CS427 Book Report: **The Pragmatic Programmer**

Submitted by: Sugandha
NetID: sugandh2

‘The Pragmatic Programmer’ makes for a very interesting read. It is for anybody who wants to improve their programming skills.
The book gives tips and tricks becoming a pragmatic programmer, i.e. developing these characteristics: early adopter (wants to try new things), inquisitive (questioning rather than passively accepting), realist and jack of all trades. A pragmatic programmer also cares and thinks about his work.
The authors believe that individual pragmatists bring their identity into large software development teams and over the years lead to development of code and processes followed by the team.

A Pragmatic Philosophy

The authors have listed some ways to achieve pragmatic characteristic in a very engaging way by drawing parallels with folklores and stories.

1. *Responsibility* (The cat ate my source code)
   A pragmatic programmer takes responsibility for his own code, and anything that goes wrong with it. Instead of making excuses, we must focus on the options or alternatives if things go wrong. Moreover, we must always have a contingency plan to fall back upon. For example, take back ups, keep a copy of original source code before making changes.
   Example We learnt from our MPs that we must always compile and run code in our local machine before committing it to repository.
   Another method that we followed in my previous job was to code directly on the application server machine so that all code is consistent and at one place, and a programmer can be sure that he is always working on the latest copy.

2. *Don’t let software rot start* (Software Entropy)
   Just like serious crime begins with a ‘broken window’ in a neighborhood, software decay begins with small flaws that can eventually take down the whole software. A pragmatic programmer gets cleaning as soon as he sees the first signs of a broken glass.
   Examples of these are constantly refactoring and cleaning code to make it maintainable.

3. *Synergize for change and look at the big picture* (Stone soup and boiling frogs stories)
   Instead of trying to convince people to do things in a certain way, a pragmatic programmer starts developing and gets people to join in once a part of it is successful. This cuts down the otherwise probable bureaucracy and disagreements. Moreover, a pragmatic programmer is not like a frog that can be put in gradually heating water till it gets cooked. He keeps an eye on changes and deviations from design before it’s too late.

4. *Quality required* (Good enough software)

A pragmatic programmer lets the users decide the quality expected from the software. He considers constraints like user requirements, promised schedules, and his company cash flow and delivers accordingly. This in no way means delivering poor software, it simply means involving customers in decisions about quality-related trade-offs.
For example, I worked on a Sustenance Engineering team in Oracle Corp for a couple of months and had a chance to observe their bug fix processes. We were working on a product called Retail Merchandising System. Whereas the forward ported bugs to be fixed in new releases were subject to a lot of testing and reviews, urgent customer bugs were often not reviewed too harshly for things not related directly to the bug (like a spelling/grammar mistake in an error message).
Similarly, if in our class projects, we are not able to implement all the features according to schedule, we have to talk to our teammate in customer role and take a decision about what features are more important from customer perspective, rather than being stuck on implementing all features perfectly and later not being able to submit our project on time.
Customer needs to be included in the decision about quality-accuracy tradeoff in software.

5. *Invest in knowledge* (Your Knowledge Portfolio)
A pragmatic programmer makes a habit of learning new things. Since technologies change rapidly, our skills need to keep the same pace, which means learning lots of new and different things every day. Based on the technology trends, we need to keep adjusting our skills and knowledge base. A no-so-popular technology today might be tomorrow's hot thing and the earlier we adopt it, more handsome the rewards will be. To do this, we can keep some specific goals in mind, like taking one training session every 6 months, learning a new language every year, reading books and technology magazines, taking part in user groups etc. This learning should be taken as a challenge to the self. Critical thinking is also important in this knowledge-gaining process.
Example In many organizations, including the one where I worked earlier, employees are sent for technical and soft skill trainings every six months. This is beneficial for the company in terms of increased efficiency, as well as employees' personal growth. Learning new technologies and shifting to projects requiring broader skill sets helps to keep employees on their toes. This also reduces attrition rate for the company by keeping employees motivated and away from boredom.

6. Communicate
Effective communication is very important because a developer has to communicate in many ways for his work. He must interact with users to know their requirements, with his co-workers, and through documentation. We must understand the audience before communicating, and plan the conversation before speaking. Communication should be interactive and it is important to be a good listener as well. Furthermore, presentation is as important as the content.
For example, we noticed differences in communication and coordination patterns when our team size increased from 3(for course MPs) to 6 (for project).We switched from using text messaging to decide when to meet, to more formal ways like e-mails and doodle.com.

