Software Development Methodology by Java Programming Language

A. Vishal Kaushik¹, B. Rohit Saini²
¹Vishal Kaushik, Dronacharya College of Engg., Gurgaon
²Rohit Saini, Dronacharya College of Engg., Gurgaon

Abstract: This paper provides a basic understanding of software development by java programming language defined with its different modules.

I. INTRODUCTION

Computers are used for solving problems quickly and accurately irrespective of the magnitude of the input. To solve a problem, a sequence of instructions is communicated to the computer. To communicate these instructions, programming languages are developed. The instructions written in a programming language comprise a program. A group of programs developed for certain specific purposes are referred to as software whereas the electronic components of a computer are referred to as hardware. It is not manufactured in the usual sense of that word. Furthermore, it recognizes an even stronger need in software development to address quality problems upstream, because that is where almost all software defects are introduction.

II. RUP AND XP

During the past couple of years, the two most popular methodologies to emerge are the Rational Unified Process (RUP) and Extreme Programming (XP). RUP, according to many developers, is too cumbersome, and XP isn't suitable for serious projects. Some people would go so far as to say that XP is for hackers and RUP is for humongous projects and not realistic for average-sized project teams (i.e., four to six developers). There are also companies that employ a blend of the two, with RUP cycles and phases, and XP pair programming and testing.

III. FAILED PROJECTS

In addition to RUP and XP, several other methodologies are used in the industry, such as ADM Scrum and Agile Modeling. Of course, many project-based IT services companies have home grown methodologies of their own. Either way, IT services companies (e.g., solutions integrators, consultants) have done well financially helping clients adopt some form of methodology. The problem with methodologies, processes, and CASE tools to date is that they are too rigid and bureaucratic, and simply do not factor in key aspects of the development process, such as personnel issues. It never ceases to amaze me how many projects fail even after large amounts of money, time, and resources are pumped in to make them successful. Often, after a project has completed, companies realize that they ignored many common sense issues, which led to the project's failure.

IV. HIGH INTEGRITY SOFTWARE

We refer to software that has a higher than normal expectation of correctness as high integrity software. This expectation of correctness is closely linked to the risks inherent in software failure. As risks increase so too does the need to ensure that there are as few software errors as possible. However, the resources (cost, time and so on) required to help ensure correctness also rise. Therefore, the development of high integrity software demands greater resources than the development of a ‘regular’ software product. A concept closely related to that of high integrity software is that of critical software. The term critical software applies to software that poses dangers should it fail. Critical software can further be categorized depending upon the types of danger imposed by failure. For example, failure of business critical software could adversely affect the economic success of an enterprise; examples include the software used to control a bank’s ATM transactions and software aimed at providing security for sensitive information. Failure in mission critical software, on the other hand, could impair the goal of the given mission. Examples here include such applications as satellite and rocket launch systems. Finally, failure of safety
critical software could result in harm to people, property or the environment. Examples include medical control software and air traffic control software. There can be degrees of danger posed by software failure, so that some software is of higher integrity than other software; that is, a higher degree of confidence is required in its correctness than is the case for other software. For example, consider the software used to monitor air traffic flow around an airport and software used to monitor the temperature in a fridge freezer. Although both are examples of critical software, failure in the former could have far more catastrophic consequences than failure in the latter. Amongst other things, software failure in a fridge freezer is likely to be protected against by some form of hardware lock, whereas hardware locks cannot protect against errors in air traffic software. We refer to these degrees of integrity as integrity levels.

V. CLASSIFYING FORMAL METHODS

Many formal methods have been established over the years. A common way of classifying these formal methods is by the approach taken in the method of specification. The two principal approaches are algebraic and model-based approaches. Using an algebraic approach, once a list of operations has been identified, their behaviour is captured indirectly by describing the relationship between these operations as a set of properties (or axioms as they are sometimes known).

VI. FORMAL METHODS

Formal methods constitute a branch of software engineering that incorporates the use of mathematics for software development. A formal method provides a formal language in which to express the initial specification and all future design steps towards the final program. These design steps are often referred to as transformations (see Figure 1.3). A formal method is more than just a specification language for recording these transformations. It also includes a proof system for demonstrating that each transformation preserves the formal meaning captured in the previous step. A proof system is a means of guaranteeing the correctness of a statement and relies upon.

VII. LIGHTWEIGHT FORMAL METHODS

Informal methods of software development (such as the Structured System Analysis and Design Methodology) often prescribe strict rules for progressing from one stage of software development to the next. The majority of formal methods, on the other hand, provide a selection of tools for the development of reliable software systems rather than prescribe their use at every stage of development. Thus, for software of high integrity, all the tools provided by a formal method (such as a modelling language for specification and a formal proof system for software design and implementation) could be utilized. Proofs themselves may be carried out (discharged) totally formally (that is, where every step is justified using the method’s proof system) or proofs may just be rigorous (in which case they can be discharged by means of a sound argument rather than a complete proof). Again, the integrity level of the software will inform this decision. 8 Formal Software Development Table 1.2 Classifying some leading formal methods Algebraic Model-based Sequential Larch Vienna Development Method (VDM) systems Z B Concurrent Calculus of Communicating Systems (CCS) Prototype Verification System (PVS) systems OBJ Communicating Sequential Processes (CSP) Where software is of lower integrity the modelling tools available in the language might be adopted for software specification, but development may then proceed using more traditional approaches with integrity checks being argued informally or by means of run-time assertions (checks) embedded into code. Using a formal method in this way, with less reliance upon the discharge of proof obligations, is often referred to as a lightweight approach to the use of formal methods. It is a lightweight approach that we shall adopt in this text (Figure 1.4). As you will come to see, a VDM specification corresponds closely to the notion of a class in an object-oriented methodology. The approach we will take in this text is to record the informal specification of software using the UML class notation. We will then provide a formal specification for a UML class in the form of a VDM specification. Following each chapter that deals with an aspect of the modelling (specification) language of VDM (known as VMD-SL), we demonstrate the development of Java programs from the VDM specifications. The correctness of any design decisions we make will be argued rigorously rather than formally, and backed up by assertions embedded in the final Java code.
VIII. LOGICAL CONNECTIVES

Simple propositions can be combined into compound statements by operators called logical connectives. The purpose of defining these connectives is to provide a rigorous framework that gives precise meaning to such words as ‘and’ and ‘or’ that occur in the natural language. The way we give semantic meaning to these connectives is to provide tables known as truth tables, which give a value for every possible combination of the values of the individual statements that make up the compound proposition. This is made clear below as we explain the meaning of the various connectives.