Switch: An Extensible Framework for the Generation of Bespoke and Dynamic Software Development Models

By

Tichaona Dhliwayo

R014248A

University *of* Zimbabwe

Faculty of Science

Department of Computer Science

A thesis submitted in partial fulfilment of the requirements for the degree of

Masters of Science in Computer Science

February 2006

Abstract

Attempts to come up with a single **Software Process Model** (SPM) that can be used in all development scenarios have failed. The same can be said for the approach that seeks to come up with an authoritative and exhaustive set of software process model. Currently the software engineering fraternity finds itself with numerous software process models none of which is perfect for all projects. Frameworks have been proposed that attempt to encompass the best features of all software process models, but still results have not been satisfactory. Most problems with current SPMs can be attributed to the underlying assumption that “in relation to SPMs all projects are the same”.

The premise of this thesis is that in relation to SPMs all software projects are unique and as such each requires its own software process model. A framework (Switch) for designing these project specific models is designed and specified after numerous investigations in software processes and Model Determinants (attributes of a software project which lands it to the construction of a unique SPM).

The Switch framework was tested to see if it produces valid SPMs as per framework specification. For numerous hypothetical and actual software development project scenarios, models produced by the framework proved to be logically superior to generic Software Process Models.

Acknowledgements

I would like to thank my supervisor Mr Kachepa for being the only person willing to supervise a Software Engineering project we need more like you.

Miss A.K Chitsika for all the help and support in validating and verifying theories, for the tireless efforts in proofreading the final document, I thank God for making you a part of my life.

Many thanks for the stunning graphics in the implementation go to Mr G T Kachambwa, the future looks bright but we have to catch the tide while it is high.

To Mr R Chinyakata and M Sibanda I can't thank you guys enough for those words of encouragement when the going got tough; Masters would never be the same without you.

Last but not least my sister Mrs C G Gwata if it was not for you we would never have gotten this far, wherever mom is you have made her proud.

Table of Contents

LIST OF TABLES VII

LIST OF FIGURES VIII

LIST OF EQUATIONS IX

1.0 INTRODUCTION 1
1.1 Background to the Research 2
1.2 Statement of the Problem 5
1.3 Significance of the Study 6
1.4 Research Objectives 9
1.5 Scope of the Study 11
1.6 Limitations 12
1.7 Thesis Organization 12

2.0 LITERATURE REVIEW 14
2.1 Challenges in Software Development 14
2.2 Inconsistent and Conflicting Terms 15
2.3 Functional and Non-Functional Requirements 18
2.4 Generic Software Process Models 20
2.5 Waterfall Model 21
2.6 Prototyping Model 26
2.7 Spiral Model 30
2.8 Rapid Application Development 33
2.9 Agile Methods 35
2.10 Adaptive Software Development 37
2.11 Extreme Programming 39
2.12 Crystal 41
2.13 Rational Unified Process 44
2.14 Software Process Model Selection 46
2.15 Software Process Model Customisation 54
2.16 Software Capability Maturity Model 54

3.0 RESEARCH METHODOLOGY 58
3.1 Switch Framework Design Method 58
3.2 Core Framework Extension Design Method 59

4.0 RESULTS OF FINDINGS 60
4.1 Software Development Projects 60
4.2 Model Extensions 61
4.3 Framework Weightings 61
4.4 Model Determinants 63
4.5 Software Development Processes 64
4.6 Determining Relationships 65
4.7 Negative Software Development Process Relationships 65
4.8 Resolving Actions 66

4.9 Switch Software Development Model 67
4.10 Core Software Development Processes 68
4.11 Core Software Development Model Extension 72
4.12 Model Construction 74
4.13 Switch Framework Acid Test 78
4.14 Test Runs of the Switch Framework 80

5.0 DISCUSSION 87
5.1 Analysis of SDM produced by Switch 87
5.2 Limitations of the Switch Framework 87

6.0 CONCLUSIONS AND RECOMMENDATIONS 89
6.1 Summary 89
6.2 Conclusions 89
6.3 Future work 90

REFERENCES 91

List of Tables

Table 1: Selecting a life cycle Model based on characteristics of requirements 48
Table 2: Selecting life cycle Model based on characteristics of the Project team 49
Table 3: Selecting a life cycle model on characteristics of the User Community 50
Table 4: Selecting a life cycle model based on characteristics of project type & risk 52
Table 5: Process Maturity Framework 55
Table 6: Switch V1.0 Framework Weightings 62
Table 7: Value Rages for Weighted Objects 62
Table 8: Set of Resolving Actions 67
Table 9: Core Switch Software Development Processes 70
Table 10: Core Software Development Model Determinants 72
Table 11: Core Software Development Processes 73
Table 12: Core Determining Relationships 73

List of Figures

Figure 1: Software Development Scenario A 2
Figure 2: Software Development Scenario B 3
Figure 3: Software Development Scenario C 3
Figure 4: Software Development Scenario D 4
Figure 5: A Software Development Scenario Using the Switch Framework 5
Figure 6: Waterfall Software Process Model (Source Software Engineering) 21
Figure 7: Evolutionary prototyping (Source Software Engineering) 26
Figure 8: Spiral model (Source Software Engineering) 30
Figure 9: Adaptive and Evolutionary Life Cycles (Source Everette R. Keith 2002) 38
Figure 10 Extreme Programming (Source Extreme Programming, 2002) 40
Figure 11: One Crystal Orange Increment (Source Agile Software Development) 43
Figure 12: RUP workflows and phases (Source: The Unified Process. 1999) 45
Figure 13: Switch's Universal Graphical SDM Notation 68
Figure 14: Core Switch Software Development Model 71

List of Equations

Equation 1: User Weighting Equation 75
Equation 2: Process Weighting Equation 75
Equation 3: PW Weighting Equation Simplified 76
Equation 4: Overall PW Equation 76

1.0 Introduction

The software crisis of the late sixties (Dijkstra E, 1972) led to the formalisation of software development and it's emergence as an engineering discipline. Escalating costs of software development largely due to a black box approach to software development made it necessary to formally define the phases a software product goes through from the time it is conceived to the time it is no longer in use. This framework for understanding and developing information systems and software successfully came to be known as the software development life cycle SDLC (Wikipedia.org, 2006). If the phases in software development could be explicitly defined then it would be possible to systematically manage the development process and consequently manage development costs. Various software process models were developed to describe the software Life-Cycle (Waterfall Model, Spiral, Formal Transformation, Prototyping, Re-use, Clean room, Time box Development, Agile Models …) each of these models were crafted to be the ultimate model that would bring an end to the woes of software development. For the greater part the software crisis is over but with its passing away came the extinction of some of the software process models that were designed to end it and those that have survived have undergone major revisions. Evidently these models have done much in trying to turn software development into an engineering discipline; however any given model leaves much to be desired in as far as providing the ultimate model for all development scenarios and the approach of choosing a model to use on a project basis has not proved to be the definitive solution either.

1.1 Background to the Research

This section provides some background information to the research area which will enable the reader to better understand and appreciate the thesis and objectives of the research.

Software Development Scenarios

Three software development scenarios are depicted below with the intension of providing an explanation how software is developed in actual software authoring houses. In section 1.2 the Switch Framework is introduced providing the fifth software development scenario.

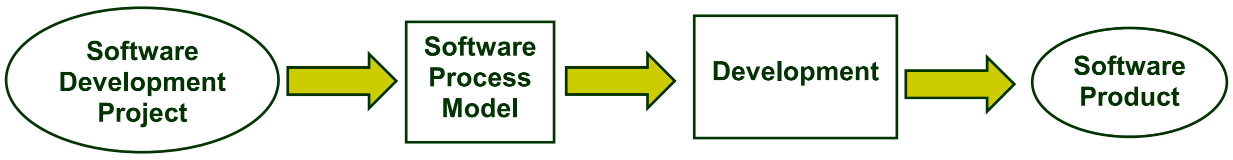


Figure 1: Software Development Scenario A

Scenario A above was the first and this typically describes a software development pattern that was followed in the early days of the waterfall model. An organisation standardises on one Software Process Model (SPM) such as the Waterfall model primarily because there were no other models available. Every project that the organisation undertakes is developed using the same process model.

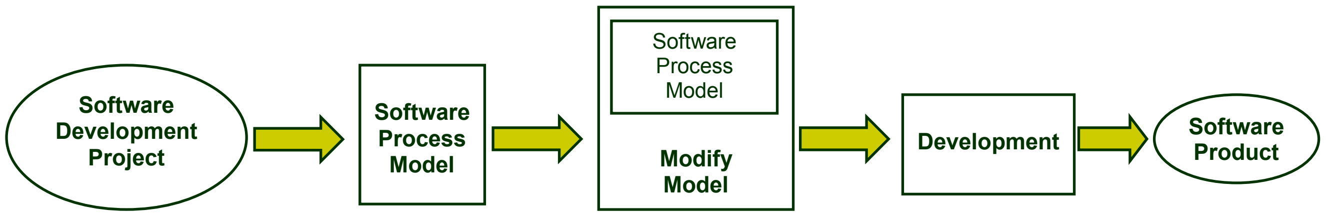


Figure 2: Software Development Scenario B

With time the single software process model for all projects proved to be inadequate as project diversity and complexity increased. In this scenario an organisation still standardises on one software process model but for each project the model is modified to suit particular project needs. This proves to be a better approach but a given model can only be modified up to a certain extent thereafter it becomes distorted consequently yields unpredictable results.

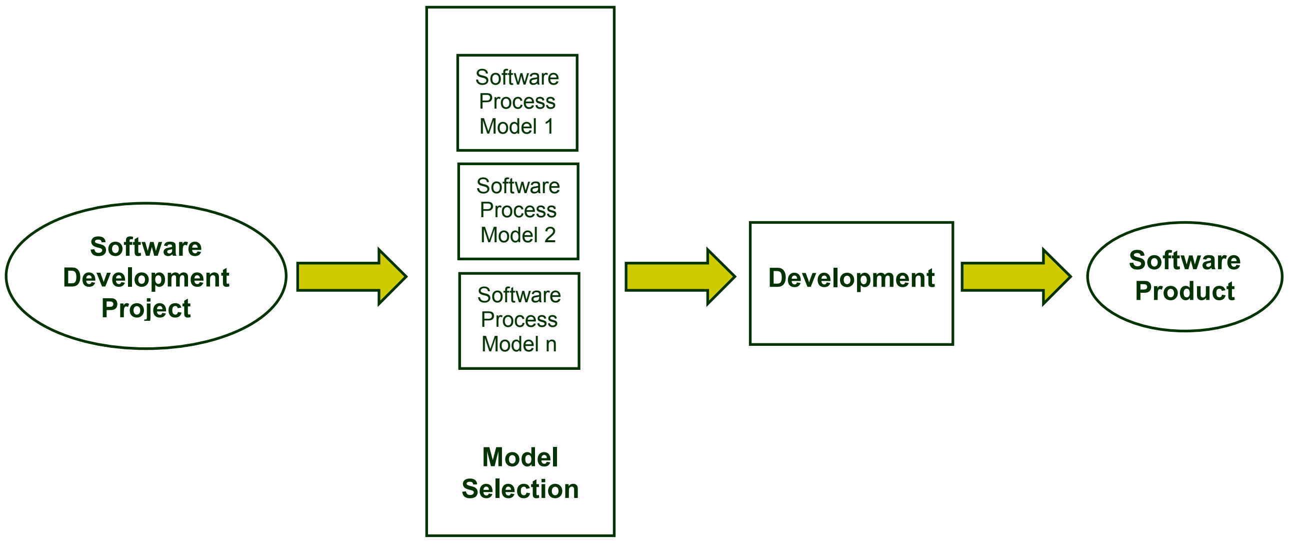


Figure 3: Software Development Scenario C

The two scenarios described above eventually gave way to the development of many Software Process Models (SPMs), each of these models were designed to be the ultimate model but without much success. Because none of them worked in all situations organisations would typically have an array of SPMs, given a particular project they would choose the SPM that would most likely produce the best results for the project at hand. Undoubtedly the process of selecting a SPM to use introduced additional complexity to the whole software development process.

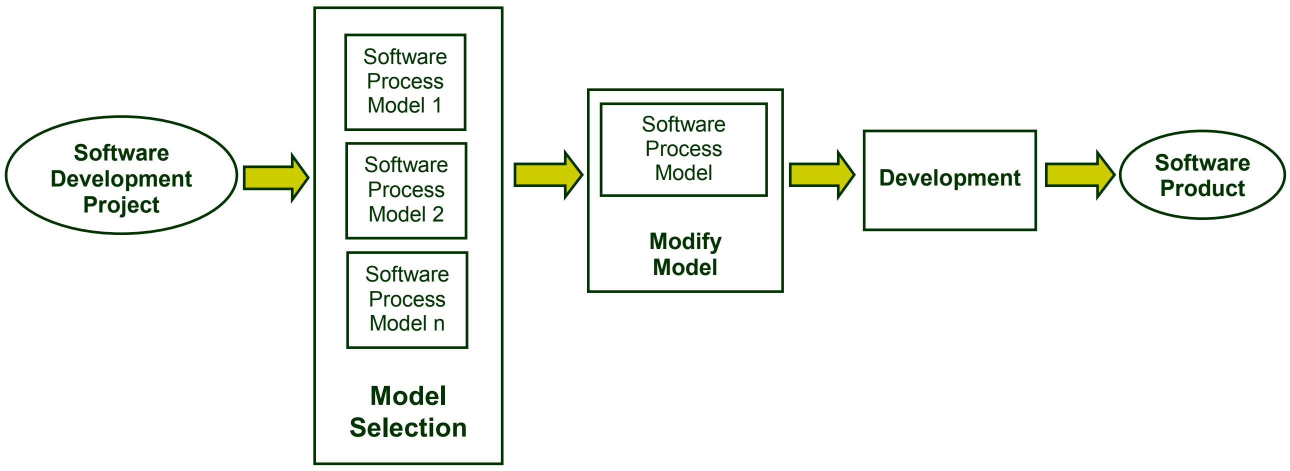


Figure 4: Software Development Scenario D

Scenario D best describes a modern software development scenario were a plethora of software process models exist, but hardly any can be used out of the box. Typically when a software development firm gets a project they select a software process model that seems most likely to produce the best results. This model is further customised to suit the project at hand. Scenario D is flawed in two aspects both the model selection and model customisation process are not formalised and well defined which makes it possible to select and customise the wrong model.

