# SYNCHRONIZE AND STABILIZE: A FRAMEWORK FOR BEST PRACTICES

by

**NALIN SATHIPARSAD**

Submitted in part fulfilment of the requirements
for the degree of

**MASTER OF SCIENCE**

in the subject

**INFORMATION SYSTEMS**

at the

**UNIVERSITY OF SOUTH AFRICA**

**SUPERVISOR: MR. TOBIAS VAN DYK**
**CO-SUPERVISOR: MRS. ALTA VAN DER MERWE**

**JANUARY 2003**

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# ACKNOWLEDGEMENTS

I would like to express my appreciation to the following people:

My supervisor Mr. Tobias van Dyk for offering insights and for his understanding, guidance and support throughout the study.

My co-supervisor Mrs. Alta van der Merwe for her suggestions and recommendations.

My wife Gudrun Maria, for her patience, understanding and support during this enduring time.

My sister Reshma Sathiparsad, for her valuable recommendations, and advice and support in the study.

The author would like to express his appreciation to the management of Quark Inc. for their support and consent to refer to the organization in this study. The reference to Quark is to indicate, by example, the practical experience of software development practices and problems of a leading software organization. Finally my gratitude to Quark for the permission granted to access all manuals, documents and data, which were used as supporting material in this work.

**ABOUT QUARK**

Founded in 1981, Quark Inc. has been on the leading edge of publishing software since 1987 when QuarkXPress™, the world's most popular layout and publication software, debuted. Today Quark maintains its industry leadership with a product line that ties together electronic and Internet publishing with information management, storage, and retrieval. From its headquarters in Denver, Colorado, Quark™ reaches around the world, providing communication software and systems to over 2 million users on six continents and in 18 languages. Employees in the United States, Europe and Asia give Quark a diverse and global perspective that is a key element of the success of the organization. QuarkXPress helped spark the revolution in desktop publishing. With avenue.quark™, Quark Digital Media System™ (QuarkDMS™), QuarkWrapture™, and other products still in development, the objective is to bring organization and efficiency to information management and publishing in the Internet age.



More information about Quark Inc. can be found on their website www.quark.com

# ABSTRACT

Software Engineering in the Small problems are neglected and are overlooked in the literature by software engineering societies and computing institutes. Small and Medium sized Organizations (SMO) are confronted with these problems and are required to revise their development strategy to overcome them. Software Engineering in the Small problems support the *synchronize and stabilize development approach* as an effective solution because the slow linear development process of the waterfall model is not suitable. The conceptual theory of Synchronize and Stabilize Approaches (SSA) is studied in depth and their variations are surveyed and thoroughly analysed. The software development advantages, disadvantages and the critical success factors of the SSA are also discussed. Three frameworks are proposed, namely, *a software engineering environment, a software process model and a framework for best* practices. In order to maintain a balance between theory and practice the author cites practical examples from his working environment.

**Key terms:**

Agile Software Process; Architecture; Best Practices; Components; Framework; Incremental Development; Process Model; Software Engineering Environment; Software Engineering in the Small; Synchronize and Stabilize

# 1 INTRODUCTION

In Fredericks P. Brooks's famous article "No Silver Bullet" (Brooks, 1987), he emphasized the difficulties associated with software development. Three years later Brad J. Cox published an article "There *Is* a Silver Bullet" (Cox, 1990). In this article he portrayed his optimism by emphasizing object oriented technology and the use of components as means to cope with the consequences of the "No Silver Bullet" syndrome. Although these respected professionals have different opinions, it can be inferred that these difficulties continue to haunt the software industry. Some general reasons for their continuing existence are listed in Table 1.0.

| REASONS FOR SOFTWARE DEVELOPMENT DIFFICULTIES |
|---|
| The constant introduction of new technology makes software development difficult and challenging. |
| The complexity of software applications is constantly growing as compared to the last decade. Architecture is more open today; as a result interfacing and integration increases a system's complexity. |
| The user community is very diverse and their increasing demand for Personal Computer (PC) software and expectations of usability and functionality are high. |
| The software industry is highly competitive and delivery time is of the essence. |

**Table 1.0:** Reasons for software development difficulties.

Apart from the abovementioned difficulties, the advancement of the Internet augments the demand for software at a global level. This was reflected in the following articles which appeared in the September 2000 Association for Computing Machinery (ACM) News:

- "China to Lead Asian Internet Growth"
- "Blair Package Backs Internet Businesses"
- "Pakistan boosts Internet Access"
- "UN: Net Growing Third World Biz"
- "Users With Disabilities Push High-Tech Limits"
- "Demand for PCs 'Continue to Expand' ".

These articles strongly emphasized the increasing global demand and importance of Information Technology (IT) in many facets of business and daily life. There is also increasing evidence that governments of the major world economies have began to recognize the global impact of the Internet. Since the challenges of the information age will continue to increase and is significant for economic growth, governments and businesses are have increased their involvement and commitment to advancement in this sector. Their involvement includes increased investment, innovation and public educational programmes in IT.

The PC and Internet software markets are diverse and Small and Medium sized Organizations (SMO) which operate in the domain are confronted with the following problems:

- They operate in a dynamic and an unpredictable environment.

- A greater degree of flexibility and adaptability is required to survive in this environment.
- They are required to employ flexible development techniques in order to enable them to bring new products faster into the market.
- They must react rapidly to consumer needs, while retaining their competitive advantage and increasing their market share.
- Their resources are limited, for example, financial and personnel resources.

These problems have an impact on their software engineering practices. To cope with them the software development strategy must be re-formulated to deliver quality products and services. Within the ambit of software engineering, the traditional waterfall development model became unsuitable some time ago because it is a linear and inflexible development process. An inherently flexible model is favoured, referred to as "Synchronize and Stabilize" (Cusumano and Yoffie, 1999; Cusumano and Selby, 1997). Basically this means that the product is developed in manageable increments. The work of the development team is continuously synchronized and periodically the system is incrementally stabilized. Because it is a sophisticated software development approach, the underlying concepts, its practical application, advantages and the disadvantages must be well understood in order to apply it. The application of the Synchronize and Stabilize Approaches (SSA) can appear to be deceivingly straightforward and simple but this is *not* the case. A misapplication of this approach would actually leave one in a worse situation than before.

The problems which confront Small and Medium sized Organizations (SMO) are referred to as the Software Engineering in the Small problems and supports the SSA as a *solution* (Moitra, 1999). A good understanding of these problems is important in order to apply the SSA as an effective solution. In the next section the Software Engineering in the Small problems are discussed.

## 1.1 The Problem Domain - Software Engineering in the Small

Small and Medium sized Organizations develop smaller, but complex systems with rapidly evolving requirements and execution environments. They are challenged through rapid changes in technology, which may result in their system being outdated as early as the first release. Although it may be an unhealthy practice, they are very often required to adopt the latest flavours in technology in order to remain competitive. Therefore they *do* need a methodology suitable for large-scale projects. However, *not* all of the methodology is required. The reason for this is that SMO also experience the same software management problems in their environments as those of large organizations (Fayad, Laitinen and Ward, 2000b). System requirements must be well managed and good software project management principles must be applied. Discipline is certainly required during the development process to overcome a resistance to change, bureaucracy and delays, as experienced in large-scale software development. Software Engineering in the Small is required to have a different viewpoint as compared to Software Engineering in the Large in order to cope with this situation. Hence the SSA was identified as an effective solution to the problems.

Software Engineering in the Small refers to SMO, which require a tailored application of Software Engineering (SE) principles and practices according to their size and type of business (Fayad, Laitinen and Ward, 2000a). The main problems with large-scale methodologies are that they are time intensive and do not scale down efficiently. SE practices for smaller organizations were ignored in the past as the focus was mainly on larger organizations. Software Engineering in the Small is a neglected area and is overlooked in the literature by software engineering societies and by computing institutes (Fayad, Laitinen and Ward, 2000b; Graham, 1989). With the rapid advancement in Web Engineering (Brusilovsky and Maybury, 2002) and Agile Software Development (Aoyama, 1998a; Aoyama, 1998b), this attitude should change in the future.