A Pragmatic Approach

In this section, the authors discuss some ways to make our code cleaner, efficient and more maintainable. They suggest the following strategies for it:

1. Avoiding code duplication

   Similar code may exist in various parts of a program. It is best to avoid duplication, because changes made later in the system will also need to be duplicated, and the chances of introducing errors increases.

   Code duplication can be because of various reasons. Some of these are:

   - *Imposed Duplication*: Where the coder doesn't have an option but to duplicate code, because of the design document/ programming language/ platform being used. Imposed duplication can be eliminated by using a meta-data based code generator that generates the same code in different languages.

     Another way to remove duplication is by making the code explanatory enough so that low-level comments explaining the code in detail are not required, as these will also need to be changed every time the code is changed.

     Example: One of the ways that we have been taught to do this in CS427 is the use of self-explanatory variable names in refactoring. This avoids the need for having comments to describe what every single variable does, and having to modify these comments in case of code changes.

     Other types of imposed duplications are the duplication of design specifications in code, and language dependent duplication. Duplication in design documents can be used to advantage by generating tests based on specifications, to ensure that code changes correspond to specification changes.
   - *Inadvertent Duplication*: There might be cases where the technical design document specifies duplication. The solution for this is to normalize the design so that each concept is represented by a different object, and relationships between them define interdependence (i.e. object oriented coding). Similarly, normalization in databases can also be done by following the established normalization techniques.
   - *Impatient Duplication*: Many times, programmers duplicate code because they find it easier to copy and change a part of code, rather than changing existing code so that it takes care of the new scenario as well. This laziness can lead to big problems later during maintenance.
   - *Inter-developer Duplication*: Many times developers working on different modules duplicate code and functionality. This kind of duplication is hard to detect and handle. It is best avoided by using clear design documents, proper project management and encouraging inter developer communication through forums and code reviews.

2. Orthogonality

   Modules should be orthogonal, i.e. independent to as much extent as possible. This reduces coupling and makes code changes at a later stage easy.

   Orthogonality increases productivity since functionality is well-divided to be localized and coding and testing becomes faster. It also promotes reuse by having self-contained

independent modules with their well-defined functions. It also reduces risk since most errors are generally localized and are easy to test for and remove.
Orthogonal designs can be implemented using the following ways:

- Project teams that have well-divided responsibilities where each member is responsible for an orthogonal component in an area.
- Using a layered architecture, with layers only abstractly dependent on underlying layers.
  Example MVC architecture. I was able to appreciate the real utility of this architecture only after I worked on Oracle ADF (Application Development Framework) in my last job. I was writing code for an application to manipulate data in a database which would later be replaced by the sales team database, and the application itself was to be replaced by a newer version with more functionality in the long term. So, no dependency between model and view could be afforded.
  The design principles for decoupling layers is also helping me in my current part time job at the University Archives where I am developing a Ruby on Rails application which is to replace a PHP application running on the same database.

- Using tools and technologies like Aspect Oriented Programming (AOP)
- Coding techniques like using design patterns, avoiding global variables, writing closed modules.

Time taken to write unit tests and number of bugs found(along with how easily they can be solved) are some ways of verifying Orthogonality in design. Documentation can also reflect the degree of orthogonality.

3. Reversability
   A pragmatic programmer always codes keeping reversibility in mind. A design change can happen at any stage during development, and the code should be flexible enough to handle it. This means not having tight coupling with platform or underlying database. Technologies such as CORBA architecture are helpful in this regard.

4. Tracer Bullets
   Tracers bullets are an alternative to heavy designing and calculations that go into building a new system. Tracer bullet codes form a lean skeleton of any new system that represents how the system behaves as a whole and the basic functionality. This can be used to demonstrate working of the final system, and adjustments can be made if the customer is not satisfied. The advantage of using tracer bullets is that the code represents the basic functionality and can be changed easily since it is very simple at this stage. The tracer code written differs from a prototype in that a prototype code explores a small problem and is thrown away after its learnings, whereas tracer bullets represent the whole system and the code is use as a part of the final product.
   For example, to test a database migration code in my current workplace, we first built a migration script prototype. Then, we ran it on a special ‘tracer’ database which had low data volume but satisfying all conditions that we would want to test in the end system – e.g. orphan data, duplicate data, erroneous data. Running the migration script on this database told us exactly where we were going to get errors during the actual migration and we fixed our script accordingly.