1.2 Statement of the Problem

The design and specification of an Extensible Framework for the Generation of Bespoke and Dynamic Software Development Models, this framework will be referred to as Switch.

Elaboration of statement of the problem

Given a software development project it should be possible to come up with a Software Process Model (The Switch framework will refer to these as Software Development Models) which is specific to the given project and is the best possible model for implementing the project. Switch is the framework that will allow for the generation of these project specific Software Development Models. The fourth software development scenario in which this Switch is to be used is depicted and explained below.

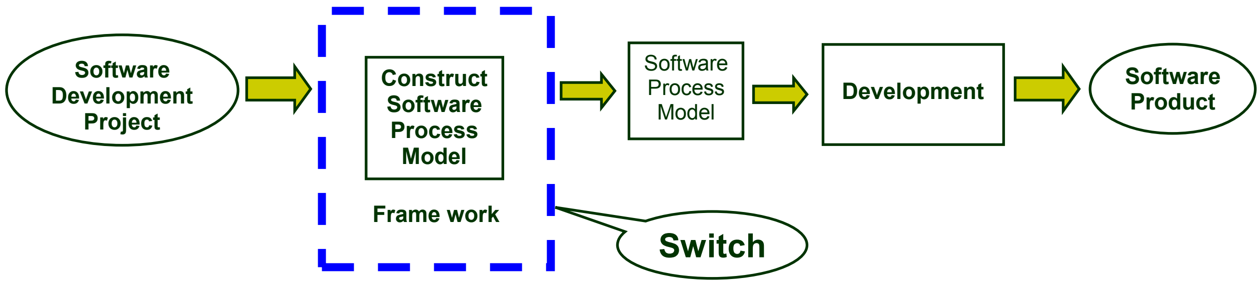


Figure 5: A Software Development Scenario Using the Switch Framework

Essentially this scenario introduces an automated pre-software development stage for constructing a software process model. Given a software project attributes of the project are ascertained, for the most part these attributes are non functional requirements which we will refer to as Model Determinants (MDs). These Model Determinants are then fed into the Switch framework which in turn constructs a software development model for that particular project based on the supplied MDs. The design and specification of this framework is the subject of this research.

1.3 Significance of the Study

This section aims to justify the importance and necessity of this study it starts of by looking at the current software process models and the problems associated with them and then goes on to explain how the Switch framework will addressee some of these problems and consequently improve Software Engineering as a discipline. Some of the areas covered by this section will further be elaborated on in the literature review as the primary concern of this section is justifying the research work.

Agile methods

These place less emphasis on analysis and design but on early implementations, working software is considered more important than documentation. They are responsive to change by closely collaborating with the client, but most agile methods do not work well with large projects due to poorly specified analysis and design.

Prototyping

The approach is to construct a quick and dirty partial implementation of the system during or before the requirements phase for the purpose of better understanding the requirements (throw away prototyping). When used this way prototyping has high overheads as considerable resources may be dedicated for the construction of a prototype that will eventually be thrown away.

With evolutionary prototyping the prototype evolves to become the solution resulting in an unstructured solution which is very difficult to maintain and has large-system integration problems.

Spiral

The spiral uses a risk management approach to software development, with its major handicap being with small projects it becomes unnecessarily cumbersome and there are no fixed phases such as specification or design, loops in the spiral are chosen depending on what is required arguably some very important steps might be left out.

Waterfall

The waterfall provides an orderly sequence of development steps and helps ensure the adequacy of documentation and design reviews to ensure quality, reliability, and maintainability of the developed software. Its limitations are that it is rigid, needlessly slow and cumbersome but more interestingly the final software produced may not meet client's needs.

Problems associated with selecting a SPM

The biggest problem associated with selecting a Software Process Model (SPM) is that often the project requires characteristics from multiple SPMs which make it difficult to choose one SPM. In practice selecting a particular SPM results in some required characteristics of the other models being neglected, which in most cases translates to some of the projects non functional requirements being violated or ignored.

Problems associated with modifying a SPM

By definition, a project is a unique undertaking, resulting in a unique product. If follows then that a project is likely to employ a unique process for product development (Futrell T R, F & I Shafer 2002). Having selected a SPM there is still need to modify it to suit a given project. There are no defined rules on how to modify a given SPM to suit a project hence this is often done in an ad hoc way results in a defective SPM. Furthermore a greater number of current SPMs are not designed to be modified but to be static and absolute.

Switch

If choosing a SPM is a complicated process that may sacrifice some of the non functional requirements and modifying a SPM may result in a non-functional model; given that each Software project is unique then the next port of call is to construct a SPM for each project. If a model is constructed for a specific project then no other generic model will do a better job of implementing the project, this is a given fact. The question is can such a model be constructed, how is it constructed and what parameters are needed to construct the model? This research is aimed at constructing a frame work for generating project specific SPM (Switch) and consequently answer the question stated above. The benefits that will be derived from the Switch framework are as stated below.

- **Improved software**, it is anticipated that if software is developed using a development model that was designed specifically for that software project the end result is better quality software.
- **More efficient software development process**, currently some simple software products are developed using SPMs that include unnecessary steps for simple projects conversely some complicated projects are implemented using SPMs that do not include essential steps for large projects. With the use of Switch model determinants will be used to determine which steps are to be included in the SPM consequently improving the development process.
- **Shorter development time,** the SPM used to develop a software product directly affects the time taken to develop the software. More processes in the model translate to longer development time at the same time omitting essential processes will eventually introduce errors in the software which in turn will increase development time.
- **Lower software development cost**, ultimately reducing development time reduces development cost.

1.4 Research Objectives

In order to come up with the Switch framework certain key objectives have to be met these are outlined below.

Objectives

I. Carry out a study of software projects with the aim of coming up with attributes that make a given software project suitable for development using a particular SPM. These attributes will be referred to as Model Determinants.

II. Design and specification of an “*Extensible* Framework for the Generation of *Bespoke* and *Dynamic* Software Development Models” this frame work is what we will refer to as Switch and it should have the following attributes.

   - **Extensible** :- Given an application domain it should be possible to extend the set of Model Determinants and Software Development Processes effectively expanding the set of Software Development Models that can be constructed by Switch.

   - **Bespoke**: - The Software Development Models generated by Switch will be tailor made for a given software project.

   - **Dynamic**: - For the same software development project changing the Model Determinants (Effectively changing the non functional requirements’) or just the emphasis on the Model Determinants will result in a different Software Model being constructed by the Switch framework.

III. The design and specification of a Core Framework Extension for Switch

IV. Comparing and analysing the Software Development Models produced by Switch against generic Software Process Models on a project by project basis.

1.5 Scope of the Study

Objective i

- Definition of Software Project
- Definition of Software Process Model
- Non Functional Requirements
- Formal Definition of Model Determinants (MDs), their acquisition and quantification.

Objective ii

- Model Extensions
- Framework Weightings
- Model Determinants
- Software Development Processes
- Determining Relationships
- Negative Software Development Process Relationships
- Resolving Actions
- Switch Software Development Models
- Core Software Development Processes
- Model Construction

Objective iii

- Model Determinants
- Software Development Processes
- Determining Relationships
- Negative Relationships

Objective iv

- Acid test for the Switch framework, having specified the Switch framework on paper it should be possible to use Switch to construct a SDM for its own implementation.
- Come up with software project scenarios and use Switch to produce the SDM then compare these models to generic SPMs.

1.6 Limitations

The Switch framework is only as good as its extension the more application domain specific extensions the more useful the framework. Apart from the core extension (which is part of the framework specification) on other extensions are provided. Even with just the core extension Switch proves to be a far more superior solution, however its potential power is limited due to the absence of additional extensions like database application extensions, multimedia, games etc.

1.7 Thesis Organization

This thesis is organised to conform to the University of Zimbabwe standards for Masters of Science Degrees. This chapter basically serves as an introductory chapter which sets the scene for the literature review (Chapter 2) and all other subsequent chapters.

The research methodology is covered in chapter 3. Chapter 4 presents the results and findings of the research which mainly consist of the switch framework specification and test runs. Chapter 5 discusses and analyses the results mainly focusing on the framework and the SDM produced by the framework. Lastly chapter 6 concludes the research.

2.0 Literature Review

This chapter takes an in depth review of related work covered by other authors. It follows the development of the Software Life Cycle and Software Process Models in their chronological order. It is hoped that this approach will make the whole review more comprehensive and complete. It should be noted that the aim is not to define all Software Process Models (SPMs) but to place emphasis on when to use a particular SPM and what its weakness are.

2.1 Challenges in Software Development

In this section challenges in software development are reviewed to clarify and substantiate the necessity of a research into software process models which are universally accepted as part of the solution to the challenges that are faced by the software engineers the world over.

The Standish (Standish Report 2003) group reports astonishing figures (reviewing 13,522 projects)

- 66% of all software projects fail
- 82% of projects experience time overruns
- 48% of projects do not have the required features on product release
- $55 Billion in US project waste (schedule, budget, scope)

In light of the above figures it is imperative that some effort must be applied to the science of Software Process Model (SPM) formulation as better SPM can result in higher success rate for software projects.

2.2 Inconsistent and Conflicting Terms

Software Process Model and Software Life Cycle

Sommerville I (2001) in his book Software Engineering defines a software process as the set of activities whose goal is the development or evolution of software. He then goes on to define ***Software Process Models*** *(SPMs)* as a simplified representation of a software process, presented from a specific perspective. Examples of SPM are cited as
Waterfall Model, Spiral, Formal Transformation and Prototyping.

Walt Scacchi in his paper “Process Models in Software Engineering” (2001) says:
In contrast to software life cycle models, ***Software Process Models*** often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing. Scacchi clearly points out that Software process models are different from software life cycles.

In a paper titled Process-Centred Software Engineering Environments A Brief History and Future Challenges by Volker Gruhn (2002) A software process is defined as a set of all activities which are carried out in the context of a concrete software development project which usually covers aspects of software development, quality management, configuration management and project management. Gruhn goes on to say the description of a software process is called a **software process model**. Further more a software process model does not only describe the activities which have to be executed, but also the tools to be used, the types of objects and documents to be created/manipulated and the roles of persons involved in software development. Gruhn fails to point out examples of his software process model and his definition is some what restrictive as his definition makes it necessary to state the tools to be used directly implying that spiral waterfall and prototyping are not software process models and hence contradicting with Sommerville and Scacchi.

The ***Software Development Life Cycle*** (SDLC) is a framework for understanding and developing information systems and software successfully (Wikipedia, 2005). Some popular models of a SDLC are cited as the waterfall model, the spiral model, and the incremental build model. This definition coincides with Sommerville definition of Software process model. This further demonstrates the inconsistencies in the use of the terms Software Development Life Cycle and Software Process Model.

IEEE (1990) defines the ***Software Life Cycle*** as the period of time that begins when a software product is conceived and ends when the software is no longer available for use. This life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. These phases may overlap or be performed iteratively, depending on the software development approach used. In the paper "Breathing new life into the waterfall model" published by IEEE water fall and spiral are cited as Software Life Cycles

Again here the definition by IEEE of Software Life Cycles coincides with the definition of Software process model provided by Sommerville.

It is clear that the terms Software Life Cycle and Software Process Model mean different things to different authors this research will use the term Software Process Model to mean a simplified representation of a software process, presented from a specific perspective (Sommerville I, 2001) examples being Waterfall spiral ,prototyping etc. The meaning of the term Software Life Cycle will be taken from IEEE (1990), which is the period of time that begins when a software product is conceived and ends when the software is no longer available for use.

Because of the above mentioned inconsistencies the Switch framework will introduce the term Software Development Model which essentially this is the same as a Software Process Model (according to the definition adopted above). The major factor that necessitates the difference in terminology is that Switch will introduce additional concepts to the idea of Software Process Models in particular the Switch frame work will concentrate on the development phase and exclude the maintenance and retirement phase of the Software Life Cycle.

Software Project

Hraconsulting-ltd in their paper Project Definition (2006) Point out that a key part of project definition is establishing the project scope, According to the paper a project can only be considered properly defined when several other things are established:

- who will fill each project role (project manager, project sponsor, etc)
- who will supply what resources and when

- what the project will cost (estimates)
- how long it will take, what will be done when (plan/schedule)
- how work will be tracked, controlled and reported
- what the risks are and how they will be managed
- how the quality of the project's products will be assured

From the above it is clear that the term software project can only be used after a project plan/schedule, risks and their mitigation are established, however the moment plans risk and mitigations plans are produced the software project is already underway. Logical it is not possible to start a project that does not exist for this reason the Switch framework will provide a less restrictive and more functional definition of software project. With the same intent Switch will use a more descriptive term **Software Development Project** as opposed to Software Project or just Project as some authors do.

2.3 Functional and Non-Functional Requirements

Functional Requirement (Function)

A Functional Requirement is a requirement that, when satisfied, will allow the user to perform some kind of function. For the most part business requirements are Functional Requirements which are generally referred to as just “requirements”. Functional Requirements have the following characteristics:

- use simple language
- not ambiguous
- contain only one point

- specific to one type of user
- are qualified
- describe what and not how

Non-Functional Requirement

A Non-Functional Requirement is usually some form of constraint or restriction that must be considered when designing the solution. For the most part constraints are referred to as Non-Functional Requirements. Non-Functional Requirements have the same following characteristics:

- use simple language
- not ambiguous
- contain only one point
- specific to one type of user
- are qualified
- describe what and not how

Non-Functional requirements tend to identify "user" constraints and "system" constraints. Business requirements are kept pure and do not reflect any solution thinking. A system constraint is a constraint imposed by the system and not dictated by a business need. Non-Functional Requirements can be further classified into such categories as "Performance Constraints, Design Constraints, Quality Constraints, etc. It is clear from these classifications that Non- Functional

Requirements can be used as an indicator as to what processes are to be used when developing the system. This concept will become important when defining Model Determinants in the Switch Framework.

2.4 Generic Software Process Models

The software process model maybe defined as a simplified description of a software process, presented from a particular perspective (Adrian Als, Charles Greenidge & P. Walcott, 2004). In essence, each stage of the software process is identified and a model is then employed to represent the inherent activities associated within that stage. Consequently, a collection of ‘local’ models may be utilised in generating the global picture representative of the software process. Examples of Generic models include the workflow model, the data-flow model, and the role model.

- **The workflow model** shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in the model represent human actions.