Historically mainly large organizations and defence ministries have benefited from advances in SE (Moitra, 1999). Most organizations were unsuccessful in applying SE principles and practices, the main reason being its application in an incorrect context. Unfortunately the essence of these principles is still not well understood and continues to be misapplied. Many SMO also find that the current SE principles do not integrate well into their business environment. The following are the main reasons for this:

- Operating budgets and resources are limited.
- Organizations are constantly faced with fierce competition and market uncertainty.
- A high degree of innovation is required within a short period.
- The correct mix of new features, cost, quality and reliability are speedily required.
- The business environment is volatile and global (the problem is to satisfy an increasingly diverse set of users).

Due to these operational constraints, they must adjust their SE techniques accordingly. Cusumano and Yoffie (1999) also hold similar views that is, they operate in a competitive, dynamic and unpredictable market. Therefore they clearly require an alternative software development approach to the waterfall model. For example as Internet applications are characterized as being nimble, flexible and adaptable (Lutz, 2001). The traditional SE principles and practices are too cumbersome to build such applications. Since some of these systems are large, complex and always evolutionary, a SSA is considered to be better suited for their development.

Although Software Engineering in the Small has a different viewpoint and the SSA is considered to be an effective solution, it is *not* a magic solution. As previously noted, SMO experience many software management problems which are common to large organizations (Fayad, Laitinen and Ward, 2000b). However, their solution would be tailored according to the nature of the problem, the SE principles and practices and the nature of the business. An important difference is that any decisions made to implement a solution will occur within a shorter time frame for the smaller organizations.

The *core* Software Engineering in the Small problems to be addressed by this research is as follows:

- Non-bureaucratic and flexible approaches in order to rapidly develop software in a disciplined manner.

- An increased attention on the Software Engineering in the Small problems is needed by the software industry as these are often ignored.
- Small and Medium sized Organizations operate under difficult conditions (as mentioned previously); and they require simpler SE principles, which is in tune with their business environment.
- Small and Medium sized Organizations must be able to manage their development problems effectively and efficiently under difficult operating conditions.

In the rapidly growing Information Technology (IT) industry, continued research in software engineering techniques and development methodologies is of paramount importance.

## 1.2 Value of the Research

The specific value of the research is a contribution in partially solving some of the major Software Engineering in the Small problems and to focus attention on a "neglected area" of software development. It is envisaged that the study should provide organizations with additional information to effectively manage their problems under the stressful conditions in which they operate.

In addition, it is intended that the research's contents may stimulate debate in this problem domain among information systems students. As a result further research may develop, especially in software engineering environments and software process models, which are aligned to SSA.

## 1.3 The Purpose of the Research

The purpose of the research is to propose a means to address the core set of Software Engineering in the Small problems as discussed. The study aims to provide balanced information about SSA that will be useful to support organizations that may decide to apply it. The author believes that through the correct application of the "Synchronize and Stabilize" concepts, it is possible to accelerate software development in a disciplined manner without sacrificing quality. The study further strives to demonstrate that an appropriate software engineering environment and software process model can be tailored to an organisation's requirements, thus facilitating the efficiency of the software development process.

The objectives of the study are to:

- Consolidate an understanding of the underlying theory and concepts of SSA in a non-superficial manner.
- Study the existing SSA and provide a critical analysis.
- Propose a Software Engineering Environment (based on the critical analysis) that is suitable for SSA. The author is of the opinion that current approaches seem to neglect this area.
- Propose a Software Process Model that is apt for the SSA. Current approaches also appear to neglect this aspect.
- Steer away from best practice hype by proposing a framework for best practices to support the

application of SSA. The author's observation is that this aspect is also neglected with respect to SSA.

- Demonstrate the conceptual theory of SSA through practical examples from the author's working environment.

## 1.4 The Research Methods

The following research methods will be applied:

- A detailed, in-depth analysis of the literature has been undertaken. Literature published by various authors on how they view and address the Software Engineering in the Small problems has been reviewed. The intention is to analyse the current solutions suggested and provide an evaluative report.
- Observation based approaches will also be used. They are based on the author's working environment (Quark Deutschland GmbH) and include overall development processes, project management, quality management and system development strategy.

### 1.4.1 Literature Review

Primary literature sources on the various SSA are:

- Selected text books
- Digital libraries such as the Institute of Electricians and Electronic Engineers (IEEE) and the ACM
- Publications of academic institutions and research organizations
- Diverse computing journals
- Internet search engines and Web directories.

The literature sources accessed and retrieved have been analysed and forms the basis for the literature review section of the study.

### 1.4.2 Observation

Observations made by the author on the application of synchronize and stabilize in his current working environment were cited as practical examples in this study. More information from this source will be in the form of documents, tables and diagrams.

## 1.5 Data Analysis

Since this is not a statistical study, the data from the literature review will be analysed manually. The results will be presented textually and where appropriate tables will be used.

## 1.6 Limitations of the Research

Only a selected subset – namely those that apply directly to the research goals - of existing SSA from the literature will be analysed. The author is of the opinion that it is adequate to meet the research's

defined goals and objectives.

With reference to best practices, only a framework is proposed. The analysis of specific best practices is beyond the scope of the dissertation.

Although Quark will be cited as a practical example during the discussion, neither a case study of Quark is intended, nor does it suggest an implementation of a best practice framework.

## 1.7 Chapter Structure of the Dissertation

The dissertation is divided into the following chapters:
Chapter 1 discusses the problem domain in detail and explains the value of the research. The intended goals and objectives to be attained are listed. The research's methods and limitations are outlined.

In chapter 2 literature on the SSA is reviewed. The key objective is to thoroughly understand the commonality and differences among them. A further aim is to identify trends and gaps in the literature and make recommendations. Observations made by the author in his working environment with respect to topics presented in the literature are also discussed in this chapter. The review will be the basis for the discussion in chapter 3.

Based on the literature review, a critical analysis of the synchronize and stabilize approach is presented in chapter 3. The methods are compared by a set of criteria and then discussed. Through an enriched analysis of the SSA that is supported by real world experience, it is possible to exploit the advantages of the SSA and implement a means to overcome its shortcomings. This critical analysis will be the basis to propose the set of frameworks, which is the topic of chapter 4.

Three frameworks are developed in chapter 4. The intention is to address Software Engineering in the Small problems namely, a software engineering environment, a software process model and a framework for best practices.

Chapter five summarizes the problems encountered, goals achieved and stated objectives. The main findings and areas where further research may be focused are discussed.

## 1.8 Summary

This chapter introduced the study's subject matter and identified the core Software Engineering in the Small problems to be addressed. The specific value of the study was outlined and the objectives to be achieved were stated. The research methods that will be used and the constraints imposed in this study were also discussed. The structure of the discussion and the topic of the subsequent chapters are summarized in section 1.7.

# 2 SYNCHRONIZE AND STABILIZE – A Literature Review

## 2.1 Introduction

In this chapter literature on the Software Engineering in the Small problems is reviewed. The review is necessary to gain insight into the dissertation's subject matter. It is important to include it because many experiments and implementations on the subject have already been conducted previously and it avoids repeating them. It will also guide the direction of the research and lay the foundation for the analysis of problems and solutions which were previously researched. These would include identifying trends, gaps, understanding what works and what does not, and proposing solutions to address the Software Engineering in the Small problems. Therefore in this respect, the review is focused on acquiring more knowledge about the current problems and exploring ways to solve them.

In addition to the literature review, the author's observation on overall software development processes, project management and product strategy in his working environment is an important part of the research. The observations relate to the practical implications and aim to provide a balanced discussion between theory and practice.

To ease understanding and provide clarity, the discussion is presented under the following headings:

- The Conceptual Theory of Synchronize and Stabilize
- The Concepts of the Incremental Development Model
- Synchronize and Stabilize – Implementation variations
- Observations.

Although the discussion is presented under the separate headings, it should be considered holistically because of the close relationship between the aspects covered.