5. Prototypes and Post it notes

   Building prototypes is a cheap and fast way to discover a solution to a problem, or to test whether a particular approach works. They are not just codes, but can also be design diagrams representing a system. Details such as proper code comments and formatting, correctness of every part of code, completeness etc. can be neglected for prototypes, since they are thrown away after relevant learning.

   For example, we have used prototypes and spike solutions in our Editor Scalability project to see whether the approaches we are considering are even possible. As a part of Milestone 1, we developed individual spike solutions for 1) automated opening of files will work 2) disabling syntax highlighting (which we later rejected as a non-relevant solution) 3) disabling semantic highlighting, 4) running parser as a separate thread, and 5) delayed parsing. These spikes gave us the confidence that we will be able to implement our ideas. Next step was to come up with exact implementation details like opening and editing a large file, and recording time taken to do so automatically.

6. Domain Languages

   We can define our own language at a high abstraction level to move the program closer to the problem. These languages can also be an extension to an existing language. Some languages work as data languages, to define a data structure for the program. Imperative languages can also include control statements that can be executed. These could also be used for defining meta-data for a program.

   Example An ER diagram is an abstract way to represent a relational database. It can be used to represent the relations in application logic without worrying about programming languages.

   After I started working with Ruby on Rails in my part time job as an application programmer at University Archives, I learnt that the models in RoR are also very close to the application logic, and are even coded in the form of “Class Book :has many bookReaders” (where bookReader is another class).

   Since the archives application had no technical documentation, I added basic relationships in the model and had to rely on my employer’s functional knowledge to complete the rest. The model representation being so natural in rails, my employer, with no programming experience, was easily able to code the remaining relationships into the model, while I worked on other tasks in parallel.

7. Estimating

   A pragmatic programmer develops the skill to intuitively estimate the time and feasibility of solutions. It is also important to consider the requirement for deciding the degree of accuracy required for the estimate. To come up with a good estimate, we need to understand the problem and model a solution, then break the model into components and estimate the difficulty of each component.

   Example We have used these techniques for giving time estimates to implement user stories in MP2 and Project Milestone 1 by dividing the stories into tasks, prioritizing, estimating difficulty and deciding on time estimates.

The Basic Tools

In this chapter, the authors have discussed the importance of using tools for software development, and provided tipis to best utilize them.

1. **Plain Text** has the advantage of being human readable and can be used by almost all compilers and SCM systems.
2. **Shell** programming can be used to invoke and combine usage of tools. Shell commands are very powerful and can be used to perform a gamut of ad hoc and automated tasks.
3. **Power editing** is possible when we know a single editor well and use it for all our editing needs. This way, we'll be able to use its shortcuts reflexively and write faster with fewer errors. We need to ensure that the editor chosen work on all platforms and is extensible and configurable.
4. **Source Code Control Systems** are very important software configuration tools. They help in version control, undoing any changes, allow users to work concurrently, automatic builds.
   Example: In my previous workplace, I have worked with CVS which although very simple to use, didn't allow branching easily.
   We have started using github extensively for our group projects in university, which is faster and merging and branching are easier.
5. **Code Generators** are used when a similar piece of code is required to be written numerous times.
   Example Code generators can be used to generate all possible input combinations for software testing purposes.
   Eclipse can automatically generate getter and setter methods for class variables.
   'Scaffold' command in Ruby on Rails automatically generates a basic UI layout with working functionality to create and delete entries, by just providing the model as input. This gives us a working prototype very early into the application development, and we can edit and change the generated code, and test after every change to make sure it didn't break. It is considered a good coding practice to gradually replace all the generated code by our own code and test at each step.

Pragmatic Paranoia

In this chapter, the authors discuss how to code in a way so as to guard the software against any errors caused by us or others, i.e., to check and validate at every step.
Some general tips to code well are as follows:

1. Software Contracts (Design by Contract)
   Like real-life contracts, interfaces must also have a pseudo-contract to fulfill their responsibilities. A software routine must be able to take care of the post-conditions and class invariance expected from it, provided that it receives the pre-conditions that it expects. A pragmatic programmer follows DBC (design by contract) and writes code that doesn't make too many promises in its contract.
   This basically means that we clearly specify the responsibilities of a module, so that it is expected to take care of only certain conditions, rather than expecting it to work under all conditions.

