UI changes but was dismissed by the logic that they would only deliver on what the Navy required and nothing more.

A clear and concise UI is an implicit requirement of design and need not be explicitly mentioned to be implemented.

Case 4: Scud missile hits american barracks

In another mishap that resulted in the loss of human life, a scud missile hit the American barracks without being challenged at all by the ultra-modern Patriot defense system put in place. The root of this was a time delay error that was caused due to a combination of exception situations. The running time was exceeded and there was also a floating

Yes, that's a SCUD missile

point error in the calculation of range that kept compounding over time. Deploying the software in a near-production like environment and stress testing it over longer-than-usual running times might have exposed the rounding off error before it had a chance to cause real harm.

An exhaustive automated testing suite as well as better deployment practices would have gone a long way to reduce the count of such errors. Another important facet of this is how non-functional needs (like logging floating point operation errors) though seemingly useless can sometimes be of immense help in identifying issues within the code base.

Case 5: NORAD reports U.S. was under missile attack (1980)

Thank you for not starting WW3

A third military goof up in the US (a bit of a pattern during that period) occurred in 1980 when a series of incorrect missile attack notifications were raised by the NORAD computers. What was even more shocking however is that this incident was reported to be "not a rare occurrence" and the defense alerts were usually verified with "human elements" of the system according to the then chief security adviser. Unfortunately the exact cause of the above case was never revealed, only that a single 46 cent chip was responsible for it. Experts have expressed skepticism regarding the validity of this explanation given another statement by the same security adviser where he said that the fault was "never reproduced during tests". Without a concrete root cause, diagnosing and consequently attempting a fix for the problem would be futile, thus we are not going to attempt that on this case study.

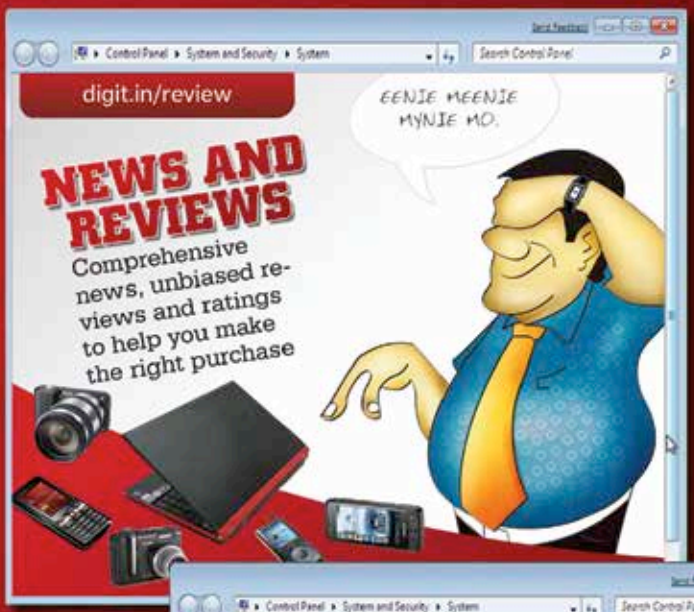
Case 6: Man deletes entire company with single command (2016)

The latest of all the case studies, this story was the subject of much hype and speculation where a man had claimed to have accidentally deleted all data from his web hosting server including his client's. Later on though he posted back saying it was a joke but was based on an actual experience of a client's. This case is of specific significance here because if true, regardless of the source

of the error it presents a unique opportunity to identify the numerous design, development and specially deployment flaws that have coupled together to create this nightmare. First and foremost, a script to remove the contents of a folder using the 'rm' command without taking a backup to an offsite location, especially in a production environment is careless to the point of being dumb. Next, any script, maintenance, release, patch or testing needs to go through design,develop and test phases (yes even shell scripts for deployment) without which we will never be able to guarantee its behaviour.

Thirdly, there are perhaps hundreds of other ways of cleaning up a folder location than using a combination of variables (that can potentially have other values associated to them) and a rm -rf command.

The fact that such a situation can occur in today's age of deployment automation is cringe-worthy and perhaps the forum was right when it collectively advised the person who posted this to start looking for a lawyer as he really had 'killed' his company with a single command.



All this and more in the
world of Technology

VISIT NOW digit.in

RNI REGN. NO. DELENG/2001/03791