## 2.2 The Conceptual Theory of Synchronize and Stabilize

The SSA is more complicated as the name may suggest. Therefore it is imperative to understand the concepts and the philosophy that it is based on. This understanding helps to gain a clearer insight of the implementation of the various SSA in the literature, which are discussed later. Another advantage is that knowledge is gained on how to proceed with this approach in systems development.

At this point it is appropriate to consider Brooks's statement (Brooks, 1987), "*I believe the hard part of building software to be the specification, design, and the testing of the conceptual construct, not the labour of representing it and testing the fidelity of the representation.*" Despite the wide acceptance of the object oriented paradigm and advancement in technology, the reality of this statement lies in the fact that the software industry continues to experience these problems. Despite the waterfall model's (Royce, 1970) weaknesses it is mature. It has gained a wide acceptance and a great degree of success in practice to develop large systems. It has also helped to alleviate the difficulties that Brooks had expressed. However, this process is linear and too elaborate for Small and Medium size

Organizations (SMO). These organizations cannot afford the long waiting time caused by an extensive development cycle to bring a product to the market. Therefore an alternative development approach is required.

The SSA is an alternative development strategy to shorten a long development cycle and to accelerate the time to market. The concepts are based on a disciplined management of uncertainties, exploration, and the implementation of a divide and conquer process in software development.
The *uncertainties* experienced are influenced by several factors such as:

- Unknown complexity.
- Uncertainty in the requirements and specifications.
- What will the market acceptance be?
- What is a competitor developing in same problem domain?
- What is the payoff?
- Risks, sudden push / pull in technology.
- How much time to ship the product?
- Financial bottlenecks.

The list is not exhaustive. The nature of problems confronted by small Internet organizations that are trying to develop a product should be clear. Brooks's statement appears to be relevant to both large and small software development projects.

With regard to a *divide and conquer* strategy, the objective is to manage complexity and hedge against *uncertainties*. With this technique the system is divided into manageable increments which are prioritised and developed. The duration of the development of an increment is about six to eight weeks, however, there is no universal agreement on it.

The impact of *divide and conquer* propagates *exploration and cumulative learning* about the system during each increment. Knowledge which is accumulated during development may be applied to improve software management and deliver a high quality system. *Exploration and cumulative learning* also provides the basis for an organization to create a knowledge base. This knowledge may be applied to identify uncertainties and plan a defence against them. In principle the SSA is an incremental software development approach, which spawns many desirable side effects and are beneficial to alleviate the problems stated above. Organizations such as Quark, Microsoft, Netscape and Mozilla apply the SSA.

In order not to sound misleading, the development phases of the SSA are similar to those of the traditional waterfall model. They are as follows:

- Requirements Specification
- System Design
- Detail Design
- Implementation and Test

- Integration and Test.

However, there is an important difference to observe, namely, that each increment of the *synchronize and stabilize* development cycle is an iteration of the all phases of the waterfall model. Thus the SSA "inherits" its development phases from the waterfall model. Through the iteration of short development cycles, software development should be more manageable and accelerated. Both these approaches are used where appropriate in the industry and each has their merits. This is best explained by an illustration:

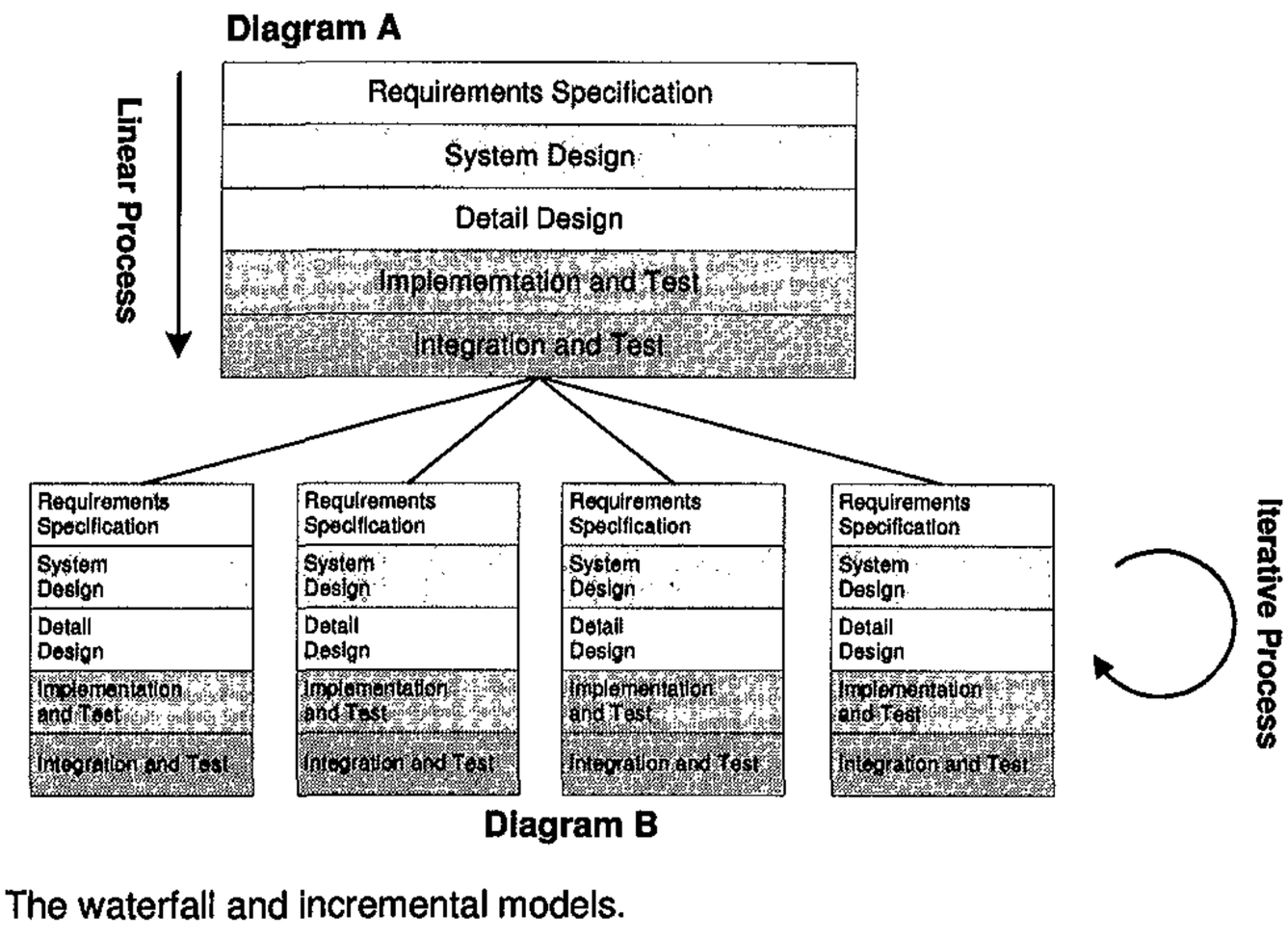


**Figure 2.0:** The waterfall and incremental models.

In Figure 2.0, diagram A illustrates the monolithic sequence of the traditional waterfall development model. With this development approach the specifications are frozen. The implementation of the target system is followed by a large testing and integration cycle. Diagram B illustrates the breakdown of the monolithic waterfall model into smaller self-contained increments to be implemented. Another way to view this is that at the end of the development, the target system will be an aggregation of a number of planned finite increments. In theory, assuming everything being equal, a system developed following the approach in Diagram B should aggregate to diagram A. An important practice at Quark is that the Quality Assurance (QA) department scrutinizes each phase of an increment. The objective is to ensure high quality throughout the development cycle.