2. Crash early (Dead programs Tell No Lies)
   It is best to terminate the program as soon as an abnormal condition is detected, rather than have it continue and spread erroneous behavior to other modules.
   Example I recently did a programming assignment in my CS412 course where a number of methods had to modify a numerical class variable. I neglected adding a pre-condition check for the value of variable and optimistically assumed that the program will work correctly since the methods were individually working correctly. The value in the end was wrong and I learnt while debugging that somewhere in the program, my numerical variable was getting the value 'Not a Number'. I found myself circling between methods to identify the rogue method and ended up adding isNaN() check as precondition in every method dealing with that variable. If I had added validation checks and terminated the program as soon as a value deviated from expected, I would have saved the two hours that I later spent in debugging my program to identify the real cause of error.

3. Using Assertions (Assertive Programming)
   If we have a condition that we know cannot occur, we must use assertions to ensure that it doesn't, rather than coding with the blind belief that it won't happen. Some optimists believe that assertions are no longer required after code testing, but they forget to consider that testing cannot find all bugs in the program. A side-effect of assertions is that the programmer has to be extra-careful to ensure that the assertion statements themselves don't have any errors!

4. Using Exceptions (When to Use Exceptions)
   Exceptions must be used when something abnormal happens. It is not a good programming practice to use exceptions for purposes other than handling exceptional situations. Error handling may be used instead of, or with, exceptions.
   Example I was recently reading about Google's Go language which was initially released without exception handling, and found a stackoverflow.com discussion about it (http://stackoverflow.com/questions/1736146/why-is-exception-handling-bad).
   Exceptions are good for avoiding error state but they should not be used in the following situations (points below quoted from one of the posts in the thread):

1. Do not use exceptions to control program flow - i.e. do not rely on "catch" statements to change the flow of logic. Not only does this tend to hide various details around the logic, it can lead to poor performance.

2. Do not throw exceptions from within a function when a returned "status" would make more sense - only throw exceptions in an exceptional situation. Creating exceptions is an expensive, performance-intensive operation. For example, if you call a method to open a file and that file does not exist, throw a "FileNotFound" exception. If you call a method that determines whether a customer account exists, return a boolean value, do not return a "CustomerNotFound" exception.

3. When determining whether or not to handle an exception, do not use a "try...catch" clause unless you can do something useful with the exception. If you are not able to handle the exception, you should just let it bubble up the call stack. Otherwise, exceptions may get "swallowed" by the handler and the details will get lost (unless you rethrow the exception).

5. Resource Usage (How to Balance Resources)

Resources needed by a program could be files, threads, memory, transactions etc. It is important to de-allocate (free) any resource that we allocate and use. The routine using the resource must be responsible for freeing it after its use. Authors have given two tips for resource allocation when a routine needs multiple resources:

- Allocate resources in the same order in all routines when multiple routines use the same set of resources, to present a deadlock.
- Deallocation order must be opposite of allocation order to avoid orphan resources.

6. Objects and Exceptions

In object oriented languages, resources can be encapsulated in classes, as the allocation and deallocation processes correspond to constructor and destructor in a class. This gives us clean, consistent code.

In cases where the normal resource allocation pattern is not applicable, for example, when a structure allocates memory and passes it to its super structure, we have to follow a strategy such as making the aggregate stricture for deallocating memory allocated by its sub structures, or disallowing deallocation of top structure till the sub structure memory is deallocated. We can also write our own garbage collection mechanism in special cases.

Checking that the resources are freed after a structure exits is also a good practice.

Bend or Break

This chapter tells us how to write code that is easy to change later.

1. Coupling

Coupling and dependence between modules must be minimized as much as possible to write stable code. This is because in a tightly coupled code, it is hard to visualize what modules will get affected if we make change in a single module, which can create maintenance problems later.

Example We have studied about feature envy and inappropriate intimacy code smells that lead to dependence between modules. This can be minimized by using refactoring techniques such as move method, extract method, extract class.

Law of Demeter states that a method on an object should be allowed to invoke methods only belonging to the same object, or on a parameter passed to that method, or any objects that it created, or any directly held component objects.

Example Following this law will minimize the response for a class, and as taught in OO metrics, it has been shown that a large RFC indicates more faults.

2. Metaprogramming

The code should be freed of configuration details, like the choice of algorithms used, underlying database, middleware etc. Metadata should be used to define these configuration options. In general terms, metadata is data about data, but the authors feel that it should be taken as ‘data about the application’.

Metadata driven applications put code abstractions in configuration, providing the benefit of an adaptable, decoupled, customizable yet robust and stable code. It has additional benefits of being closer to problem domain. A single application engine can also be used for various projects with the use of metadata. Anything that is subject to change at a later time should be preferably coded as metadata. Another way

to add flexibility when using metadata is by writing programs that can scan this metadata whole they are running and change behavior at runtime, as opposed to a program which scans configuration file only at startup, and has to be restarted for every configuration change.
Example EJB is an example of metadata use for configuration.

3. Temporal Coupling
   Concurrency and Ordering are important time elements to be considered while writing software. Temporal coupling(having a rigid order) puts a restriction on order of execution and makes programs less flexible. A pragmatic programmer applies concepts of concurrency and parallel programming to decouple this ordering in time. UML activity diagrams can be used to model these concurrency and temporal dependencies.
   A 'hungry consumer model' can be used to get load balancing among multiple consumer processes. Here, any consumer task can take a temporally decoupled task from a central work queue and proceed concurrently with other consumers. This is called designing using 'services'.
   Some advantages of using concurrent programming are judicious use of global and static variables, and cleaner, maintainable interfaces. It also lets us deploy our application as standalone as well as client server, by decoupling operations.

4. Events and Subscription
   Event subscription model has a publisher that the subscribers can subscribe to. Whenever an event occurs in the publisher, it notifies all the subscribers subscribed to it. This way, each subscriber gets only the information that it is interested in.
   The publish/subscribe model can be used to implement model-view-controller architecture.
   This architecture has a model which represents database objects in the system. The view is the layer that the user interacts with, and provides input to. Controller layer is in between these two layers and implements relevant methods based on view inputs. Thus, model and view are decoupled.
   Example Observer design pattern gives a great way to design event-subscription code with loose coupling and scalability.

5. Blackboards
   Publishers and listeners are loosely coupled but must still have some information about each other, for example, they must agree on common conventions of communicating with each other. Blackboard technique can be used to further reduce coupling between publishers and listeners.
   A blackboard is a common area where publishers can post anything, and subscribers can access it, without them being aware of each others' presence.
   It is a place where data exchange can take place asynchronously and anonymously. JavaSpaces and T Spaces are examples of blackboard systems. Values can be retrieved from the blackboard in these systems by using templates to match objects on blackboard. Blackboards can also be used to store objects, on which the subscribers can perform operations. There is a single consistent interface to the blackboard. The

blackboard can also be partitioned and organized into zones or groups. Blackboards can be used to coordinate workflows consisting of independent entities.

While You Are Coding

In this chapter, the authors have advocated that coding is much more than mechanically converting design to code. They have provided tips and techniques to become a better coder. There are a number of decisions that need to be taken while coding. The authors have provided some of these problems, and the solution approach that should be adopted. There are some standard techniques that can be used to improve the speed of code, keep the code clean and maintainable when it is changing (refactoring), writing code that is easy to test, and the use of code generator tools.

1. Programming by Coincidence

   A good programmer is always wary of potential errors and tests his code as he develops it. His code doesn't work by coincidence, rather because he puts careful efforts into his code's correctness.

   Some cases where a program might work by coincidence are as follows:

   - Accidents of Implementation

     A program module might work because it relies on another module that is wrongly coded. This means that when the wrong module will be corrected, the dependent module will stop working. Calling routines in a wrong order or context can create a similar situation. It is only a coincidence that the dependent modules work in such cases.
   - Accidents of Context

     Sometimes the code that we write is dependent on the context (hardware/platform/ another module) without us being aware of this dependency. In such cases, the same code will stop working in a different context.
   - Implicit Assumptions

     A coder may make some assumptions while coding which do not always hold. In such cases, the code stops working as soon as the assumption ceases to be true. A pragmatic programmer doesn't assume anything that he can't prove.

     We can program 'deliberately' without relying on coincidences if we are on a constant alert and lookout for any potential errors. A coder should always try to develop a complete working of the application before attempting to develop it. We must always 'plan' the code and not rely on any implicit assumptions. Any assumptions made must be documented and made explicit for other programmers and users. It is also important to test the assumptions using assertion statements. We must refactor and improve existing code as much as possible.

2. Algorithm Speed

   Algorithms make take anything between constant to exponential time. A pragmatic programmer always estimates the resources that his algorithm will use. It is much easier to improve a complex algorithm while it is still being written, rather than doing it when the final application becomes unresponsive.

Big'O' notation can be used to do these time estimations. Apart from this, we can make an intuitive guess looking at the loop nesting, and algorithms like binary and divide and conquer algorithms. We must also test our estimates.
Example In a programming assignment, I wrote a piece of code using ArrayList data structure, and was simultaneously testing it on an input file with 20 records. The program while testing was giving the output instantly and improving the program further didn't even cross my mind. Real testing was to be done on a file with 6k lines and the program took a few seconds, but I did not think about improving it since it was a one-time assignment. This presumption came back to bite me when I realized that my next program had to run 10 iterations- each with a call to my current program. After writing the second program (which was taking around 5 minutes to return the output), I had to go back to my first program and change the data structure to Hash Map. I learnt the hard way that the earlier I make changes to improve performance, the lesser effort it will take later.

3. Refactoring
   We must refactor as soon as we detect a 'code smell' in an existing code. If we see anything abnormal happening, like code duplication, feature envy, too many switch statements, etc., it must immediately remind us about refactoring.
   Some IDEs provide automatic refactoring capabilities. (e.g. Eclipse Java IDE, Fortran plug-in).

4. Code That's Easy to Test
   It is important to test code as it is written. Individual modules must be unit tested before integrating them into a system.
   Example An even better strategy is the XP strategy of writing tests even before writing code. This ensures that the programs written are always correct, as long as the tests are correct.
   We can also look at unit testing as 'Testing by Contract', where we test every module to see that it fulfills its design contract. A design must also include the tests to verify its correctness.

Unit test code should be easily accessible, so that it can be used for regression testing by other developers.

- A **test harness** can be built for repeated testing. It can be used as a framework to perform common testing related operations. A test harness must provide a standard way to setup the environment for testing, a method of selecting tests to be run and output checking, as well as a failure reporting mechanism.
- **Test windows** can be used to monitor the code in production environment. This provides a view into the internal state of program. Some ways of doing this are through log files, and hot keys.

5. Evil Wizards

Wizards can generate hundreds of lines of code with just the click of a button. They generate the basic skeleton of the code to which we can add our methods and other functionality. We must never use wizard code unless we understand it. This is because the wizard-generated code will become direct part of the final application, and of any additional code that we write alongside.

Things to do Before a Project

1. Requirements pit
   Requirements have to be dug, rather than just collected from the surface. It must be understood 'why' user wants a particular thing, rather than just knowing 'what' he wants. Communication is very important and no assumptions must be made without consulting the user. It is important to step into the user's shoes. Use cases and use case diagrams can be used to document requirements. Tracking of requirements is essential for keeping a track of project schedules, because addition of requirements often causes project delays and it is important to maintain a record of expanding requirements.
   Example: A project where I worked in Oracle Corp was completed 6 months behind schedule because what was intended to be an upgrade from version 10.1.23 to 13.1.2 of ORMS (Oracle Retail Merchandising System) later became an upgrade project from v10.1.25 to v13.2.3 due to new branches being released by the company after we had started on migration. Because of this, we had to revisit our designs, make required changes to code and repeat unit and functional testing all over again. It would have been on schedule if we had had the foresight to consider that a customer, if migrating, would normally want to migrate to the latest version, and if 13.2.3 is slated for release before our project completion, we should from the very beginning work on creating a migration path to that version.

2. Solving Impossible Puzzles
   It is important to identify all constraints and consider all solutions before dismissing any solution as irrelevant.

3. Not Until you are Ready
   It is important to listen to our instinct when we start programming. With experience, we develop a sense of what will work and what can go wrong, and any doubts or suspicion must be heeded to before starting working. At the same time, it is important to start working on a prototype rather than procrastinating and waiting for all parts of the project to fall into place.

4. Automation
   Doing things manually can introduce bugs because everybody may do a certain thing in a slightly different way, and possibly interpret written instructions also differently. The way to make things foolproof is by automating processes.

5. Ruthless Testing
   Major types of software tests that need to be performed are unit testing, integration testing, usability, performance and resource exhaustion testing.
   Example Tests that are run many times (regression tests) can be automated to make sure that existing code hasn't broken after modifications.

6. It's All Writing
   Pragmatic principles must be followed not only for writing code, but also for writing documentation. It is easier to keep web documentation up to date.

Example: When I started working after my undergraduate degree, I was of the opinion that documentation simply means writing user manuals and this only requires good language skills. I was oblivious of how much domain knowledge goes into doing that task. When our upgrade project in Oracle was in its final stages, my manager requested me to work with the documentation team and explain to them the product working. I was surprised to see that the documentation team had in-depth functional knowledge and was even running the code to understand the internals of the product.

7. Great Expectations
   An application is successful not just if it meets the requirements, but when it meets the customers' expectations. A product must always be built upon users vision, and if anything 'exceeding' their expectations is to be incorporated, the users should be gently explained why it is important. Communication is the key here.