- **The dataflow model** represents the process as a set of activities each of which carries out some data transformation. It shows how the input to the process such as specification is transformed to an output such as design. The activities here maybe lower than in a workflow model. They may represent transformations carried out by people or computers.

- **The role model** represents the roles of people involved in the software process and the activities for which they are responsible.

2.5 Waterfall Model

According to Sommerville I (2001) the *Waterfall Model* is the earliest method of structured system development. Although it has come under attack in recent years for being too rigid and unrealistic when it comes to quickly meeting customer's needs, the *Waterfall Model* is still widely used. It is attributed with providing the theoretical basis for other *Process Models*, because it most closely resembles a "generic" model for software development.

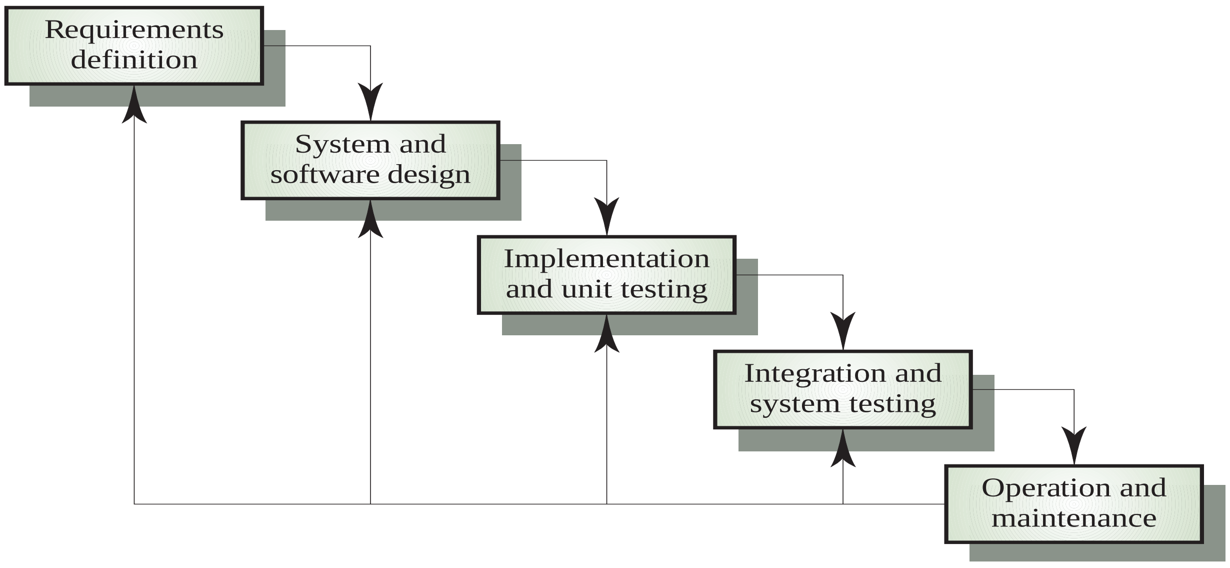

Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Figure 6: Waterfall Software Process Model (Source Software Engineering)

The *Waterfall Model* consists of the following steps:

- **System Conceptualization.** System Conceptualization refers to the consideration of all aspects of the targeted business function or process, with the goals of determining how each of those aspects relate to one another, and which aspects will be incorporated into the system.
- **Systems Analysis.** This step refers to the gathering of system requirements, with the goal of determining how these requirements will be accommodated in the system. Extensive communication between the customer and the developer is essential.
- **System Design.** Once the requirements have been collected and analyzed, it is necessary to identify in detail how the system will be constructed to perform necessary tasks. More specifically, the System Design phase is focused on the data requirements (what information will be processed in the system?), the software construction (how will the application be constructed?), and the interface construction (what will the system look like? What standards will be followed?).
- **Coding.** Also known as programming, this step involves the creation of the system software. Requirements and systems specifications from the System Design step are translated into machine readable code.
- **Testing.** As the software is created and added to the developing system, testing is performed to ensure that it is working correctly and efficiently. Testing is generally focused on two areas: internal efficiency and external effectiveness. The goal of external effectiveness testing is to verify that the software is functioning according to system design, and that it is performing all necessary functions or sub-functions. The goal of internal testing is to make sure that the computer code is efficient, standardized, and well documented. Testing can be a labour-intensive process, due to its iterative nature.

Weaknesses of the Waterfall model

According to Futrell RT, Shafer D and L (2002) when the waterfall model is applied to a project it is not suited if it has the following weaknesses:

- It has an inherently linear sequential nature- any attempt to go back two or more phases to correct a problem or deficiency results in major increases in cost and schedule.
- It does not handle reality of iterations among phases that are so common in software development because it is modelled after a conventional hardware engineering cycle.
- It doesn't reflect the problem-solving nature of software development. Phases are tied rigidly to activities, not how people or teams really work.
- It can present a false impression of status and progress- "35 percent done" is a meaningless metric for the project manager.
- Integration happens in one big bang at the end. With a single pass through the process, integration problems usually surface too late. Previously undetected errors or design deficiencies will emerge, adding risk with little recovery time.
- There is insufficient opportunity for a customer to preview the system until very late in the life cycle. There are no tangible interim deliverables for the customer; user responses cannot be fed back to developers. Because a completed product is not available until the end of the process, the user is involved only in the beginning, while gathering requirements, and at the end, during acceptance testing.

- Users can’t see quality until the end. They can’t appreciate quality if the finished product can’t be seen.
- It isn’t possible for the user to get used to the system gradually. All training must occur at requirements, but still not be operational.
- Each phase is a prerequisite for succeeding activities, making this method a risky choice for unprecedented systems because it inhibits flexibility.
- Deliverables created for each phase are considered frozen- that is, they should not be changed later in the life cycle of the product. If the deliverable of a phase changes, which often happens, the project will suffer schedule problems because the model did not accommodate, nor was the plan based on managing a change later in the cycle.
- All requirements must be known at the beginning of the life cycle, yet customers can rarely state all explicit requirements at that time. The model is not equipped to handle dynamic changes in requirements over the life cycle, as deliverables are “frozen.” The model can be very costly
- To use if requirements are not well known or are dynamically changing during the course of the life cycle.
- Tight management and control is needed because there is no provision for revising the requirements.
- It is document driven, and the amount of documentation can be excessive.
- The entire software product is being worked on at the same time. There is no way to partition the system for delivery of pieces of the system. Budget problems can occur because of commitments to develop an entire system at one time. Large sums of money are allocated with little flexibility to reallocate the funds without destroying the project in process.
- There is no way to account for behind-the-scenes rework and iterations.

When to use Waterfall model

Because of its weaknesses, application of the waterfall model should be limited to situations in which the requirements are well understood. The waterfall model performs for product cycles with a stable product definition and well-understood technical methodologies.

If a company has experience in building a certain genre of system- accounting , payroll, controllers ,compilers, manufacturing- then a project to build another of the same type of product, perhaps even based on existing designs, could make efficient use of the waterfall model. Another example of appropriate use is the creation and release of a new version of an existing product, if the changes are well defined and controlled. Porting an existing product to a new platform is often cited as an ideal project for use of the waterfall.

In all fairness, critics of this model must admit that the modified version of the waterfall is far less rigid than the original, including iterations of phases, concurrent phases, and change management. Reverse arrows allow for iterations of activities within phases. To reflect concurrence among phases, the rectangles are often stacked or the activities within the phases are listed beneath the rectangles showing the concurrence. Although the modified waterfall is much more flexible than classic, it is still not the best choice for rapid development projects.

Waterfall models have historically been on large projects with multiple teams and team members.

2.6 Prototyping Model

The approach is to construct a quick and dirty partial implementation of the system during or before the requirements phase for the purpose of better understanding the requirements (throw away prototyping). When used this way prototyping has high overheads as considerable resources may be dedicated for the construction of a prototype that will eventually be thrown away.

With evolutionary prototyping the prototype evolves to become the solution resulting in an unstructured solution which is very difficult to maintain and has large-system integration problems.

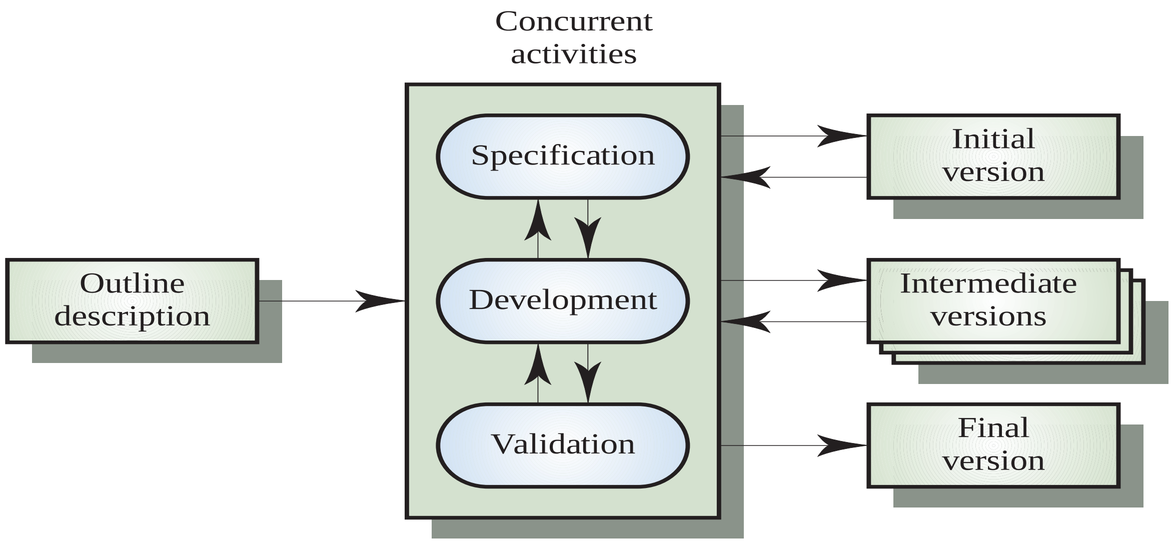


Figure 7: Evolutionary prototyping (Source Software Engineering)

Weaknesses of the Structured evolutionary Rapid Prototype

When applied to a project for which it is not suited, the weaknesses of this model can be seen in the following ways (Futrell R, Shafer D and L 2002):

- The model may not be accepted due to a reputation among conservatives as a “quick-and-dirty” method.

- Quick-and-dirty prototypes, in contrast to evolutionary rapid prototypes, suffer from inadequate or missing documentation.
- If the prototype objectives are not agreed upon in advance, the process can turn into an exercise in hacking code.
- In the rush to create a working prototype, overall software quality or long term maintainability may be overlooked.
- Sometimes a system with poor performance is produced, especially if the tuning stage is skipped.
- There may be a tendency for difficult problems to be pushed to the future, causing the initial promise of the prototype not to be met by subsequent products.
- If the users cannot be involved during the rapid prototype iteration phase of the life cycle, the final product may suffer adverse effects, including quality issues.
- The rapid prototype is a partial system during the iteration phase. If the project is cancelled, the end user will be left with only a partial system.
- The customer may expect the exact same “look and feel” of the prototype. In fact, it may have to be ported to a different platform, with different tools to accommodate size or performance issues, resulting in different user interface.
- The customer may want to have the prototype delivered rather than waiting for full, well-engineered version.
- If the prototyping language or environment is not consistent with the production language or environment, there can be delays in full implementation of the production system.

- Prototyping is habit-forming and may go on too long. Undisciplined developers may fall into a code-and-fix cycle, leading to expensive, unplanned prototype iterations.
- Developers and users don't always understand that when a prototype is evolved into a final product, traditional documentation is still necessary. If it is not present, a later retrofit can be more expensive than throwing away the prototype.
- When customers, satisfied with a prototype, demand immediate delivery, it is tempting for the software development project manager to relent.
- Customers may have a difficult time knowing the difference between a prototype and a fully developed system that is ready for implementation.
- Customers may become frustrated without the knowledge of the exact number of iterations that will be necessary.
- A system may be over evolved; the iterative process of prototype demonstration and revision can continue forever without proper management. As users see success in requirements being met, they may have a tendency to add to the list of items to be prototyped until the scope of the project far exceeds the feasibility study.
- Developers may take less-than-ideal choices in prototyping tools (operating systems, languages, and inefficient algorithms) just to demonstrate capability.
- Structured techniques in the name of analysis paralysis avoidance. The same "real" requirements analysis and attention to quality for maintainable code is necessary with prototyping, as with any other life cycle model (although they may be produced in smaller increments).

When to use the Structured Evolutionary Rapid Prototype Model

A project manager may feel confident that the structured evolutionary rapid prototyping model is appropriate when several of the following conditions are met:

- When requirements are not known-up front;
- When requirements are unstable or may be misunderstood or poorly communicated;
- For requirements clarification;
- When developing user interfaces;
- For proof-of –concept;
- For short-lived demonstrations;
- When structured, evolutionary rapid prototyping may be used successfully on large systems where some modules are prototyped and some are developed in a more traditional fashion;
- On new, original development (as opposed to maintenance on an existing system);
- When there is need to reduce requirements uncertainty-reduces risk of producing a system that has no value to the customer;
- When developers are unsure of the optimal architecture or algorithms to use;
- When algorithms or system interfaces are complex;
- To demonstrate technical feasibility when the technical risk is high;
- On high-technology software-intensive systems where requirements beyond the core capability can be generally but not specifically identified;
- During software acquisition, especially on medium-to-high-risk programs;

- In combination with the waterfall model- the front end of the project uses prototyping, and the back end uses waterfall phases for system operational efficiency and quality;
- Prototyping should always be used with the analysis and design portions of object-oriented development.

Rapid prototyping is well suited for user interface intensive systems, such as display panels for control devices, interactive online systems, first-of-a-kind products, and decision support systems such as command and control or medical diagnosis, among others.

2.7 Spiral Model

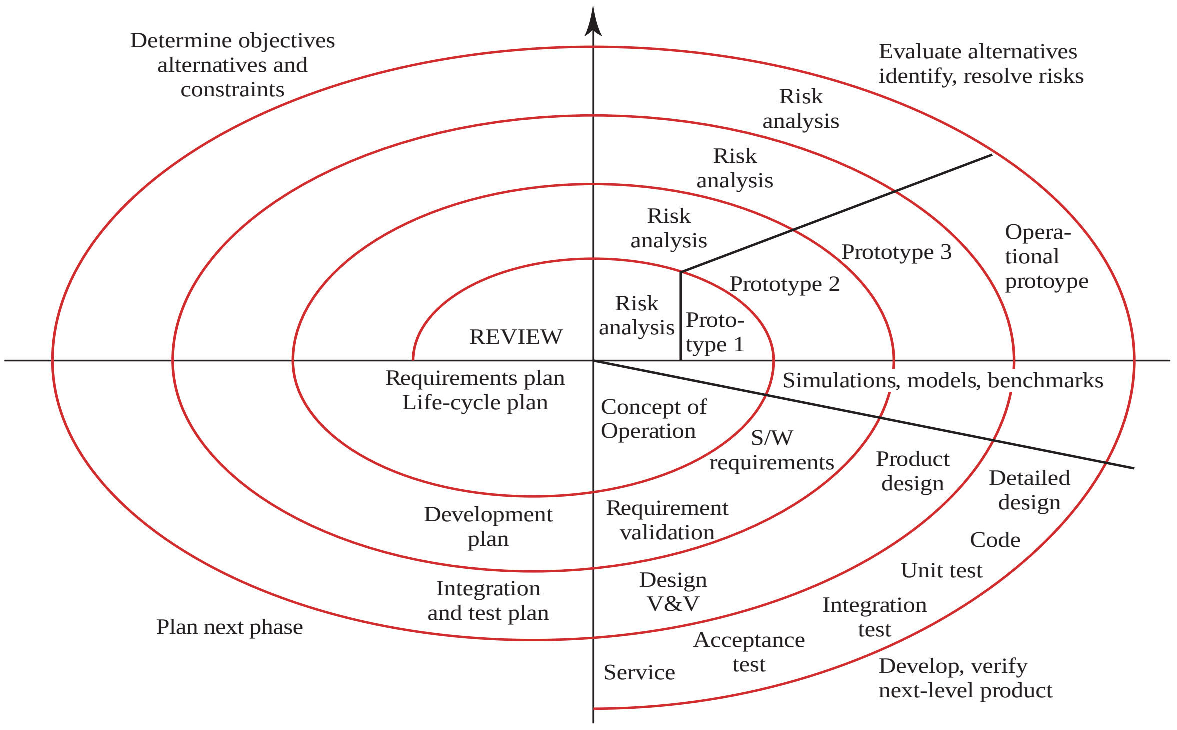


Figure 8: Spiral model (Source Software Engineering)

Weaknesses of the Spiral Model

When applied to a project for which it is not well suited, weaknesses of the spiral model include the following:

- If the project is low risk or small, this model can be an expensive one. The time spent evaluating the risk after each spiral is costly.
- The model is complex, and developers, managers, and customers may find it too complicates to use.
- Considerable risk assessment expertise is required.
- The spiral may continue indefinitely, generated by each of the customer's responses to the build initiating a new cycle; closure (convergence on a solution) may be difficult to achieve.
- The large number of intermediate stages can create additional internal and external documentation to process.
- Use of the model may be expensive and even unaffordable- time spent planning, resetting objectives, doing risk analysis, and prototyping tool or technique can make this model clumsy to use.
- The industry has not had so much experience with the spiral model as it has with others.

Some users of this technique have found the original spiral model to be complex and have created a simplified version.

When to use the Spiral Model

A project manager may feel confident that the spiral model is appropriate when several of the following conditions are met;

- When the creation of a prototype is the appropriate type of product development;
- When it is important to communicate how costs will be increasing and to evaluate the project for costs during the risk quadrant activities;
- When organizations have the skills to tailor the model;
- For projects that represent a medium to high risk;
- When it is unwise to commit to a long term project due to potential changes in economic priorities and when these uncertainties may limit the available time frame;
- When technology is new and tests of basic concepts are required;
- When users are unsure of their needs;
- When requirements are complex;
- For a new function or production line;
- When significant changes are expected, as with research or exploration;
- When it is important to focus on stable or known parts while gathering knowledge about changing parts;
- For large projects;
- For large organizations that cannot afford to allocate all the necessary project money up front, without getting some back along the way;
- On long projects that may make managers or customers nervous;
- When benefits are uncertain and success is not guaranteed;
- To demonstrate quality and attainment of objectives in short period of time;
- When technologies are being employed such as first time object-oriented approaches;

- With computation-intensive systems such as decision support systems;
- With business projects as well as aerospace, defence, and engineering projects, where the spiral model already enjoys popular use;

2.8 Rapid Application Development

Weaknesses of the RAD model

Weaknesses of this model when applied to a project for which it is not well suited include the following;

- If the users cannot be involved consistently throughout the life cycle, final product will be adversely affected.
- This model requires highly skilled and well trained developers in the use of the chosen development tools to achieve the rapid turnaround time.
- It requires a system that can be properly modularized.
- It can fail if reusable components are not available.
- It may be hard to use with legacy systems and many interfaces.
- It requires developers and customers who are committed to rapid –fire activities in an abbreviated time frame.
- Blindly applied, no bounds are placed on the cost or completion date of the project.
- Teams developing commercial projects with RAD can over evolve the product and never ship it.

- There is a risk of never achieving closure –the project manager must work closely with both the development team and the customer to avoid an infinite loop.
- An efficient, accelerated development process must be in place for quick response to user feedback.

When to use RAD model

A project manager may feel confident that the RAD model is appropriate when several of the following conditions are met:

- On systems that may be modularised (component –based construction) and that are scalable ;
- On systems with reasonably well-known requirements;
- When the end user can be involved throughout the life cycle;
- When users are willing to become heavily involved in the use of automated tools;
- On projects requiring short development times usually about 60 days;
- On systems that can be time-boxed to deliver functionality in increments;
- When reusable parts are available through automated software repositories;
- On systems that are proof-of-concept, non-critical, or small;
- When cost and schedule are not a critical concern (such as internal tool development);

- On systems that do not require high performance, especially through tuning interfaces;
- When technical risks are low;
- On information systems;
- When the project team is familiar with the problem domain, skilled in the use of the development tools, and highly motivated.

2.9 Agile Methods

In software development there exists a tension between quality, cost, and time. Delivering cost competitive quality software is a difficult task. Many traditional software processes are top heavy with documentation and rigid control mechanisms making it difficult applying them to different software projects. New families of processes, referred to as Agile Processes, are making headway into the software industry. These processes focus on code rather than documentation calling themselves agile because, unlike the traditional processes, they are adaptable, not rigid.

The Manifesto for Agile Software Development (The Agile Alliance, 2001) is given below.

  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan

Principles behind the Agile Methods

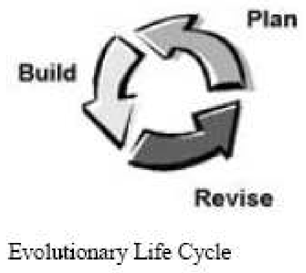
- The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Changing requirements are welcome, even late in the development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

There are many variations of agile processes an overview of four of the most popular processes is provided. The processes to be looked at are Adaptive Software Development (ASD), Extreme Programming (XP), Crystal, and the Rational Unified Process (RUP).

2.10 Adaptive Software Development

ASD was developed by Jim Highsmith (2000) and it does not provide the details often associated with a process. Milestones, methods, and deliverables are not specific elements discussed by ASD. Instead, ASD places its emphasis on applying ideas originating in the world of complex adaptive systems (i.e. Chaos theory). ASD provides the fundamental base to develop adaptive systems from which arise agile and adaptive processes.

At the core of ASD is the premise that outcomes are naturally unpredictable and, therefore, planning is a paradox. It is not possible to plan successfully in a fast moving and unpredictable business environment. Adaptive development is essential when you have developers, customer, vendors, competitors and, stockholders all attempting to interact with one another at such a pace that linear cause and effect rules cannot assure success. ASD replaces the evolutionary life cycle model with the adaptive cycle model (fig 9).



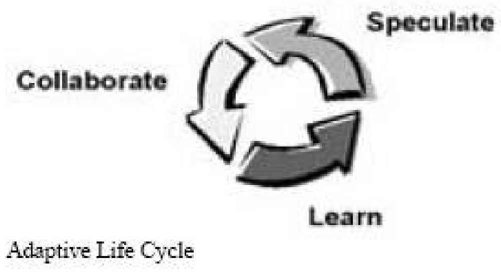


Figure 9: Adaptive and Evolutionary Life Cycles (Source Everette R. Keith 2002)

Comparing the two models, ASD recognizes that it is not desirable to wander around experimenting endlessly hoping to find success. On the other hand, the word planning is too deterministic for our mostly unpredictable world so ASD prefers the word Speculate. Develop a good idea of where the project is heading, and put mechanisms in place to adapt to changing customer needs, changing technology and a changing market.

Collaboration replaces build because of ASD's recognition that people are the essential ingredient to producing a successful product. Collaboration is the activity of balancing: managing a project, such as configuration control and change management, with creativity the act of trusting people to find creative answers in an unpredictable environment. Completing the cycle is Learning.

Learning is preferred over revise because revise is backward looking. Revise, in the context of the evolutionary life cycle implies that while change may be necessary it is change based on the original plan. Learning, however, is the act of gaining knowledge through experience. Placing the product under scrutiny and questioning all previous assumptions, using the results of the cycle to learn which direction the next cycle should take. ASD is not presented as a methodology for doing software projects but rather it is an approach or an attitude that must be adopted by an organization when applying agile processes. Adopting the premise that the world is fast paced and unpredictable requiring adaptive processes to compete and be successful is the message communicated by the author of ASD, Mr. Jim Highsmith.

In reference to the Switch frameworks ASD is not a software process model or in Switch's own terms ASD is not a software development model. However the ideologies presented by ASD can easily be embraced by the Switch framework by means of ASD framework extension.

2.11 Extreme Programming

Extreme Programming (XP) is a high profile agile process known to many advocates and novices alike and is likely the most widely used (fig. 10). There are four basic values supporting XP:

- Communication – Without communications project schedules slip, quality suffers, and the customer's wants and needs are misinterpreted or overlooked.

- Feedback – The need to check our results is important. Without feedback, a project will most likely fail. Feedback tells how a project is doing while providing direction for future iterations.

- Simplicity – Do not add unnecessary artefacts or activities to a project. Eliminate everything not completely justified.

- Courage – Putting faith in the people over the process requires courage. It is important to realize, however, that if processes do become more important than the people do, a project is headed toward failure.

Extreme Programming has defined practices and guidelines that implementers should follow. The process begins by gathering stories. These are short use cases, small enough to fit on an index card. Each story is business-oriented, testable, and estimable. From the stories, the customer selects the most valuable set. This set comprises of iteration and is coded first. Coding is done in pairs, two people coding on one machine, and an iteration is typically one to two weeks long. Once complete the set is tested and put into production. "The goal of each iteration is to put into production some new stories that are tested and ready to go."

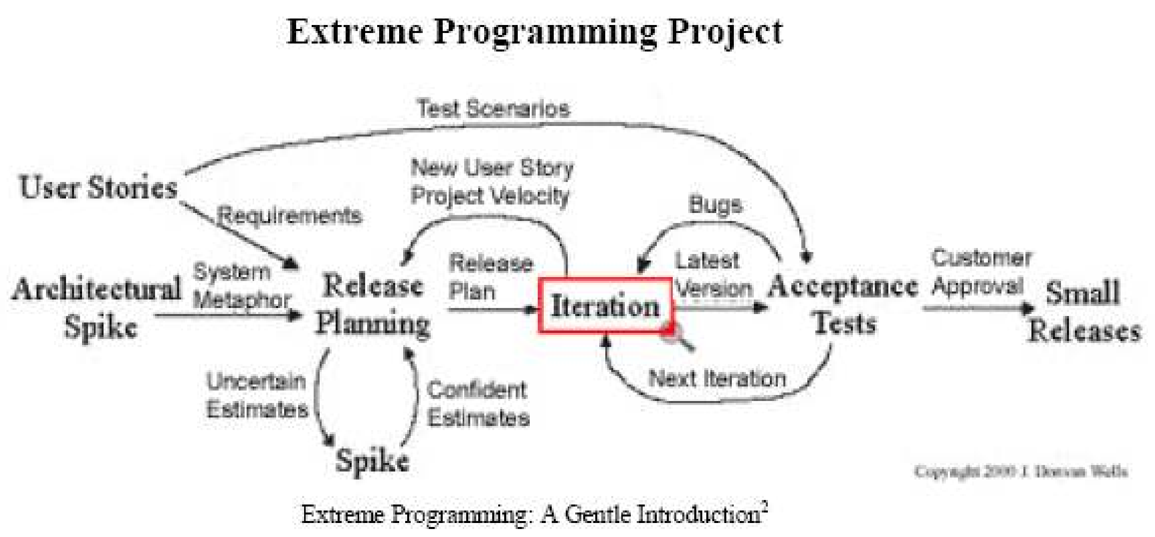


Figure 10 Extreme Programming (Source Extreme Programming, 2002)

Testing plays a major role in XP. Each iteration is subjected to unit testing. Writing all unit tests prior to writing any code is mandatory. A particular iteration must pass its unit testing prior to going into production. Customers determine system wide tests. Considering their needs and referencing the stories, customers think about what it would take to satisfy them that the iteration is successful. These needs are translated into system wide tests. Testing regularly and often at the unit level and system level provides feedback and confidence that the project is moving ahead and the system is functioning according to the customer's requirements.

This process of selecting a set of stories, doing short iterations, working in pairs to code, test, and integrate is repeated until the project is complete. Working in short iterations with constant feedback gives the project the chance to adapt to changing needs. The focus is always on the current iteration. No design work is done in anticipation of future requirements. XP is a highly disciplined process. To be successful the organization implementing XP must embrace the XP values and principles. Its proponents would advise that it should be implemented for the first time on small to medium sized projects and that all process steps should be explicitly followed until experience is gained using it. It should how ever be noted that the concept of project completion is very elusive with XP as there is not limit to the number of times a story can be refined. For a large project XP becomes unsuitable as these have high integration needs.

2.12 Crystal

Crystal is a family of processes each applied to different kinds of projects. The idea of having multiple processes stems from the thinking that some projects require fewer rigors than others do. Small and non-critical projects can be developed using less rigorous Crystal methods. Larger and more critical projects, however, demand more attention and therefore, a more rigorous Crystal process is used.

Selecting a Crystal process requires that a project be matched to one of four critical levels.

- Comfort
- Discretionary money
- Essential money
- Life

A system failure for the first level may cause a loss of comfort whereas a system failure for the fourth level may cause a loss of life. Using this as an example a less rigorous process is applied to the former while the latter would demand a highly rigorous process. Each of the processes share common policy standards.

- Incremental delivery
- Progress tracking by milestones based on software deliveries and major decisions rather than written documents
- Direct user involvement
- Automated regression testing of functionality
- Two user viewings per release
- Workshops for product and methodology-tuning at the beginning and in the middle of each increment.

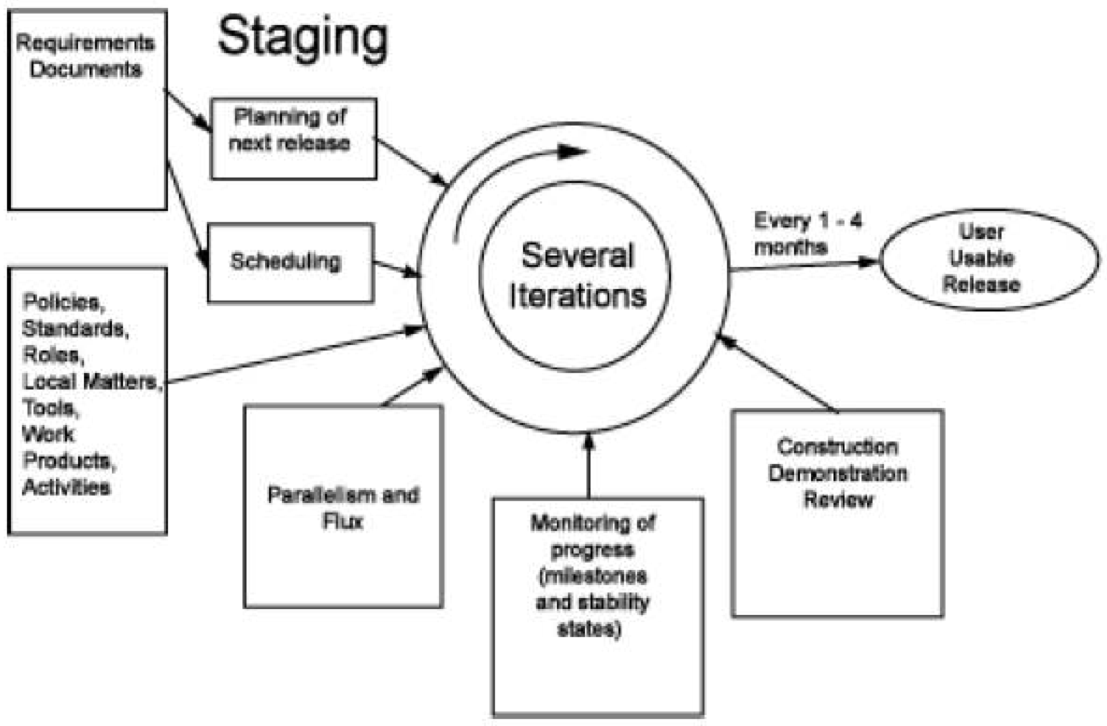

Staging
Requirements Documents
Planning of next release
Scheduling
Policies, Standards, Roles, Local Matters, Tools, Work Products, Activities
Several Iterations
Every 1 - 4 months
User Usable Release
Parallelism and Flux
Monitoring of progress (milestones and stability states)
Construction Demonstration Review

Figure 11: One Crystal Orange Increment (Source Agile Software Development Methods)

While each of the Crystal processes share the standards the rigor involved is dependent on the project and the chosen process. For example, less critical projects suggest two-month incremental delivery time spans whereas critical projects, demanding a rigorous process, may extend time-to-delivery increments to a maximum of four months. Projects are comprised of multiple increments (fig.11). Crystal processes define the functions contained in an increment.

The functions or practices are explicitly defined. Staging, this is the scheduled time frame for one increment ranging from one to four months duration. Increments include several iterations during which construction, demonstration, and reviews are included activities. Iterations are monitored and team deliverables gage the progress and stability of an iteration. User viewings, (i.e. reviews by users), are conducted one to three times per iteration depending on the criticality of the project. Methodology-tuning is a basic technique used to fix and improve the process applying knowledge gained in one iteration to the next iteration. In addition to the methodology-tuning workshops there are reflection workshops conducted at the start of an increment and midway into it.

The multiple processes offered by the Crystal methodologies are adaptive and agile. They are adaptive because they offer multiple solutions for projects having different criteria. They are agile because they deliver work products at the completion of each project increment and are able to apply lessons learned to the next iteration. Additionally, they demand customer involvement, and decisions are made based on outcomes rather than trying to force conformance to a plan developed early in a project's phase.

Of all the Software Process Model currently available Crystal is probably the closest to the Switch Framework because it embraces the concept that each software project is unique by introducing the concept of "criticality levels". Switch extends the idea that software projects are unique by actually allowing for the construction of a Software Development Model for the given project. Furthermore in the Switch framework critical levels will be viewed as part of Model Determinants, which are a more comprehensive way of looking at the attributes of a software project.

2.13 Rational Unified Process

The Rational Unified Process (RUP) is a generic process framework that uses a specific methodology to accomplish the tasks associated with it. The RUP uses the Unified Modelling Language developing use cases for the software system design. The nature of its versatile frameworks something that different organizations can apply to different applications using people with different skill levels working on projects with varying levels of complexity. How it is used depends on the environment and how the project chooses to tailor it.

In its simplest form its workflows mimic classic BDUF waterfall processes. Gain knowledge of your customer's business, translate the business needs into a set of requirements, do analysis and design creating the software architecture, implement the design and test it extensively prior to delivering it to the customer. These attributes are core to RUP's framework. The core attributes are applied to four unique phases occurring over the lifecycle of a RUP project.

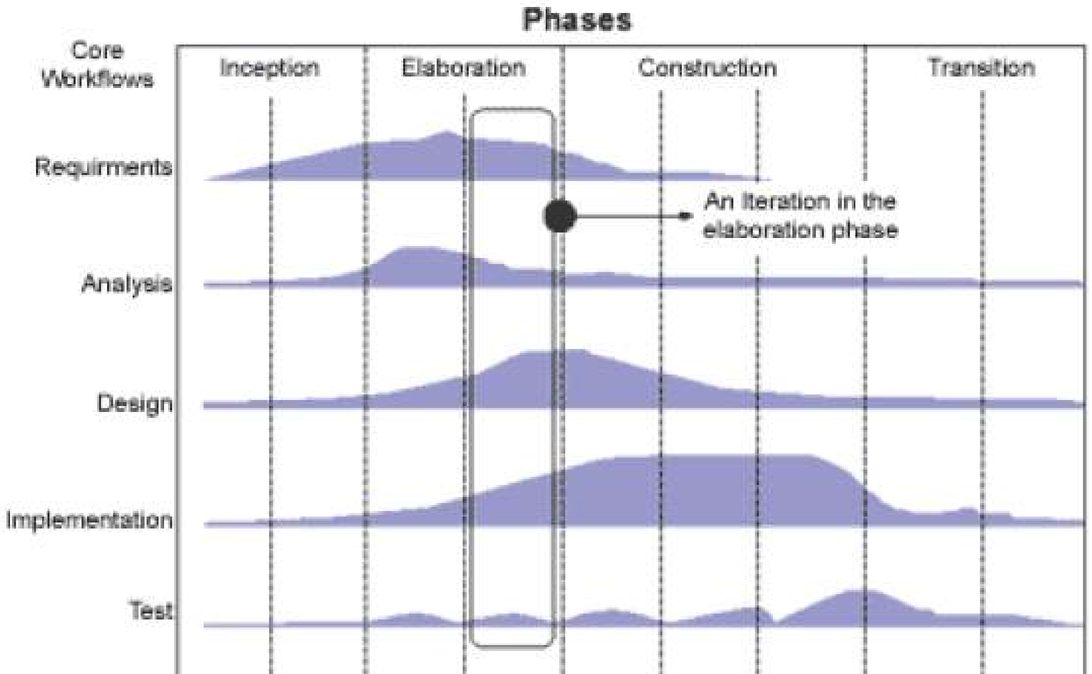


Figure 12: RUP workflows and phases (Source: The Unified Process. 1999)

The tailoring occurs by controlling the iterations (fig. 4). The organization and the project complexity drive the iterations. Organizations could use RUP in its simple configuration, one iteration for each phase, creating a waterfall style process or use it in a more agile manner creating multiple iterations, having lengths of one to two weeks. Ultimately, “the goal of each iteration is to develop some working software that can be demonstrated to all the stakeholders, and that the stakeholders will find it meaningful.”

Each phase iterates through each workflow while the project lifecycle moves through the phases. Inception is concerned with gathering requirements and putting together the project plan. During elaboration the software architecture is defined and the detailed plan for future iterations is established. During construction the software system is built and tested and the user documentation is completed. Finally, the Transition phase closes the project and delivers the product to the customer.

RUP is tailored according to an organization’s culture and by project complexity making it highly versatile. It uses defined workflows to complete its work products and has four phases defining the process life cycle. In many ways it resembles a classic waterfall styled process, however, because of its versatility it is generally tailored for use as an agile software process.

2.14 Software Process Model Selection

In the Book Quality Software Project Management (Futrell RT, Shafer D and L 2002) a framework for selecting a Software Process Model for use with a particular software project is provided. Futrell and his colleagues are among the first writers in the field of Software Engineering to provide such a framework, this is provided below.

The selection of an appropriate life cycle model for a project can be accomplished by using the following process:

1. Examine the following project characteristics categories, as demonstrated in Tables below:
    - Requirements: Table 1
    - Project Team: Table 2
    - User community: Table 3
    - Project type and risk: Table 4

2. Answer the questions presented for each category by circling a yes or no in the matrices provided.
3. Rank the importance of the category, or question within the category, in terms of the project for which you are selecting a life cycle model.
4. Total the number of circled responses for each column in the matrices to arrive at an appropriate model.
5. Use the category ranking to resolve conflicts between models if the totals are close or the same.

Project Characteristics Categories

A brief description of the characteristics of requirements, project team, user community, and project type and risk follow. Table 1 through table 4 provide a set of matrices for use in steps 1-5 of the life cycle model selection process described in the preceding section.

Requirements

The requirements category (Table 1) consists of questions related to things that have been requested by the user for the project. They are sometimes termed as functions or features of the system that will be provided by the project.

Table 1: Selecting a life cycle Model based on characteristics of requirements

Requirements	Waterfall	V-Shaped	Prototype	Spiral	RAD	Incremental
Are the requirements easily defined and/or well known?	Yes	Yes	No	No	Yes	No
Can the requirements be defined early in the cycle?	Yes	yes	No	No	Yes	Yes
Will the requirements change often in the cycle?	No	No	Yes	Yes	No	No
Is there a need to demonstrate the requirements to achieve definition?	No	No	Yes	Yes	Yes	No
Is a proof of concept required to demonstrate capability?	No	No	Yes	Yes	Yes	No

Do the requirements indicate a complex system?	No	No	Yes	Yes	No	Yes
Is early functionality a requirement?	No	No	Yes	Yes	Yes	Yes

Project Team

Whenever possible, it is best to select the people for the project team before selecting the life cycle model. The characteristics of this team (Table 2) are important in the selection process because they are responsible for the successful completion of the cycle, and they can assist in the selection process.

Table 2: Selecting life cycle Model based on characteristics of the Project team

Project team	**Waterfall**	**V-Shaped**	**Prototype**	**Spiral**	**RAD**	**Incremental**
Are the majority of the team members new to the problem domain for the project?	No	No	Yes	Yes	No	No
Are the majority of team members new to the technology domain for the domain?	Yes	Yes	No	Yes	No	Yes
Are the majority of team members new to the tools to be used on the project?	Yes	Yes	No	Yes	No	No

Are the team members subject to reassignment during the life cycle?	No	No	Yes	Yes	No	Yes
Is there training available for the project team, if required?	No	Yes	No	No	Yes	Yes
Is the team more comfortable with structure than flexibility?	Yes	Yes	No	No	No	Yes
Will the project manager closely track the team's progress?	Yes	Yes	No	Yes	No	Yes
Is ease of resource allocation important?	Yes	Yes	No	No	Yes	Yes
Does the team accept peer reviews and inspections, management/customer reviews, and milestones?	Yes	Yes	Yes	Yes	No	Yes

User community

The early project phases can provide a good understanding of the user community (Table 3) and the expected relationship with the project team for the duration of the project. This understanding will assist you in the appropriate model because some models are dependent on high user involvement and understanding of the project.

Table 3: Selecting a life cycle model on characteristics of the User Community

User community	**Waterfall**	**V-Shaped**	**Prototype**	**Spiral**	**RAD**	**Incremental**
Will the availability of the user representatives be restricted or limited during the life cycle?	Yes	Yes	No	Yes	No	Yes
Are the user representatives new to the system definition?	No	No	Yes	Yes	No	Yes
Are the user representatives' experts in the problem domain?	No	No	Yes	No	Yes	Yes
Do the users want to be involved in all phases of the life cycle?	No	No	yes	No	Yes	No
Does the customer want to track project progress?	No	No	Yes	Yes	No	No

Project Type and Risk

Finally, examine the type of project and the risk (Table 4) that has been identified to this point in the planning phase. Some models are designed to accommodate high-risk management while others are not. The selection of a model that accommodates risk management does not mean that you do not have to create an action plan to minimize the risk identified. The model simply provides a framework within which this action plan can be discussed and executed.

Table 4: Selecting a life cycle model based on characteristics of project type and risk

Project type and risk	**Waterfall**	**V-Shaped**	**Prototype**	**Spiral**	**RAD**	**Incremental**
Does the project identify a new product direction for the organization?	No	No	Yes	Yes	No	Yes
Is the project a system integration project?	No	Yes	Yes	Yes	Yes	Yes
Is the project an enhancement to an existing system?	No	Yes	No	No	Yes	Yes
Is the funding for the project expected to be stable throughout the life cycle?	Yes	Yes	Yes	No	Yes	No
Is the product expected to have long life in the organization?	Yes	Yes	No	Yes	No	Yes
Is high reliability a	No	Yes	No	Yes	No	Yes

must?						
Is the system expected to be modified, perhaps in ways not anticipated, post-deployment?	No	No	Yes	Yes	No	Yes
Is the schedule constrained?	No	No	Yes	Yes	Yes	Yes
Are the module interfaces clean?	Yes	Yes	No	No	No	yes
Are reusable components available?	No	No	Yes	Yes	Yes	No
Are resources (time, money, tools, and people) scarce?	No	No	Yes	Yes	No	No

The Software Process Model selection framework is very useful as it helps to automate the laborious and daunting task of selecting a SPM. A problem arises when two models have desirable characteristics for the software project at hand selecting anyone of the model directly results in loosing all of the required characteristics of the model that is not selected. Switch intends to solve this problem by tacking the concept a step further, one does not select a model but instead a model is constructed this approach ensures that all the desired characteristics are part of the model.

2.15 Software Process Model Customisation

For most Software Process Models (SPMs) customisation is not a defined process and those that do have the leeway for customisation the process is more of an after thought the end result is that any attempts of customising a SPM usually results in a model that is none functional.

2.16 Software Capability Maturity Model

The Software Capability Maturity Model (SW-CMM) provides a set of requirements that organizations can use in setting up a software process used to control software product development. The SW-CMM specifies “what” should be in the software process
but not “when” or “for how long.” This is one of the major differences with the Switch framework which will state what processes are to be in the model and when they should be executed (sequence of execution).

The SW-CMM has what is called a process maturity framework (Paulk et al. 1995). There are five levels of process maturity, Level 1 (lowest) to Level 5 (highest). To be rated at a specific level, an Organization has to demonstrate capabilities in a number of Key Process Areas (KPA) associated with a specific SW-CMM level, Table 1. The capabilities demonstrated in transitioning from lower levels to higher levels are cumulative. In other words, a Level 3 Organization must demonstrate KPA capabilities from Level 2 and from Level 3. The Process Maturity framework is presented in Table 5 below.

Table 5: Process Maturity Framework

SW-CMM Level	Key Process Areas
Level 1	None
Level 2 Repeatable	Requirements Management
	Software Project Planning
	Software Project Tracking and Oversight
	Software Subcontract Management
	Software Quality Assurance
	Software Configuration Management
Level 3 Defined	Organization Process Focus
	Organization Process Definition
	Training Program
	Integrated Software Management
	Software Product Engineering
	Intergroup Coordination
	Peer Reviews
Level 4 Managed	Quantitative Process Management
	Software Quality Management
Level 5 Optimizing	Defect Prevention
	Technology Change Management
	Process Change Management

All Organizations start at Level 1. This is called the Initial level. At this level few processes are defined, and success depends on individual effort. This makes the software process unpredictable because it changes as work progresses. Schedules, budgets, functionality, and product quality are generally unpredictable. To achieve Level 2 the organization demonstrates capability in 6 KPAs. A Level 2 Organization has basic management processes established to track cost, schedule, and functionality. Problems in meeting commitments are identified when they arise. Software requirements and work products developed to satisfy requirements are base-lined and their integrity is controlled. Software project standards are defined and the organization ensures they are faithfully followed. The project works with its subcontractors to establish a strong relationship. The necessary process discipline is in place to repeat earlier successes on projects with similar applications. Level 2 is called the Repeatable level.

A Level 3 Organization has demonstrated capabilities in an additional 7 KPAs. At this level the software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the whole organization. Projects tailor the standard software process to develop their own unique defined software process. A well-defined process includes readiness criteria, inputs, standards and procedures for performing the work, verification mechanisms, outputs, and completion criteria. Level 3 is called the Defined level.

A Level 4 Organization has added 2 more KPAs to its capabilities. At this level detailed

measures of the software process and product quality are collected. Projects achieve control over their products and processes by narrowing the variation in their process performance to fall within acceptable quantitative boundaries. Both the process and product are quantitatively understood and controlled. Level 4 is called the Managed level.

At Level 5 an Organization has capabilities in 3 more KPAs and is in a continuous improvement state. Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies. Software project teams analyze defects to determine their causes. Processes are evaluated to prevent known types of defects from recurring, and lessons learned are disseminated to other projects. Level 5 is called the Optimizing level.

From the above it is clear that what ever level an organisation is it can benefit from the automation of its Software Process Models, hence the use of Switch will allow an organisation to consolidate its level classification and in some cases it may assist an organisation to move to a higher level classification.

3.0 Research Methodology

To define and analyse Model Determinants a study will be carried out that looks at current SPMs with emphasis on recommendations of when a SPM is most appropriate for use with a given project.

3.1 Switch Framework Design Method

The Switch Framework will be design using a top down approach, as this is the most appropriate design technique for situations where the internal specification of a design are not fully understood at the time of commencing the design.

In the top-down model an overview of the system is formulated, without going into detail for any part of it. Each part of the system is then refined by designing it in more detail. Each new part may then be refined again, defining it in yet more detail until the entire specification is detailed enough to validate the model. The top-down model is often designed with the assistance of "dark boxes" that make it easier to bring to fulfillment but insufficient and irrelevant in understanding the elementary mechanisms.

3.2 Core Framework Extension Design Method

The Core Extension will be designed using a bottom-up approach. The individual components of the extension will be specified in detail. The components will then be linked together to form larger components, which are in turn linked until the complete extension is formed. This bottom-up information flow seems potentially necessary and sufficient because it is based on the knowledge of all variables that may affect the elements of the core framework extension.

4.0 Results of Findings

As stated in section 1.2 the research problem is concerned with solving the problem of coming up with SPMs that are specific to a given software project. This chapter lays out the method of coming up with project specific models. In essence this chapter is dedicated to the specification of the Switch framework which in turn allows us to construct model on a project by project basis.

The Switch framework is defined as an Extensible Framework for the Generation of Bespoke and Dynamic Software Development Models. The specification provided in this chapter is for version 1.0. The emphasis on versioning is made necessary by the fact that as with all areas of practical application a specification has to be used extensively before it can be perfected it is therefore hoped that after extensive usage with real world software projects the framework will be perfected to higher versions. Lastly this chapter will provide some software projects and their Model Determinants and the corresponding SDM produced by Switch.

4.1 Software Development Projects

Within the context of the Switch framework a Software Development Project is defined as a problem statement that requires the development of some software the non functional requirements for the problem statement the client for whom the software is to be developed and the development team. The availability of non-functional requirements is not essential but all else is mandatory.

4.2 Model Extensions

A Model Extension (ME) is simply a collection of related Model Determinants (MD). A MD that is part of an Extension can not be part of another Extension; however a MD can determine more than one Software Process (SP). There are two types of Model Extensions, Application Domain Extensions (e.g. Multimedia Applications, Database Driven Applications, Web Applications…) and Software Development Paradigm Extensions (e.g. Agile, Formal Transformation, Component Reuse). Extensions enable Switch to be a dynamic and extendable framework, the more extensions are added to Switch the more functional and power the framework becomes. The addition of an Extension to Switch should only be done after a model extension study has been carried out.

Attribute of Model Extensions

Name: - A unique Extension identifier.

Description: - A description of the extension.

Extension Type: - The type of Model Extension, Application Domain or Software Development Paradigm.

Model Determinants: - The set of MDs that belong to the extension.

4.3 Framework Weightings

Weightings are a measure (perceived or actual) of relevance or trueness of the following framework objects.

- Model Determinant (MW:- Mitigation Weighting and LW:- Level Weighting)

- Determining Relationships (DW:- Determining Weighting)
- Software Development Processes (PW:- Process Weighting)

There are two types of Weightings **Boolean** (actual values) and **Scaled** (perceived values). Scaled weightings apply to all the above mentioned objects and Boolean applies to Level Weightings (Model Determinant Levels) only. The Measurement of the Scaled level is from zero to five, five being the highest which means the highest possible relevance and zero the lowest which means the object should be ignored. Boolean levels have values true (weighting value 5) and false (weighting value 0, ignore object). In this version of Switch framework (V1.0) weightings have been set as specified in the table below, however the user is at liberty to define more weighting.

Table 6: Switch V1.0 Framework Weightings

Value	Description	Type
1.0	True	Boolean
0.9	High	Scaled
0.7	Above average	Scaled
0.5	Average	Scaled
0.3	Below average	Scaled
0.1	Low	Scaled
0.0	False	Boolean

A value of 0 means that the object should be ignored; it should not be used in any calculation, or for determining the final SDM in any way.

Table 7: Value Rages for Weighted Objects

Object	Min Value	Max Value
MD Weighting	1	5
Project MD Weighting	0/ False	5/ True

Determining Weighting	1	5
SP Weighting	1	Infinity

4.4 Model Determinants

These are attributes of a software project which lands it to the construction of a unique and custom built Software Development Model (SDM) Non Functional Requirements are a subset of Model Determinants.

Attribute of MDs

The design time attributes of MDs are attributes whose values are set when the MD is created, these are specified below:

Name: - A name is unique MD identifier

Extension Name: - The name of the Extension to which the MD belongs

Level Weighting Type: - The measurement type for Level Weighting, the two possible types are Boolean and Scaled (See section 4.3 Framework Weightings).

Description: - A description of the MD.

Model Determinants have two attributes that are specified at run time (the time when a SDM is constructed for a specific software development project):

Mitigation Weighting: - This is a weighted measure of the effort that must be applied to mitigate undesirable effects in a Software Project in relation to a given MD. A High weighting (5) is an indication that it is very important to include a (Software Development Process) SDP and or SDPs that cater for the undesirable effects in relation to MD. Conversely a Low weighting (1) indicates that a MD is not very important and it’s associated SDPs can be excluded from the Software Development Model (SDM) if need be.

Level Weighting: - This is a weighted measure of the perceived or actual real world level of a MD for a Software Project. There are two types of levels Boolean Levels (actual value) and Scaled Levels (perceived). Measurement of the scaled level is from one to five, five being the highest and one the lowest. Boolean levels have values true and false.

For a given software development project Model Determinants (MD) their Mitigation and Level Weightings are obtained by carrying out a MD Fact Finding Study which is simply a meeting involving the client and the development team.

4.5 Software Development Processes

A Software Development Process (SDP) is a set of logically related activities whose aim is to achieve a certain goal that aids to the development of a software product.

Attribute of Software Development Processes

Name: - Unique SDP identifier.

Description: - A description of what is done in the SDP.

Parent Process: - Name of the process which the SDP is a child process of.

Sequence Number: - Number used to specify order of execution in relation to other SDPs.

Process Weighting: - A construction time attribute which specifies the emphasis to place on the process in relation to other SDP

4.6 Determining Relationships

A Determining Relationship is a relationship between a MD and a SDP. A Model Determinant "A" determines a Software Development Processes "B" means the presence of MD "A" points to the presence of "B" in the Software Development Model for the project that has MD "A". However the presence of "B" in the Software Process Model is determined by the Model Construction Algorithm. One MD can determine many SDPs and a SDP can have many MDs.

Attribute of Determining Relationship

Model Determinant: - The MD that is part of the Determining Relationship.

Software Development Process: - The SDP that is part of the Determining Relationship.

Description: - A description of the relationship.

Determining Weighting: - A weighting that indicates the level of validity of a relationship.

4.7 Negative Software Development Process Relationships

Negative Software Development Process Relationships (NSDPR) can occur in two situations:

- When the existence of Software Development Processes A and B in the same Software Development Model results in the model being less efficient.

- When Software Development Process A becomes redundant because it serves a process that is specified and or done in greater detail and accuracy by software process B.

If any or both of the above mentioned conditions occur then we say the two software processes have a Negative Relationship also referred to as a Negative Software Development Process Relationship. If software process A has a Negative Relationship with software process B then it follows that B has Negative Relationship with software process A. Only one Negative Relationship can exist between any two SDPs.

Attribute of an NSDPR

Description: - A description of the relationship and why it exists.

Related Processes: - The two names of the SDPs that are negatively related.

Resolving Action: - The name of the resolving action to take in order to mitigate the effects of the negative relationship. The resolving action is performed on either of the two SDPs.

4.8 Resolving Actions

This is a set of all possible actions that can be taken to mitigate the impact of a negative relationship (NSDPR).

Attributes of Resolving Actions

Name: - A unique Resolving Action identifier.

Description: - A description of the Resolving Action to be taken.

Application: - Specifies which of the two conflicting SDPs the action should be applied to. A value of HIGH indicates the SDP with a higher SDP Weighting, LOW indicates the SDP with a lower weighting and BOTH indicates both process.

Table 8: Set of Resolving Actions

Name	Description	Application
1) Deemphasise	Deemphasise the SDP with a lower weighting	low
2) Emphasise	Emphasises the SDP with a higher weighting	high
3) Deemphasise Emphasise	Deemphasise the SDP with a lower weighting and Emphasises the SDP with a higher weighting	Both
4) Warn	Warn the user of the two confliction SDP	low
5) Skip	Show process in model but the process should not be executed, the process should be marked as SKIP PROCESS	low
6) Remove	Remove SDP from the software process model	low

4.9 Switch Software Development Model

With respect to Switch a Software Development Model (SDM) is a Software Project specific abstraction of real world software development activities. It consists of a state of SDPs which are executed in a specified sequence and were applicable the iterations on a set of SDP. Switch does not specify the number of iterations to perform. Each SDP in a SDM has a weighting which specifies the importance (emphasis) of that SDP for the given project, the SDP with the highest weighting is called the Critical Success Factor (CSF). Were iterations must be performed the weight can be used as in indicator when deciding the number of iterations to perform. It should be noted that SDP are inherently self iterating so iteration is only specified for a set of two or more processes. The Switch Framework specifies a universal graphical notation for representing SDM.

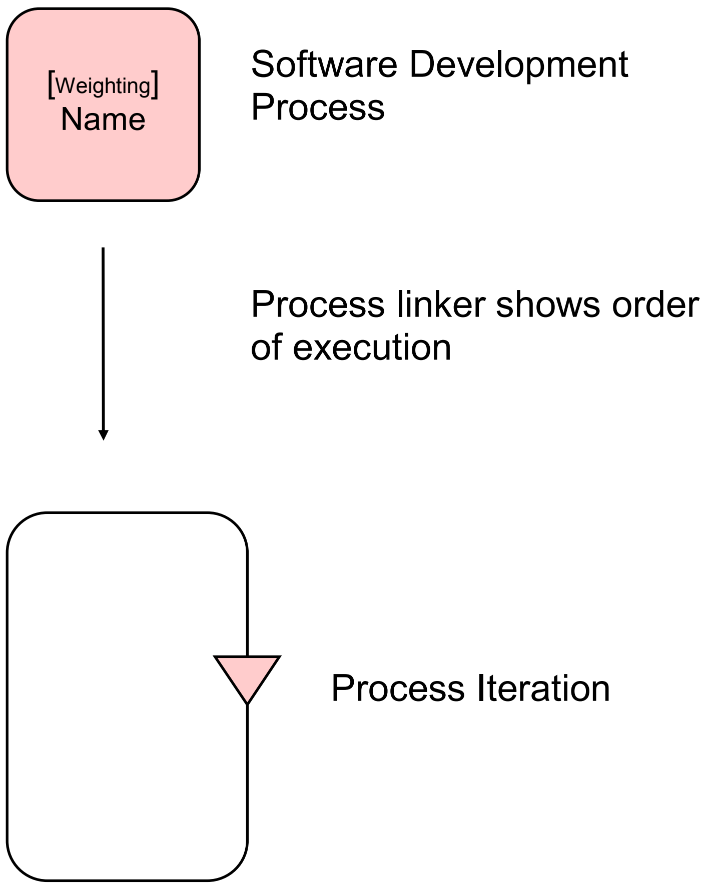

[Weighting]
Name
Software Development Process
Process linker shows order of execution
Process Iteration

Figure 13: Switch's Universal Graphical SDM Notation

4.10 Core Software Development Processes

The Switch framework is based on a set core SDP these processes (or their variations) are a part of every Switch generated Software Development Model and have no Model Determinants except for their variations. These processes also do not have weightings as they are core processes and can not be removed from a SDM, the exception to this is when they are to be replaced by child processes in which case the child processes will have been determined by MDs and will have SDP Weightings.

Table 9: Core Switch Software Development Processes

SDP Name	Description	Sequence number
Requirement Specification and Analysis	The requirements of a system are specified and analysed this stage ends when the development team is satisfied that the requirement are correct and complete.	1
System Design	The Requirements of a system are turned into a design	2
Implementation	The System Design is turned into an executable program or set of executable programs. This SDP transforms the Design into software.	3
Testing and Fixing Errors	The implemented software is tested and any arising bugs are fixed	4
Deployment	The software system is set up in the live environment for use by the end users	5

If the Switch frame work is run with on MD’s specified the SDM produced will be as the core SDM as shown below.

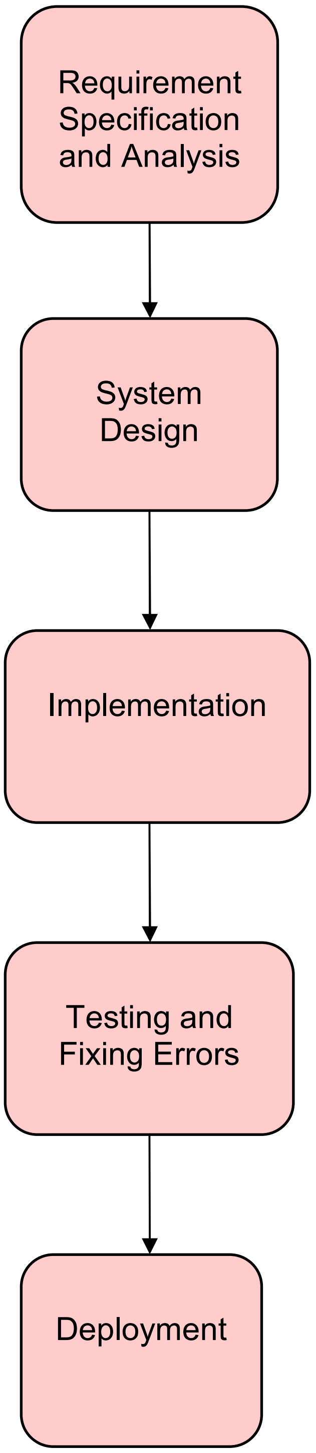

Requirement Specification and Analysis
System Design
Implementation
Testing and Fixing Errors
Deployment

Figure 14: Core Switch Software Development Model

4.11 Core Software Development Model Extension

The Core Software Development Model Extension consists of the set of MDs that apply to both Software Development Paradigm and Application Domain Extensions. Users can extend and edit the Core Extension subject to them carrying out a model extension Study.

Model Determinants

Table 10: Core Software Development Model Determinants

Name	Level Weighting Type	
Level of lack of understanding of problem	Scaled	
Reverse engineering project	Boolean	
Level of lack of understanding of current system	Scaled	
Level of lack of understanding of application	Scaled	
Risk Level	Scaled	
Level of complexity of implementing the Design	Scaled	
Level of error rate when implementing the	Scaled	
Dynamic requirements level	Scaled	
Level of incomplete requirements	Scaled	
New software product	Boolean	

Software Development Processes

Table 11: Core Software Development Processes

Name	Parent Process	Sequence Number	
Problem Definition	N/A	0	
Feasibility Study	N/A	-2	
Current System Analysis	N/A	-1	
Risk Analysis and Mitigation	N/A	1.5	
Iterate Design, Implementation	N/A	N/A	
Iterate Implementation, Testing and Fixing Errors	N/A	N/A	
Iterate SDM	N/A	N/A	
Iterate Requirements Analysis and Specification, System Design	N/A	N/A	

Table 12: Core Determining Relationships

Model Determinant	Software Development Process	Determining Weighting
Level of lack of understanding of problem definition	Problem Definition	1.0
New software product	Feasibility Study	1.0
New software product	Risk Analysis and Mitigation	1.0
Reverse engineering project	Current System Analysis	1.0
Level of lack of	Current system Analysis	1.0

understanding of current system		
Risk level	Risk Analysis and Mitigation	1.0
Risk level	Feasibility Study	0.5
Level of complexity of implementing the Design	Iterate Design and Implementation	0.5
Level of error rate when implementing the design	Iterate Implementation, Testing and Fixing errors	0.5
Level of incomplete of requirements	Risk Analysis and Mitigation	0.5
Level of incomplete of requirements	Iterate requirements Analysis and specification, System Design	0.5
Dynamic requirements level	Iterate SDM	0,1
Dynamic requirements level	Risk Analysis and Mitigation	0,3

The core extension contains no negative relationships this is to be expected as the core forms the foundation of the framework and as such should not contain conflicting process.

4.12 Model Construction

At model construction time for a given MD the user specifies two weightings Mitigation Weighting (MW) and Level Weighting (LW)

User Weighting (UW)

A UW is defined as the average of the two user specified weightings each of these weightings have a constant coefficient alpha and beta

Equation 1: User Weighting Equation

$$UW = \frac{\alpha MW + \beta LW}{2}$$

Where α and β are constants that can only be determined experimentally for a given software development environment. Their values can only be determined statistically after the model has been used extensively over a long period of time. It might be that for particular environment software projects that have a high value for MW tend to succeed more that those with a low MW. In which case the value of α should be increased.

For the switch core these are set to 1.

Process Weighting (PW)

These are construction time attribute which specifies the emphasis to place on a SDP in relation to other SDPs in the SDM.

Equation 2: Process Weighting Equation

$$PW = \frac{\lambda UW + \theta DW}{2}$$

Where λ and θ are constants that can be only be determined experimentally as explained for Equation1 above. For the Switch Core theses are set to 1

Solving Equation 1 and 2 with all constants set to 1 gives Equation 3 listed below

Equation 3: PW Weighting Equation Simplified

$$PW = \frac{1}{4}(MW + LW + 2DW)$$

It is possible that a process can be determined by more that one of the MDs supplied by the user in which the overall PW for that process is simply a summation of all the PW for each of the model determinants.

Equation 4: Overall PW Equation

$$PW_{overal} = \sum_{all} PW$$

Model Construction Algorithm

The actual process of constructing a (Software Development Model) SDM is specified as an algorithm given below.

```
Begin
Capture MDs
For each MD
{
     Look up Determining Relationships
     For each Determining Relationship
    {
        Add determined SDP to model taking into account sequence numbers
        Calculate PW for determined SDP

        Look up Negative Relationships
        For each Negative Relationship
        {
            Look up resolving actions
            Apply resolving action
        {

    }
    Calculate overall PW for each SDP
 }
End
```

4.13 Switch Framework Acid Test

With the Switch framework Specified it should be possible to dry run it to determine how the framework should be implemented as a software application. It should however be noted that this dry run uses the Switch framework without other extensions. It should be emphasised here that the framework only specifies the core Model Extension all other extensions are user enhancements that can be specific to their environment.

Model Determinants

MDs and their weightings are based on the development team and their client. Put in other words given the same project a different development team and or client the MDs for the same project can be completely different. In this case the development team consists of the author of Switch (T P Dhliwayo) and there is no client. Based on the author's experience with the development platform and extensive knowledge of the framework the MDs that were selected and their weightings are listed below.

1. Level of complexity of implementing the Design

   MW = 0, 5 (Average)

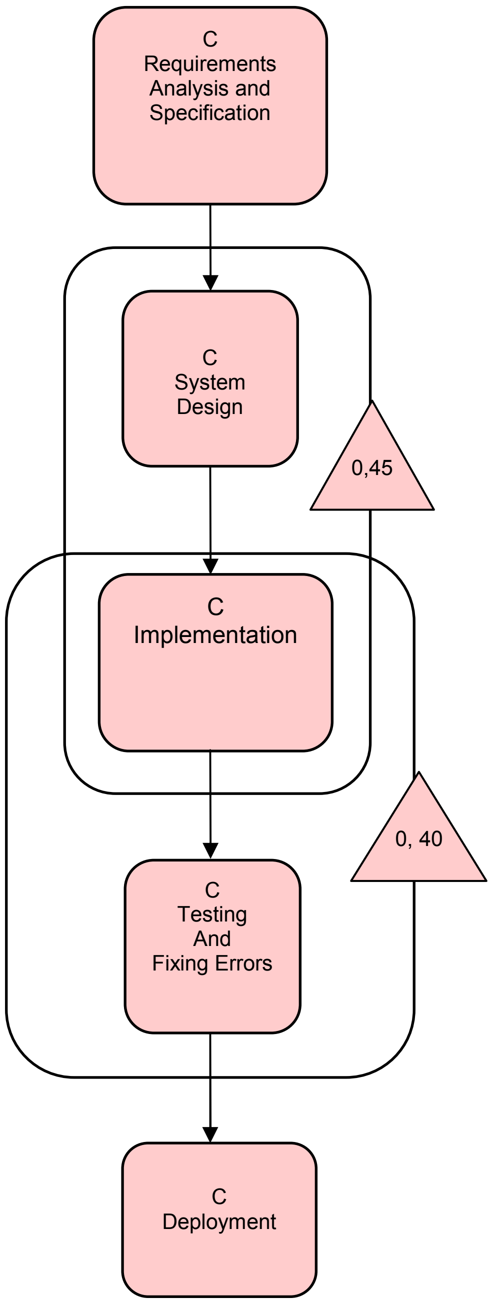
   LW = 0, 3 (Below Average)

2. Level of error rate when implementing the design

   MW = 0, 5 (Average)

LW = 0, 1 (Low)

The MDs listed above are based on the constraints surrounding the projects in particular the complexity of transforming the framework specifications into a design, then into an implementation and testing the final implementation.



4.14 Test Runs of the Switch Framework

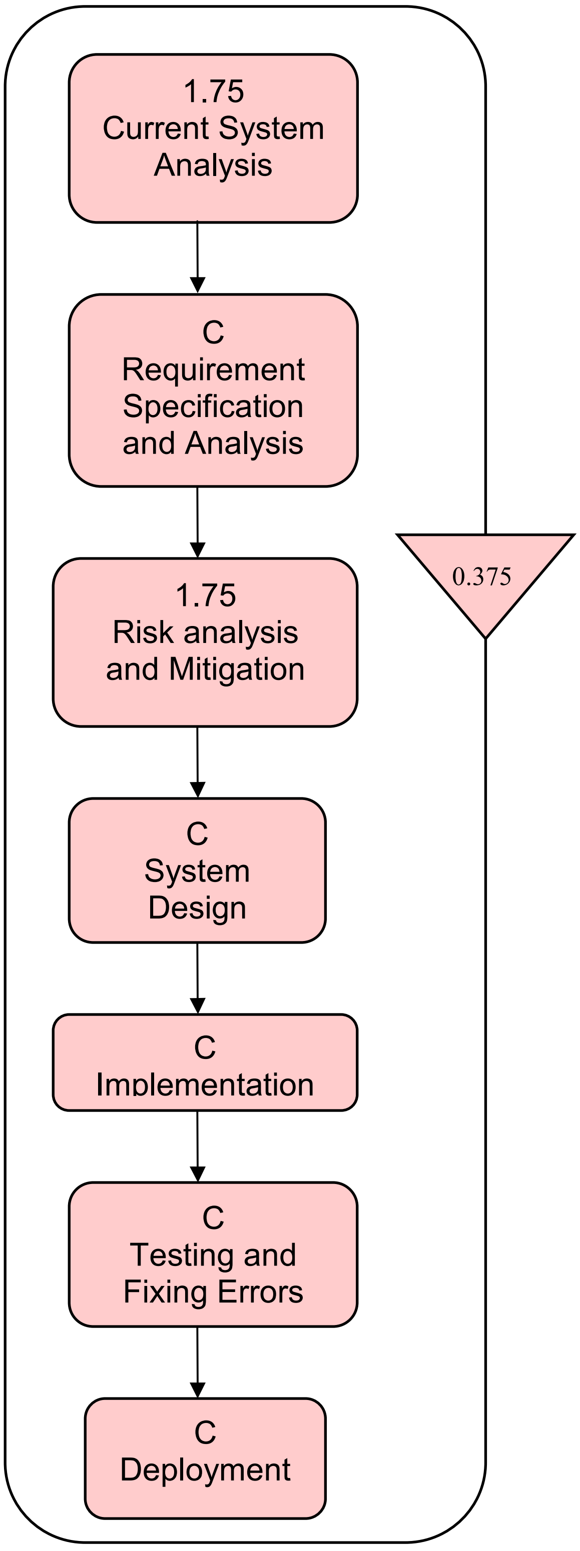
This section provides test runs of the framework so that a general comparison of model produce by Switch can be made against standard model like waterfall and spiral. The intention is not to test the framework exhaustively as this can only be done by using the framework with real Software Projects and such an endeavour can only be done within the context of another research.

Software Project 1

Problem statement: - The client requires the development of an inventory system to replace a manual system.

Development team: - The team consist of two developers who are not well versed with inventory systems

Model Determinant	**Mitigation Weighting**	**Level Weighting**
Dynamic requirements level	1	**0.3**
Risk Level	0.5	**0.5**
Level of lack of understanding of application domain	0.9	**0.9**
level of lack of understanding of current system	**0.1**	**0.1**

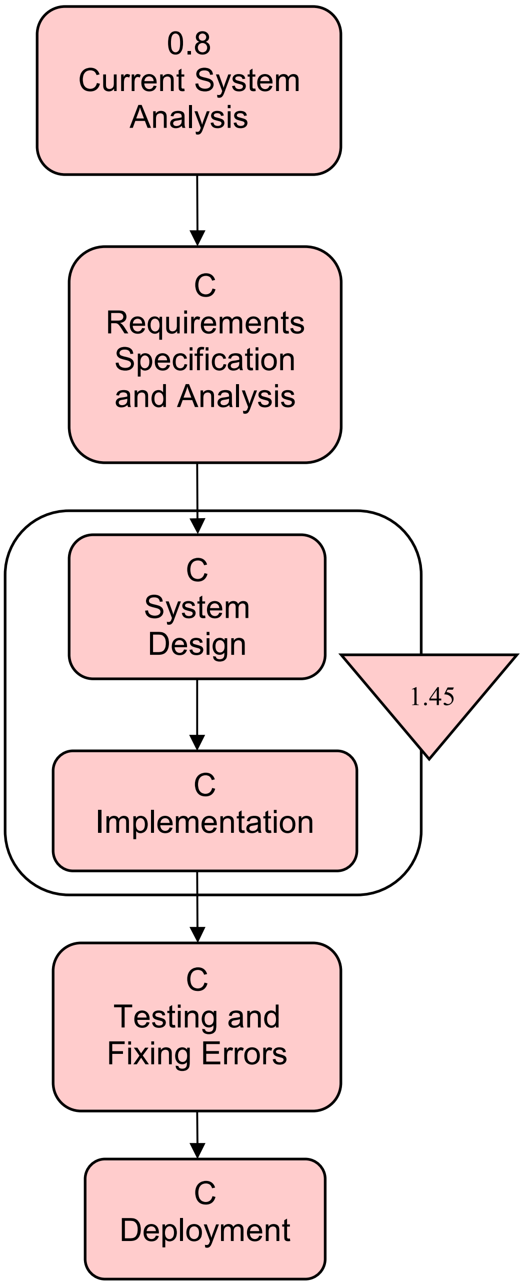

1.75
Current System Analysis
C
Requirement Specification and Analysis
1.75
Risk analysis and Mitigation
C
System Design
C
Implementation
C
Testing and Fixing Errors
C
Deployment
0.375

Software Project 2

Problem statement: - The client requires the development of a system that monitors the operations of a security guard company. For this project, there exists a database but the whole system must be reengineered.

Development team: - The team consist of developers who do not know much about the operations of a security company.

Model Determinant	**Mitigation Weighting**	**Level Weighting**
Reverse Engineering	0.6	**0.7**
Level of complexity of implementing the Design	0.8	**0.4**
level of lack of understanding of current system	**0.9**	**0.9**

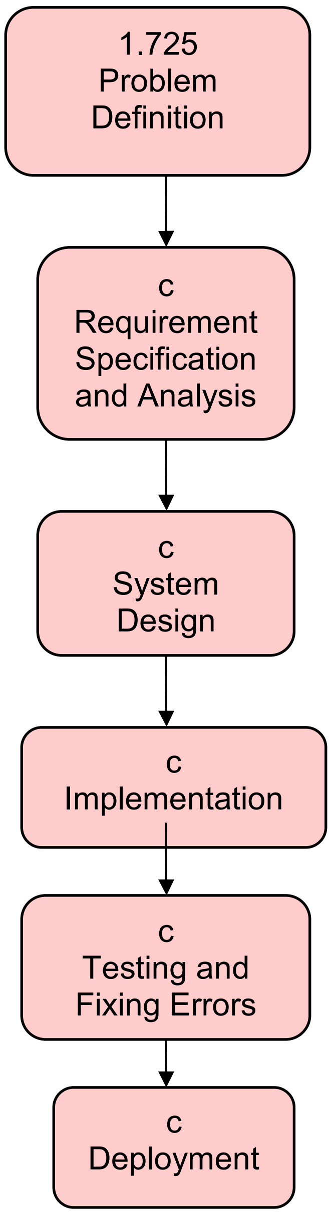


Software Project 3

Problem statement: - The client requires the development of a system that monitors the operations of a real estate agency. No similar system is available on the market.

Development team: - The team consist of developers who do not understand the requirements.

Model Determinant	**Mitigation Weighting**	**Level Weighting**
New Software project	0.6	**1.0**
level of lack of understanding of problem	**0.8**	**1.0**

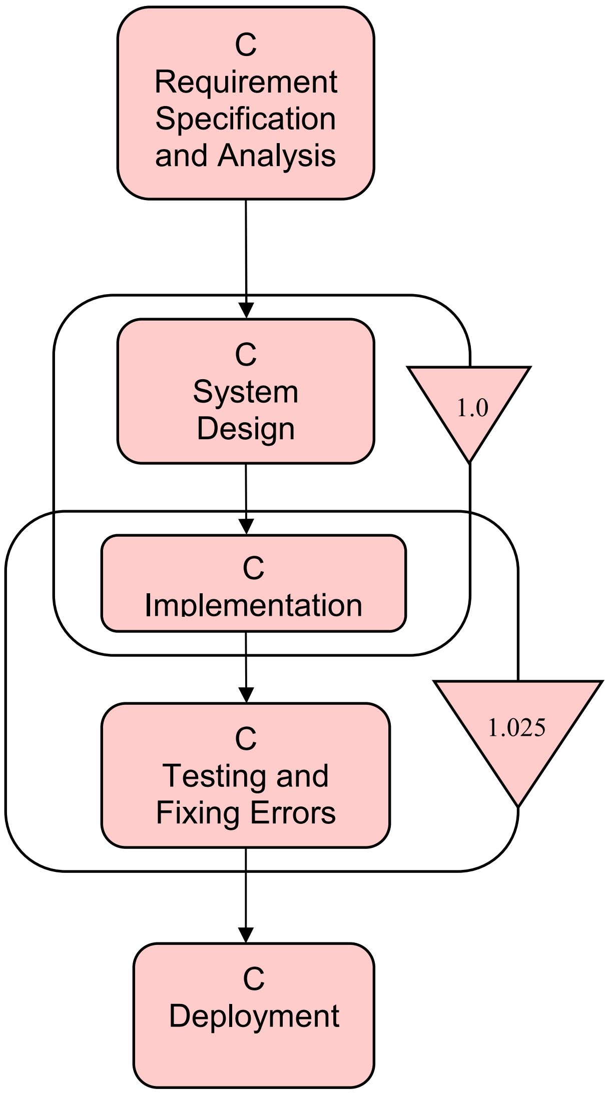


Software Project 4

Problem statement: - The developers are required to design and implement pension fund calculating software using the results of salaries from a database. The process of calculating the values involves complex mathematical manipulations of figures that would have been supplied.

Development team: - The team consist of developers who have limited experience

Model Determinant	**Mitigation Weighting**	**Level Weighting**
Level of complexity of implementing the design	0.4	**0.9**
Level of error rate when implementing the software	**0.1**	**1.0**


C
Requirement
Specification
and Analysis
C
System
Design
1.0
C
Implementation
1.025
C
Testing and
Fixing Errors
C
Deployment

5.0 Discussion

5.1 Analysis of SDM produced by Switch

- For the same software project a different development team may come up with deferent model determinants and consequently a different Software Development Model this is obviously not the case with traditional models as they do not take into account the development team. This is a better approach as the model constructed takes into account the strengths and weakness of the development team.
- The Switch framework introduces the concept of weighted processes which tell the development team the relative amount of emphasis to place on a process which helps the development process to be more focused. This feature is not available in traditional software development models.
- If the models produced by Switch are required to be more detailed this can easily be done by extending the framework which is not the case with current software process models.
- Process iterations are considered as Software Processes and as such are assigned a process weighting this again is a concept that is unique to Switch and also aids the software development team when performing iteration the relative importance of each iteration is specified.

5.2 Limitations of the Switch Framework

- Values of alpha and beta in the process weighting equations are not currently known, the starting point of a value of one is justifiable as the true values of these constants can only be acquired after extensive use of the framework further more it is possible that these values are a property of the actual development team in which case it would not make sense to specify them at a global level.

- Extending the frame work requires expert knowledge of the application domain or the software development paradigm
- Some MDs can not be early acquired especially if the model was extended incorrectly
- Should the model determinants change during the course of development it is possible to construct a new model but switching from the old model to the new model is not a defined process and the effects it will have on the final software product is unknown.

6.0 Conclusions and Recommendations

6.1 Summary

This research has succeeded in meeting its primary objectives mainly the design and specification of the Switch framework which is extensible, bespoke and dynamic. Given a software development project (as defined in the Switch framework) it is possible to ascertain it Model Determinants pass these into the Switch framework which will in turn construct a Software Development Model for the given project. The design of the framework was done using a top down approach in contrast the core extension used a bottom up approach. Although the main objectives were met the framework still needs to be extensively tested with real software development projects.

6.2 Conclusions

1. In light of evidence provided in the literature review it is clear that, in relation to software process models software projects are different and there can never be a Software Process Model that will be perfect for all projects.
2. Software Development Projects have certain attributes that can be used to determine and or construct a SDM for a given project.
3. A Software Development Model constructed for a particular software project proves to be better than a generic or standard model as it caters for the project's exact needs.
4. It is possible to construct a framework that automates the generation of project specific Software Development Models as demonstrated by the Switch framework specifications as presented in this dissertation.

6.3 Future work

In its current state the Switch framework can be used to produce Software Development Models that are arguably better than traditional models however to fully harness the benefits of the framework there is need for further research and development in the areas outlined below.

- Research should be carried out on how to effectively extend the framework as this is a non-trivial task that requires expert knowledge in the application domain of extension.
- A study should be carried out on how to accurately acquire Model determinants.
- The framework should be extended to accommodate the fact that model determinants can change during the course of software development and provide specification on how to change models during software development should the need arise.
- The Switch framework only provides the Core Software Development Extension for maximum productivities there is need for more extensions both application domain and software paradigm extensions.

References

1. **Futrell T R, F and I Shafer**. 2002 Quality Software Project Management. USA: Prentice Hall PTR
2. **Sommerville, I.** 2001 Software Engineering. 6th edition. USA: Addison-Wesley.
3. **Dijkstra's, E.** 1972 The Humble Programmer, The Communications of the ACM Dijkstra states
4. **Wikipedia.org**. 2006 Software development life cycle (SDLC), http://en.wikipedia.org/wiki/Software_development_life_cycle
5. **Institute of Electrical and Electronics Engineers**. 1990 IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries. New York
6. **Walt Scacchi**.2001 Process Models in Software Engineering. Institute for Software Research, University of California, Irvine.
7. **Hraconsulting-ltd.** 2006 Project Definition http://www.hraconsulting-ltd.co.uk/project-definition.htm
8. **Volker Gruhn.** 2002 Process-Centered Software Engineering Environments A Brief History and Future Challenges. Annals of Software Engineering 14, 363–382
9. **IEEE.** 1997 Breathing new life into the waterfall model. IEEE Software
10. **Adrian Als, Charles Greenidge & P. Walcott**, 2004 Software Process Models http://scitec.uwichill.edu.bb/cmp/online/cs22l/
11. **Gil Taran.** 2004 Why Software Projects Fail. Masters of Software Engineering Carnegie Mellon University

12. **Everette R. K.** 2002 Agile Software Development Processes A Different Approach to Software Design

13. **Agile Alliance.** 2001 Manifesto for Agile Software Development http://agilemanifesto.org

14. **Standish Group.** 2003 White paper on why software projects fail

15. **Jim Highsmith**. 2000 Advanced Software Development http://www.adaptivesd.com/index.html

16. **Beck Kent**. 1999 Extreme Programming Explained. First Edition. Addison-Wesley Publishing Co

17. **Jacobson, Ivar, Booch, Grady, Rumbaugh, James**, 1999 The Unified Process

18. **TRW Incorporated. Eckman III.C Emmet**. 2002 XP Transition Roadmap Denver Systems Operations

19. **Bradford K.**1997 Dissertation The Effects of Software Process Maturity on Software Development Efforts .University of Southern California

20. **Paulk M C Paulk. C V Weber. B Curtis. M B Chrissis**. 1995 Guidelines for Improving the Software Process. The Capability Maturity Model. Addison-Wesley

21. **W Cotterman and J Senn.** 1992 Challenges and Opportunities for Research into Systems Development, John Wiley, Chichester

22. **A L Friedman (with D.S. Cornford),** 1989.Computer Systems Development: History Organization and Implementation, John Wiley, Chichester.

23. **Frank Kand**, 1998 A Contingency Based Approach to Requirements Elicitation and Systems Development. London School of Economics .J. Systems Software.

24. **Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber**, 2000 .Capability Maturity Model for Software, Version 1.1, Software Engineering Institute

25. **H.D. Rombach, B.T. Ulery, and J. D. Valett**, 1992 Towards Full Life Cycle Control: Adding Maintenance Measurement to the SEL, Journal of Systems and Software

26. **M. Krishnan**, 1996 Cost and Quality Considerations in Software Product Management, Dissertation, Graduate School of Industrial Administration, Carnegie Mellon University,

27. **Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn W.** 1993 Bush, Key Practices of the Capability Maturity Model, Version 1.1, Software Engineering Institute,

28. **M. Durbeck and V. Hensley** 1992 Flight Dynamics Division Code 550, 552-FDD-92/001R1UD0, Flight Dynamics Software Development Environment (FD/SDE). SDE Version 4.2.0 User's Guide: Revision 1

29. **KRAUT R E & STREETER L A**, 1994: Coordination in Large Scale Software Development. Communications of the ACM

30. **Kazman, Rick and G. Abowd, L. Bass, P. Clements**, 1996 Scenario-Based Analysis of Software Architecture, IEEE Software, November

31. **Requirements Authority.** 2006 Functional and Non-Functional Requirements http://www.requirementsauthority.com/