The concept of an increment is simple to understand and should be clear. The question is *what* must be *synchronized* and *stabilized* ? An indispensable control and co-ordination mechanism of the SSA is the daily build (Cusumano and Selby, 1995; Cusumano and Yoffie, 1999; McConnell, 1996). The work of the developers is checked in daily at some stipulated time, a build is created and a series of smoke tests are performed (McConnell, 1996). A smoke test is a test script or test procedures that tests basic system operations. Smoke testing maybe manual or automated. An important objective of the smoke

test is to check that each increment of code to the system harmonizes with the existing code. If the build breaks, then the presumption is that the code that was last added is the cause. In this manner the work of the development team is continuously *synchronized* and periodically the system is incrementally *stabilized*. Stabilization will depend on the quality of code added during synchronization. The stabilization phase deals with software issues such as performance, system stress, system resource consumption and fine-tuning. If the stabilization phase is not well managed then the potential danger is that the system will degrade. The system should execute without any crashes and produce consistent results, which are repeatable and reproducible. The techniques of creating a daily build and smoke testing are practised by many organizations such as Quark, Microsoft, Netscape and Mozilla.

As a result, the work is performed in parallel and in small phases. For example, the specification, coding, testing and integration activities are done in parallel as compared to the waterfall models sequential execution of tasks. The SSA aims for speed, however, without jeopardizing usability and quality. As an advanced developed technique it requires excellent co-ordination skills and increases the responsibilities on the entire development team. It can be stressful if it is not well managed. Although the SSA evolved from a hacker development style, (Cusumano and Selby, 1995; Cusumano and Yoffie, 1999), it is no longer a pure hacker approach because good software engineering disciplines are being applied. Since there is a need to launch a product on the market fast, schedules are tight and good Software Engineering (SE) principles are essential. Slipped schedules are costly.

Depending on the severity of problems, the level of programming faults encountered and system performance, additional builds may be created for system testing. During a critical situation (the system cannot converge) three builds were created daily at Quark. This meant that the builds were smoke tested frequently than usual in order to detect and eliminate faults. In McConnell (2000), a controversial argument for and against the daily build is presented in relationship to incremental development. In the author's opinion and experience the daily build is the "heart beat" of the system being developed and also agrees with Dave Card's views (McConnell, 2000) that the build is a "brute force" system integration.

During the initial increment of Quark's E-Commerce application, system integration problems were experienced despite the creation of daily builds and performance of smoke tests. The causes of the problems were that the developers tested only their own sub systems and neglected to test across the system. The strategy to resolve this problem fast was to set up a test laboratory that was managed by a process-testing manager. Integration test cases were created and the process flows of the application were tested and documented. Several builds were created daily in order to accelerate the progress. In extreme cases when nothing functioned, check-in stop mode resulted; no new code is allowed to be checked-in until the system is fixed. All efforts are expended on fixing faults in order to bring the system to an accepted level of stability. This would appear to be "brute force" integration, but it was necessary. "Brute force" integration is the persistence to ensure that the system functions *before* proceeding with the next increment. Although there were delays, the laboratory testing method had reduced the impact of faults because its formal approach. According to Cusumano and Yoffie

(1999) the theory is to give the developers more autonomy in their work. However, Quark's experience in this specific regard (testing) reflected that the developers required supervision. This is a classical situation that is commonly experienced; namely, developers do not test adequately because they tend to focus only on their own code.

By managing complexity through the SSA, system integration problems were easily resolved. The test laboratory was set up without a lengthy bureaucratic process. The lightweight formal testing approach was well accepted and was not considered to be tedious. At the end of each integrated test session the team discussed their progress. Therefore the application of simple SE principles blended well with the development environment and Quark was able to manage its problems effectively.

Another observation made at Quark was that the duration of an increment is usually longer than six to eight weeks. The duration may be of any length from three to twelve months. For example, QuarkXPress 5.0 was released in December 2001; however, the planned release date for version 6.0 is December 2002, which is one year later. An increment may be viewed as a minor release and a version as a major release. The usability and added value of a major release influences the system's competitiveness. During the development of Quark's E-Commerce application, some incremental cycles were seven weeks long. There is no hard and fast rule because much depends on the scope of the release. The planned feature set for a release also determines the schedule. The general intention is to produce a feature rich release that would encourage users to upgrade. That is, the users want to see significant new functionality in the product, which are useful in solving their business problems. Hence, the six to eight weeks is too short for a significant release. On the contrary, it is well suited for an intermediate release (or patches or service packs) with reported faults being corrected.

Strohm (1991) quoted in Rautenberg (1992) identified four types of software, which are important factors for software development. They are as follows:

- Type A: Specific application for internal use of an organization
- Type B: Specific application for an external user
- Type C: Standard application for an external user
- Type D: Standard application for an anonymous user.

The applications, which are developed by Internet software organizations, are mostly of Type D and are release driven. That is each release is an increment and these organizations may benefit from the SSA. In-house development is of types A, B and C. Although a strong focus has been on rapid software development by SMO, an opinion that SSA is solely restricted to them should be avoided. Organizations which fall into types A, B and C can also apply the SSA for software development. Generally the choice of a process model is subject to the problem statement. Because Internet software organizations are faced with the Software Engineering in the Small problems SSA has become a viable solution for them.

It can be concluded that the incremental model forms the key pillar on which the SSA is based while it

has its roots in the waterfall model. In the next section the incremental development model is discussed in order to better understand the various applications of it in the literature.

## 2.3 The Concepts of the Incremental Development Model

The development of the *incremental model* emerged from the practical realization that software is engineered step by step (Schach, 1993). The model pivots around this development philosophy and the development process is also evolutionary. As noted in Figure 2.0, a system is divided into a finite number of incremental units. These are developed and tested in a series of incremental builds and the process stops when the target system satisfies all requirements (Cusumano and Yoffie, 1999; Russ and McGreggor, 2000). Thus the development process is bottom up, which iterates the waterfall lifecycle (Dunn and Ullman, 1994). This important point should be remembered at all times. Therefore in reality the development principles and habits practised by using the waterfall model should not be considered to be obsolete and useless. It is these traditional principles and habits that are innovatively applied by the SSA to address Software Engineering in the Small problems.

Graham (1989) provides a detailed discussion on incremental development and proposes several incremental life-cycle models. The following problems of the monolithic lifecycle, as in the waterfall model were identified:

- Software resource (labour, material, personnel) estimations are weak.
- All requirements cannot be accurately and completely defined.
- Reaction to changes is long and costly.
- System testing and integration is difficult.
- Software maintenance is costly.

He dismisses the view that the incremental model can result in sub-optimal systems development and strongly favours the incremental software process model. The above problems are also those which small software organizations experience today. A positive side effect of incremental development is evolutionary delivery because an increment may be delivered to customers as a useful version of the system.

Ahituv and Neumann (1990) refer to it as an evolutionary approach and distinguish it from an ad hoc approach, because the system planning process (short, medium and long) is vital, whereas an ad hoc approach is arbitrary. They identify two forms of evolution, one the comprehensive information system evolves with changing needs, and two a specific subsystem evolves over time in response to changing operational needs and to the change of the comprehensive information system. Aoyama (1998a) refers to the above changes as major and minor enhancements respectively. In practice it is common that some sub-systems incur more changes than others.

As with other development models, a clear statement of the system's intentions must be provided to establish the scope and effort (Powers, Cheney and Crow, 1990). This is similar to a product vision

statement that is produced to define the product goal (Cusumano and Yoffie, 1999). Such a statement is used to steer the requirements definition process. In practice it is difficult to define the complete scope and the objectives comprehensively in the beginning because the problem statement is often not clearly understood. Through the concept of iteration and layering during development the problem statement becomes more precise iteratively as a deeper level of understanding is reached.

Rational Software [W1] discusses the application of incremental software development in favour of the traditional waterfall approach in detail. They refer to their incremental approach as the Rational Unified Process (RUP). Their discussion is interesting since it emerges from industrial experience instead of an academic environment. In the discussion Kruchten [W1] states "it makes sense to try to see how the two apparently *conflicting lifecycle models* might pair up..." However, what exactly is the conflict is not explained. The author does not agree that these are "*conflicting lifecycle models*", but is of opinion that they are *different* life cycles or software process models. A close examination shows that RUP inherits its phases from the waterfall model. The steps and iteration of an increment are actually "small doses" of the waterfall model and RUP does exactly that. It has its own terminology for each phase which in the author's opinion is confusing and adds little value.

The preceding discussion emphasized on the concepts of the incremental development model. Motoyoshi and Otsuki (1998) exploit the concept of incremental development further by proposing an Incremental Development Project Planning Approach (IDPA). With the IDPA method, the technology to be used is assessed, decomposed and scheduled according to technical dependency at the time of introduction. That is, the project is planned in increments and technology is also introduced incrementally. The model is aimed at embracing changes in the development environment in order to increase its acceptability, improve communication and improve productivity of the development team. Therefore the focus is strongly on people and it is suggested personnel must satisfy some perquisites before they are required to learn new technical issues. The IDPA method provides a strategy to adapt to technological changes in the development environment by explicitly incorporating them into the project-planning task. Since organizations developing in Internet time are faced with rapid changes in technology the ideas suggested by the IDPA method may be beneficial.

Yoshioka, Suzuki and Katayama (1998) propose a method called Incremental Software Development method that is based on Data Reification (ISDR). It focuses on writing programs incrementally. The concept is based on the premise that a program can be constructed before its specification is completed. Their incremental development process is as follows:

- Reifying the data that is used. That is, abstract data is defined in concrete terms during each increment.
- Refining the program according to the data reified.
- Executing the program using abstract interpretation.

With this technique a program is stepwise refined as the specification becomes clearer. As a result it can be executed in its intermediate and incomplete form with the advantage of early error detection.

This concept is consistent with the fact that since system requirements are not always complete, the program specifications are also unlikely to be clear and well understood. This programming style is flexible and adaptable to changes. A flexible software development style is necessary to cope with Software Engineering in the Small problems.

From the above discussion it can be concluded that the concept of the incremental model does not necessarily apply only to iterating the waterfall model. It was further exploited and applied in project management and programming, which are equally important in software development. The development of the Agile Methods, Aoyama (1998a, 1998b) and Extreme Programming, Beck (1999), is an indication that incremental techniques are favoured to overcome the restrictions of the waterfall model.

Over time many variations of the SSA have emerged to overcome the restrictions of a linear development process. These variations have adopted the incremental model as their core development process and are discussed in detail in the next section.

## 2.4 Synchronize and Stabilize – Implementation Variations

In this section literature on the implementation of the various SSA is reviewed. The underlying concepts and principles of the approaches are examined with respect to the problems they address within the scope of Software Engineering in the Small and the solutions proposed.
The following variations are discussed:

- Incremental Development
- Evolutionary Project management
- Extreme Programming
- The Scrum Software Development Process
- Product Line Concepts
- Improvisation Techniques
- Whitewater Interactive System Development with Object Models (WISDOM)
- An Integrated Approach
- Agile Software Process (ASP)
- Synchronize and Stabilize using Components.

The abbreviations that are used to refer to the SSA during the discussion are listed in Table 2.0.

| ABBREVIATIONS | EXPLANATION |
|---|---|
| **ICM** | Incremental Model |
| **EMP** | Evolutionary Project Management |
| **XP** | Extreme Programming |
| **SCR** | Scrum Software development Process |
| **PLE** | Product Line Engineering |
| **IMP** | Improvisation Techniques |
| **WSM** | Whitewater Interactive System Development with Object Models( WISDOM) |
| **INP** | Integrated Approach |
| **ASP** | Agile Software Process |
| **SSC** | Synchronization and Stabilization using Components |

**Table 2.0:** The SSA abbreviations and their explanations.

## 2.4.1 Incremental Development

Graham (1989) provides a detailed discussion on the application of the incremental model and presents his framework for an incremental life-cycle model. Generally he concentrates on the incremental process and prototyping in incremental development. The benefits of incremental development are viewed in relationship to a typical monolithic life-cycle model such as the waterfall model and a shift from it is proposed. The reason for this is that the waterfall model is inflexible and cannot adapt easily to changes during the development cycle. However, changes are easier to incorporate with the incremental model.

He clearly distinguishes between the concepts of incremental development and incremental delivery. In this paper an increment is defined as "a self-contained unit of software with all supporting material such as requirements and design documentation, test plans, cases and results and user manuals and training". The definition is easy to understand and consistent with Software Research's differentiation between programming and software development [W2]. The distinction is defined as "*the difference between programming and software development is that the latter deliver documented, tested and maintainable software. The first just delivers a program.*" This distinction is important for SMO because the former impacts on the quality and usability of their deliverables and influences the value they offer to customers. The implication of incremental development is that user manuals and documentation are also created by increments. Therefore an increment must deliver value to the user.

Incremental development in this paper is presented as what may be referred to as its "early standard form". That is, the "*development of a system in a series of increments throughout the project time scale*". The incremental life cycle model presented here is identical to the first three phases of the waterfall model, namely, requirements specification, system design and detailed design. The difference lies in the code test and integration test phases, which are performed incrementally. It is in the last two phases that the system is developed by a series of build and test increments. Prototyping and formalism are also proposed in conjunction with the incremental model. However, Graham does not consider the effect of project management and an appropriate software engineering environment for his incremental development model.

According to Graham incremental development can be used with or without incremental delivery. Incremental delivery is defined as "*The delivery of increments to the customer or user at intervals throughout the project timescale.*" He favours evolutionary delivery (which includes incremental delivery and incremental development) because early customer feedback received is useful in learning and understanding the system. This feedback is also useful input for future increments.

## 2.4.2 Evolutionary Project Management

Gilb (1988) presents a detailed discussion on the Evolutionary Project Management (EPM) model. He makes an important distinction between Incremental Development and EPM. He views the Incremental Development model as follows, "...in the 'Incremental Development' model multiple build-and-test steps are based on detailed requirements and design specifications. These are more or less 'frozen'. The point is gradual delivery of partial results. Incremental project management models might be *forced* to revise their requirements and design, but they do not *intend* to, and it is not a regular part of the process".

According to Gilb the above differs with EPM in the following respect, "In the Evolutionary Development model, less effort is put into the initial overall system level requirements and design specification, initially. These specifications are, however, continually updated as step experience dictates. At each step, *detailed* requirements and design for that step are specified. " As a result the following activities may result:

- new experience gained to date
- new user needs are defined
- new technology may be required
- economical influences experienced and new market insights.

The emphasis is on learning rapidly and applying the lessons for improved customer satisfaction and ability of the developer to manage the project. Since SMO must manage their problems effectively under difficult conditions, EPM appears to be appropriate because it helps to guide the development in a disciplined manner. Furthermore EPM may be used in various types of projects [W5].

Graham (1989) also acknowledged that the implementation of each increment might result in changes, for example, in the requirements, design and code. In practice whether they are really "frozen" and not part of the "intended process" is a development philosophy. This is further influenced by an organization's priorities as to what would add value to the software and how its competitive advantage will be affected. The importance is what must be incremented and to what extent. These are some of the issues confronting SMO. Very often the situation on hand influences the decision making process. A closer study of the Incremental Model and Evolutionary Project Management indicates that they are based on the basic concept of incrementing. The important difference is what must be incremented and to what extent.

In a case study titled "Evolutionary Project Management" Woodward (1999) describes the adoption of EPM to deliver software rapidly. Based on the fact that a software project is rarely completed, but evolves, he found that by using EPM such evolution could be better managed. This was achieved by shorter development cycles (with logically related functionality considered) being implemented iteratively in small increments. According to Woodward, the basic idea was to provide frequent deliveries (six to eight weeks duration) of the product so that it could be installed as a self-contained entity. This view is similar to the definition of an increment stated by Graham (1989). In their discussion on the SSA, the development period is also between six to eight weeks (Cusumano and Yoffie, 1999; Cusumano and Selby, 1997). Therefore with EPM the delivery of a cycle is not the final product, but an existing cycle with increased functionality. That is each evolution builds upon the previous ones. EPM also emphasizes on the importance of regular feedback from users which helps to identify problems to be solved. Gilb also refers to this method of learning about the system and consequent adaptation as the key idea of EPM.

In Woodward's project, Order Management and Routing (OMAR) strong emphasis was placed on the software being traceable and this was regarded to be an important contributor to quality software. That is, the systems must be traceable to the requirements and functional specification. However, Royce (2000) views this as a "principle that didn't make the cut". The author is of the opinion that software traceability is important for organizations confronted with the Software Engineering in the Small problems because it affects future software releases and its quality.

The development teams and the testers worked in close co-operation on the OMAR project. They were also given a fair degree of autonomy, such as deciding on the initial features to develop and objectives of each development cycle. This indicates that a good team relationship is a contributing success factor. The daily build was used as a co-ordination and control mechanism during the development. With EPM it was also possible to test the complete system frequently, which had a positive impact on the quality of the system. The implication is that if anything went wrong then the addition of an increment is most likely to be the cause.

EPM was considered superior because by providing an estimate for each cycle within the global project plan, a continuous learning process occurred. As a result each time the estimates tended to be more realistic. The major achievements which Woodward experienced were that the project was completed ahead of schedule and the system included additional functionality than initially planned. This is important for SMO because time, speed and quality are of essence for them.

### 2.4.3 Extreme Programming

Extreme Programming (XP) an agile method (Boehm, 2002) is currently a highly debated topic and a full discussion on it is beyond the scope of the discussion. The first principle of the agile manifesto states that, *"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."* This principle implies that speed and quality are important and development priorities must be well defined. This section examines the application of XP as a technique to produce

quality software fast. Beck (1999), the founder of XP, also regards the sequential waterfall model as being too rigid and its application is time consuming to produce a product fast for the market. His reason was that the planning, analysis and design results produced by the waterfall model delved too much into the future. He is also of the opinion that the problem domain is often not clear in the beginning and changes later are costly. His proposal is to perform these activities "a little at a time" throughout the development, which is the basic principle of XP. XP also adopts a radical philosophy, that is "do what is required for now".

XP leans towards producing a simple design and delivering the system in small releases. The features to be implemented are referred to as *stories*. For each development cycle the customer decides on the scope and releases and selects which *stories* must be implemented. The customer is free to pick the *stories*. Once selected, they are transformed into subtasks and implemented by the programmer. The aim is to implement the subtasks within a few days. During an iteration new stories which are tested and ready for delivery are put into production. With XP a system may be put into production within a few months, but without solving the complete problem domain. It is clear that XP also adopts EPM concepts and the system will grow in functionality over time. A new release cycle will vary from daily to monthly. Although XP is a radical and fast paced development approach, the involvement of users is advantageous. Since they decide on the scope, the acceptability of a release is increased. Furthermore, informal discussions also occur with the users and provide valuable information for future system requirements. This is useful in designing an open architecture system because it helps the designer to think ahead.

Speed is an important factor in XP and new code may be integrated with an existing system within few hours. During integration the build is created from the beginning. If all tests are passed, then the build is accepted, else the changes are discarded and the build is rejected. The build also serves as a control mechanism to get the system to converge after each increment. In other words the programmer tasks are frequently *synchronized* and the system being *stabilized* after each new *story*.

The most controversial aspect of XP is that the programmers collectively own the system and two people (pair programming) write a program. This programming technique is considered to improve productivity and code quality. The technique of re-factoring also makes it possible to improve code quality and to reuse existing code. The team is free to set their working rules and can change them at anytime. However, everybody must sign up to the rules. This implies that the team has a fair degree of autonomy and is committed.

### 2.4.4 The Scrum Software Development Process

Scrum an agile lightweight development process, is based on defined and black box process management. It is a methodology for the management, enhancement and maintenance of an existing system and for new systems development. It also assumes that the analysis, design and development processes are unpredictable and has a control mechanism to manage the unpredictability and risk. Scrum considers that this unpredictability, risk and complexity is operating on the edge of chaos, and

the process is aimed at to avoid falling into chaos [W3]. Scrum (a sports term) is used as a metaphor to indicate the co-operation of the team to reach a single common goal.

Rising and Janoff (2000) describes their experience in implementing the Scrum software development process. They also found that the traditional approach was too rigid and that requirements are not always known at the beginning and faster reaction to changes was required. Therefore Scrum was applied to cope with requirement changes during the development life cycle in order to meet changes in business demands. Similar to EPM they also emphasized on learning about the system during the development cycle. Scrum is also an incremental software development process. The fact that it allows development to occur close to the edge of chaos implies that it has a common element with XP. Organizations using Scrum were also found to experience an increase in productivity.

A Scrum team consists of 8 to10 people. Scrum is not recommended if the development requires large complex team structures, because it is easier for smaller teams to work independently and effectively. Project planning is considered an important phase during which the initial system architecture is developed in conjunction with the project's vision. This vision would influence the addition of future increments and how to *synchronize and stabilize* them over time. However, it is acknowledged that the architecture can change during subsequent increments. Therefore it is important that the system's architecture is designed for extensibility. The ability to react fast to these changes is challenging for SMO because it impacts on the entire product.

After project planning short development phases called *sprints* are identified and are developed incrementally. Since changes are inevitable during each increment planning is quick. Therefore Scrum strongly focuses on accelerating the development process; as a result tangible system is visible faster. This acts as a motivational factor for the team. In the Scrum process, the marketing group or the customer will determine the release date. In order to establish a base product the features with the highest value or which are important to the customer are given high priority. Thus the customer's input is important during this process.

*Sprints* are subdivided into tasks with the aim to be completed within less than a week. When a *sprint* is completed it is merged with a previous version of the software. Each *sprint*, once incremented produces a useable version of the software and this makes it possible to deliver the product at anytime. In other words, a deliverable version of the software is always available. Hence delivery is phased and any feedback (from customers and marketplace) is considered. After each *sprint* a decision may be made whether the project should be continued or stopped. The key objective of a *sprint* is to deliver valuable functionality fast.

A *sprint* is actually a *time-box development* procedure which is regarded as a best practice in software development and is discussed in detail by McConnell (1996). The delivery date of a time-box chunk is about 90 days. According to Avison and Fitzgerald (1995) it is argued that a system cannot be divided into 90 days chunks and the idea should be discarded. A *sprint* cycle lasts between one to four weeks

and an entire project only a few months. For this approach to work a radical change in the development culture is necessary and a breakaway from strict formalized methodologies is required. In order to meet the delivery date of a *"sprint time-box"*, functionality may be omitted from the development schedule if the user agrees. However, the delivery date does *not* change. In this manner Scrum is aggressive in preventing missed schedules. Once the Scrum closure phase is reached the development is completed.

A *Scrum Master* is appointed to lead the team and acts as the control instrument during the development. He controls and tracks the team's progress and all procedures at the meeting. Although control is necessary, the procedure in Scrum may have a negative effect because people may feel intimidated that they are being closely watched. The meetings are short and are aimed to identify problems and co-ordinate the development. Issues such as backlogs, progress, problems and the setting of priorities are discussed. Project risks are also addressed and decisions in this respect are taken. The Scrum process appears to be non-bureaucratic and flexible to develop software rapidly in a disciplined manner as required by SMO.

### 2.4.5 Product Line Concepts

Knauber, Muthig, Schmid and Widen (2000) also found that small organizations must be flexible and quick in reacting to customers needs and cannot operate like large organizations by taking a long view. They propose the Product Line Engineering (PLE) concepts which aims at maximizing reuse of core ideas. For example, common system functionality is developed and used across multiple diversified systems. Thus saving time and distributing the development costs across systems. These aspects are important for SMO because they can stretch their limited financial resources effectively and speed up development. The core idea of PLE is to synchronize and organize the development of multiple products based on a single reusable infrastructure. It is also imperative that there is a clear vision of which products will be developed in the future. This suggests that management must have a long-term plan of proposed systems. Therefore the system architecture must be open and extensible. A careful study shows that PLE slants towards the modern development philosophy of using components and a black box development approach. The common system functionality can be developed as components or black boxes and integrated with other products of the product family.

The method used is called Pulse and was developed at the Fraunhofer Institute for Experimental Software Engineering in Germany. The method is customisable to support conception, construction, usage and evolution of product lines. The Pulse method commences with the requirements definition and is more comprehensive than that of XP and Scrum. The reason may be that the requirements influence individual products and are categorized as to what is common to all the products and what is product specific. Therefore in the long-term it is recommended that more time be allocated during the requirements definition phase. A commonality analysis is performed to filter out common requirements among the products, which are proposed for development. Apart from analysing what is common, the differences are also analysed. The entire procedure is informal and the structure is open. Those involved can decide which structure to use. From the literature it can be inferred that the aim is to

create generic solutions as much as possible and treat exceptions accordingly. Another inference is that several systems are developed in parallel; as a result the development process is accelerated.

The product line core infrastructure is the reference architecture that is common to all systems in the product line and ensures conceptual integrity. The benefits are high reuse, increased quality, reduced cost and decreased time-to-market. Pulse uses also incremental development to develop the reference architecture. The system is then built iteratively by adding increments at a time. System refinement takes place as the development proceeds and the architecture is extended as required. The development requires a great degree of flexibility and product line development flexibility is difficult to conceptualise as compared to individual systems.

Since the complete requirements are not known at the beginning, the teams also worked closely with their customers. This approach enabled them to react faster once they are known. The customers also provided useful feedback for the implementation of subsequent increments. It was observed that SMO have limited resources, lack proper documentation, employ no systematic development process and focus on producing code. Through the Pulse method they succeeded in introducing discipline in the overall development process.

Macala, Stuckey and Gross (1996), discusses similar experiences with the product line development approach. They also emphasized that a solid architecture is a foundation for successful product line development. Product line development requires skilled software engineers and this must not be compromised. A new way of thinking is also necessary. Although product line development aims to deliver quality software rapidly, good project management, a mature software development process and strategic planning are essential.

### 2.4.6 Improvisation Techniques

Dyba (2000) suggests improvisation techniques for small software organizations. He proposes that organizations should endeavour to find a balance between refining the use of existing skills and experimenting on new ideas with a goal to improve on old ones. That is improvement is through exploitation and exploration. A concentration on exploitation will result in an improved use of old technologies, while concentration on exploration will cause an organization not to realize the advantages of its discoveries. This is the key concept of improvisation as an instrument for improvements in small software organizations. He also proposes a shift from a linear development model as small software environments are non-deterministic and constant negotiations and re-negotiations of the system prevail. His emphasis was on viewing the development environment realistically because it is unpredictable. That is there are many unknowns, it changes constantly and it is difficult to control.

Improvisation is an improvement approach to understand the relationships between action and learning in small software organizations. It helps to deal with the unseen and through continuous experiments new possibilities and innovation may be inspired to improve matters. As a result small

organizations may survive the increasing turbulence and complexities in their environments by deploying their limited resources optimally. For example, old software modules may be re-examined and exploited, and reworked with new features being added each time. In such a situation the principles of code re-factoring would be an appropriate technique [W4]. Poorly written code has always been a hindrance and time consuming. Re-factoring can contribute significantly to improve this. Therefore instead of rewriting anything from the beginning, the suggestion is to use what is already available by fully exploiting it. This is a viable and economical proposition when resources are limited. A side effect of exploitation is a gain in development speed; however, no time frame is mentioned. Improvisation also requires the development teams to be creative. The extent of their creativity will depend on their interpretation of the environment and choice from alternatives.

There are no detail development plans in the beginning. Furthermore the results of the work cannot be specified early in the project, but only a projection of the planned product can be produced. This situation is common for most small software development organizations. However, the product usability will evolve over time, and the time to market the product will determine its competitiveness. Dyba warns that successful improvisation is difficult, because the concepts of exploitation and exploration must be learnt and additional work is required in this field. The intention was to demonstrate that through improvisation organizations could cope with changes and unpredictable environments, which are hurdles in Software Engineering in the Small.

### 2.4.7 Whitewater Interactive System Development with Object Models (Wisdom)

Wisdom is a lightweight software engineering method that was developed by Nunes and Cunha, (2000). The method has three components, namely, a software process, conceptual modelling and project management. It has its own modelling notation, which is a subset of the Unified Modelling Language (UML). It is aimed at the needs of small teams that develop interactive systems. It also addresses the needs of small organizations, which must be flexible and react fast to changes. Furthermore small organizations have limited resources and cannot afford expensive high-end tools. Wisdom addresses this problem with low cost tools and improves the development process from a chaotic way to a methodical one. These objectives are common to Wisdom and Pulse.

During development the goal and evaluation cycles are based on the requirements, which are exploratory processes conducted in multiple short sessions. The deliverables from this process is a requirements workflow, with the purpose to develop a system that satisfies the user. The requirements workflow is the product's high-level concept, user profile, task analysis, business model and requirements model. The task flows are free from technology and the implementation details. They describe the user's intentions and the functionalities which are required to accomplish specific tasks. Since Wisdom focuses on interactive applications, the interactive scenarios are defined by Use Cases to guide the development.

An analysis workflow is then created to refine the structure of the requirements model. The results produced are the requirements description, system internal interface definition and the User Interface (UI) definitions. UI development accounts for about half the development effort of interactive systems. During analysis the necessary classes are also identified to represent an abstraction of the system's requirements. Therefore knowledge of object oriented software development techniques is necessary. An interaction model is also created to represent the interaction between the system and the actors in order to indicate relationships and navigational paths.

The design workflow drives the implementation and refining of the system and architecture. During the design workflow an incremental refinement of the analysis and interaction models is performed. Any non-functional requirements for the system are also considered. The work is of a technical nature and is focused around the development environment with respect to the programming language, database and operating system.

Wisdom emphasizes strongly on system's modelling. A series of workflows and models which flow in a logical refinement sequence are created. One of Wisdom's philosophies is to design for reuse which can contribute to rapid development in the long term. The development speed of Wisdom results from avoiding chaos (improvement in the development process), unnecessary re-work (saving time) and using simple tools (low cost and reduced complexity). Since the tools are simple, high training costs may be avoided.

### 2.4.8 An Integrated Approach

The problems of incomplete requirements and rapid changes in the problem domain are a hindrance to systems development. Russ and McGregor (2000) propose an integrated development process with an aim to develop high quality software and timely results economically. Selected portions of several standard process models were adapted to create a development process for small projects. Their software development process integrates the following principles:

- Iterative Development
- The Incremental Model
- Guided Inspection (quality control)
- Measurement Framework.

The developers are guided during each project step and as to what emphasis to place on the activities in each development phase. To save time and improve efficiency, some of the development processes are logically re-arranged (or integrated). For example, guided inspection is integrated into the development activities to eliminate the need for formal design reviews. The team structure is clearly defined and the roles, tasks and responsibilities of the members are clarified at the beginning of the project. The roles guide the interaction throughout the process cycle. Thus a clear communication channel is established early in the project to avoid miscommunication. Some examples of roles are, manager, domain expert, tester, programmer and customer. If special skills are required then they are

also acquired.

The Integrated Approach is methodical, well structured and tends to lay a sound foundation on which the development must take place. Similar to Improvisation Techniques it encourages the principle of reuse. In this regard existing software development phases (as listed below) are used optimally. This is advantageous for small organizations, because it is easier to adjust the changes in an environment as compared to for example, that of XP. The discipline that SMO seek in order to develop software fast is available at an "affordable price" with the Integrated Approach.

The development phases which Russ and McGregor (2000) proposes comprises are as follows:

- Define the Requirements Scope
- Domain Analysis
- Application Analysis
- Architecture
- Application Design
- Program Implementation
- Integration
- System Test.

These phases are similar to those of the waterfall model. The development is incremental and iterative, thus speeding up the development process. An increment can be completed between one to two months. Despite the speed factor their approach also aims to introduce more discipline and control into the development process. This is similar to the Wisdom and Pulse approaches which were discussed in the previous sections. It is recommended that if the team is small, then the increment should be small. This is common sense because a team's capacity is limited. Based on the feedback from customers and experience from previous iterations, the requirements and design models are modified because the understanding of the problem becomes clearer.

### 2.4.9 Agile Software Process (ASP)

The concept of agility was initially associated with the production of hardware and with low importance to software development. In this section literature on the Agile Software Process (ASP) model is reviewed. Aoyama (1998a) proposes and discusses the ASP and its experience in large-scale software development. He also recognizes that the demand for software is increasing rapidly and that the Internet has changed software development's priority from *what* to *when* (Aoyama, 1998b). Furthermore due to the high operating costs in their home country, many organizations have offshore development centres. In reality software development in a global development environment is difficult to manage. He also indicates that there are communication problems which results in project delays. He proposes the ASP model as a solution to manage faster software development and communication difficulties over distributed environments. The ASP model is complex and a software engineering environment is recommended for a problem-free operation.

The aim of the ASP model is to deliver software rapidly, but at the same time to be adaptable to changes in software requirements and delivery schedules. It applies the concept of incremental development and incremental delivery over a specified period of the project. With ASP the system is broken into a number of lightweight modular processes to ease management. Similar to Scrum, a time-box approach is used where the delivery times are fixed. As a result a large monolithic application is developed incrementally and concurrently and is shipped in multiple releases. With this approach the slow development progress that is experienced with the traditional waterfall model is alleviated. In comparison to the waterfall process model the requirements are not frozen. The impact of a change in the requirements is viewed as a change in product, process and environment and emphasizes on the ability to react rapidly and adapt to changes.

ASP's agility stems from its macro process architecture. As explained in Figure 2.0, each incremental unit is based on the waterfall model and ASP recognizes this in terms of its architecture, which is incremental and iterative. In ASP the unit development process was redesigned to make concurrent development possible. As a result software development can evolve fast, adapt to changes and cope with strict delivery schedules. Multiple processes can be started over multiple releases to speed up the development. This makes it possible to develop multiple features in a fixed cycle time. An interesting concept of ASP is that it distinguishes between major and minor releases. Based to the effort required time is allocated appropriately. For example, a major release is allocated six months and a minor release is allocated three months. It is possible to deliver a release every three months and Aoyama has reported a reduction in development time by 75%, that is, from one year to three months. Such reduction would appear to be drastic and aggressive.

ASP is divided into two parts, namely, an upper-stream process in which analysis and implementation are performed, and a lower-stream process where integration testing and system testing are performed. The output of the upper-stream process once completed is checked in and integrated into the whole system as a release. The intended release then goes through the lower-stream process of integration testing and system testing. This *is* the *synchronize and stabilize* part of the system.

According to Aoyama, the ASP model has the following major contributions, and are summarized as follows:

- It is a new process model which is time based.
- It promotes evolutionary delivery.
- The ASP model accommodates an architecture that integrates concurrent and asynchronous processes. It is an incremental and iterative process and supports distributed multi-site development. ASP is people oriented.
- It provides recommendations for a process centred software engineering environment.
- Valuable experience in the use of ASP in large-scale systems for over a period of five years is now made available to the software industry.

ASP is an effective solution to the Software Engineering in the Small problems that small

organizations may adopt. It recognizes the difficult conditions in which organizations operate and addresses problems experienced in a global software development. As a solution it is based on sound SE and management principles. Aoyama also proposes an Agile Software Engineering Environment (ASEE) that is appropriate to ASP. Organizations may benefit from this proposal as system planning is strongly emphasized. ASP is complex, non-bureaucratic and flexible. However, due to its complexity there is a learning curve to follow. It is clear that a new way of thinking is required on software development and an organization must embrace change if it wishes to implement ASP.

### 2.4.10 Synchronize and Stabilize using Components

Software development with components is a comprehensive subject on which a complete book can be written. As a result a detailed discussion on it is beyond the scope of this discussion. Since development with components can be an option to overcome the Software Engineering in the Small problems, it is discussed. The basic concept is to accelerate the development process by using third party software components to build a complete product or subsystems of it. Another advantage of using components is to reduce cost and improve quality. However, an organization may develop its own components which may be used across the spectrum of products it develops (Voas, 1998). For example, a font selector component may be used in a word processor and a spreadsheet application. A well known example is Microsoft's Words and Excel applications. These are different products which use a common fonts selector component.

An application may be created solely with components or the base application is developed and specialised third party components are integrated with it. The former case is an ambitious option, while the latter is common practice during development. The objective is to reduce the development time and cost by using software (components) that is already available. Since components are tried and tested, they are generally considered to function well and improve the reliability of a system. Therefore all that is required is to manage the integration and interfacing of components. For example, components which calculate sales tax and perform customer address verification are specialised components which are used in many commercial applications.

Repenning et. al.(2001) provides a detailed discussion on using components as highly inter-connectable software building blocks. Theoretically by using components the complexity of a system maybe reduced, because the development team does not have to care about the internals functioning of them. In other words certain aspects of the system has already being completed by a third party. All that is required is to develop a clean interface with it. Component usage also fosters reusability and modular design. They also discuss their development process, called Component Oriented Rapid Development (CORD) to develop software with components. With the CORD approach activities are performed in parallel by distributed teams working on sets of components. The components are small and as a result of the distributed nature of CORD the development activities are small.

The use of components can accelerate software development and provide a competitive edge. Components can also help to scale down the development process and help organizations to cope

with its limitation on human resources. It is interesting to note that components may be used in conjunction with the SSA already discussed, for example, XP, PLE and ASP. The motivation behind the SSA is to accelerate software development, however, in theory the use of components can *accelerate the synchronize and stabilize approaches.*

Griss and Pour (2001) discuss the use of components to reduce development time and cost. According to them component based software has the potential to do the following:

- Reduce development cost and time to market.
- Increase reliability of the system, since components are reusable and testing and review are stringent.
- Improve maintainability.
- Enhance quality.

Software development with components will introduce new activities. For example, a component selection and evaluation phase. These phases could be sometimes with risks because the right choices are vital to the overall product. Therefore if components are used then the development life cycle is extended with new activities. For example, the analysis phase may be extended with the component selection and evaluation phase. During this phase it must be guaranteed that the selected components can satisfy the requirements, else gaps will occur in the overall functionality of the system. There are other considerations as well, such as:

- Technological Compatibility
- Localization
- Technical Support
- Ease of Integration.

Although the development costs may be reduced, organizations will encounter other costs associated with components. If components are complex, then a specialist may be required for customization, integration and support. In some cases royalties and licence fees must be paid when a component is included in the system sold. Therefore a balance must be found and a proper cost analysis should be made whether to make or buy.

The author has made many observations in his working environment which relate to some of the topics discussed above. These observations aid in understanding the practical implications of software development problems and are discussed in the next section.

## 2.5 Observations

Quark, a leading software product developer is also confronted with the Software Engineering in the Small problems, which were discussed in the previous section. The discussion of the observations made is intended to enrich the discussion of the literature review. In this section focus is on real problems as encountered in the real world.

The following observations are examined:

- Communication
- Incomplete Requirements
- Documentation
- Functionality Grouping
- User Feedback and Reviews
- Product Line Development
- Teamwork
- Components
- Development Standards.

## 2.5.1 Communication

Aoyama (1998b) has noted that software development in a global development environment is difficult to manage and communication problems result in project delays. Quark also has development centres globally (America, Germany, India and Singapore), in order to keep its development costs low. The organization also experiences difficulties to co-ordinate the work of developers at the Singaporean and Indian sites due to the following reasons:

- Weak communication skills.
- Poor language proficiency.
- Technical proficiency varies due to level of education.
- Poor infrastructure (especially in India). For example, unreliable telecommunication services.
- Ambiguity in software specifications.
- Time differences between countries.
- Cultural habits which influence working habits.

The most important aspect is communication. In order to gain clarity during system development, communication is oral and written. For example, a conference call discussion will be summarized and distributed so that all participants are of the same mind to the subject matter. In the case of strategic decisions, a visit to the respective country may be necessary. Recently videoconferences were successfully conducted which helped to reduce the anonymity in communication. