Jeevan Koshy

Abstract

1. **Software Application Domain and Legacy Software**
2. **Software Process Engineering Practices and Myths**

SOftware engineering

1740256

**CONTENTS**

**SL NO. TITLE PAGE NO.**

|  |  |  |
| --- | --- | --- |
| 1 | Software Application Domain | 2 - 4 |
| 2 | Legacy software | 5 – 6 |
| 3 | Software Engineering Practices | 6 |
| 4 | Software Myths | 7 |
| 5 | References | 8 |

**SOFTWARE APPLICATION DOMAIN**

We can broadly categorise the software application domain into different types –

1. System Software
2. Application Software
3. Engineering Software
4. Embedded Software
5. Product line software
6. Web applications
7. Artificial Intelligence Software
8. Open world Computing
9. Net sourcing
10. Open source

---------------------------------------------------------------------------------------------------------

**System Software –**

The system software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,4 information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

**Application Software –**

The application software is a stand-alone program that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time (e.g., point-of sale transaction processing, real-time manufacturing process control).

**Engineering Software –**

The engineering software has been characterized by “number crunching” algorithms. Applications range from astronomy to volcanology,

from automotive stress analysis to space shuttle orbital dynamics, and

from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation,

and other interactive applications have begun to take on real-time and

even system software characteristics.

**Embedded software –**The embedded software resides within a product or system and is used to

implement and control features and functions for the end user and for the

system itself. Embedded software can perform limited and esoteric functions

(e.g., key pad control for a microwave oven) or provide significant function

and control capability (e.g., digital functions in an automobile such as fuel

control, dashboard displays, and braking systems).

**Product line software –**

The product line software is designed to provide a specific capability for use by

many different customers. Product-line software can focus on a limited and

esoteric marketplace (e.g., inventory control products) or address mass

consumer markets (e.g., word processing, spreadsheets, computer graphics,

multimedia, entertainment, database management, and personal and

business financial applications).

**Web applications –**

Web applications also known as “WebApps”, a network-centric software category spans a wide array of applications. In their simplest form, WebApps can

be little more than a set of linked hypertext files that present information

using text and limited graphics. However, as Web 2.0 emerges, WebApps are

evolving into sophisticated computing environments that not only provide

stand-alone features, computing functions, and content to the end user, but are integrated with corporate databases and business applications.

**Artificial Intelligence software –**

The artificial intelligence software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing. Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work a program that is older than she is!

**Open world Computing –**

The rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

**Net sourcing –**

The World Wide Web is rapidly becoming a computing engine

as well as a content provider. The challenge for software engineers is to

architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

**Open source –**

The open source is a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development. The challenge for software engineers is to build source code that is self-descriptive,

but more importantly, to develop techniques that will enable both customers

and developers to know what changes have been made and how those

changes manifest themselves within the software.

---------------------------------------------------------------------------------------------------------

**LEGACY SOFTWARE**

Legacy software is basically older software that is useful. Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases much older. These older programs—often referred to as legacy software—have been the focus of continuous attention and concern since the 1960s. Legacy software is characterized by longevity and business criticality. Unfortunately, there is sometimes one additional characteristic that is present in legacy software poor quality. Legacy systems sometimes have inextensible designs, convoluted code, poor or non-existent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long. And yet, these systems support “core business functions and are indispensable to the business.” What to do? The only reasonable answer may be: Do nothing, at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn’t broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

* The software must be adapted to meet the needs of new computing environments or technology.
* The software must be enhanced to implement new business requirements.
* The software must be extended to make it interoperable with other more modern systems or databases.
* The software must be re-architected to make it viable within a network

environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable into the future. The goal of modern software engineering is to “devise methodologies that are founded on the notion of evolution”; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other”.

The problems with legacy software are –

* Original developers not available.
* There are outdated development methods used.
* Extensive patches and modifications have been made.
* Missing or outdated documentation.

---------------------------------------------------------------------------------------------------------

**SOFTWARE ENGINEERING PRACTICES**

There are many approaches that we can implement towards software engineering. The most important points to note while we execute the software engineering practices are –

1. Analyse
2. Design
3. Coding
4. Testing
5. Verification
6. Modification
7. Implementation

The 7 points above can be categorised into 4 simpler options.

1. **Understand the Problem** – It is important that we understand the problem to apply the practice of software engineering. If we do so, analysing the problem to get a solution will be really helpful.
2. **Plan the solution** – After we understand the problem, we will have to work towards getting a solution and therefore it is important that we plan a solution. Only if we plan a solution can we achieve a solution.
3. **Carryout the Plan** – Now since we have a solution in mind, the next step is to execute the plan.
4. **Examine the result for accuracy** – The last step after carrying out the plan is to examine and carefully look at the solution and conclusion we have come to. We have to check whether it’s the same or similar to the solution we have planned.

---------------------------------------------------------------------------------------------------------

**SOFTWARE MYTHS**

Software myths are erroneous beliefs about software and the process that is used to build it which can be traced to the earliest days of computing. They are a number of common belief’s that software managers, customers and developers believe falsely. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who “know the score.” Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

There are different types of Myths –

**Management** = Managers with software responsibility, like managers in

most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. So, in all that the manager does, there might be a false belief which might pop up.

**Customer** = A customer who requests computer software may be a person

at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

**Practitioner** = Myths that are still believed by software practitioners have

been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

***Example -***

**Myth**: If I decide to outsource the software project to a third party, I can just

relax and let that firm build it.

**Reality**: If an organization does not understand how to manage and control

software projects internally, it will invariably struggle when it outsources

software projects.

**REFERENCES**

1. Software Engineering – A practitioner’s approach. [7th Edition]
2. Presentation used in class.
3. Notes Taken down during lecture.

---------------------------------------------------------------------------------------------------------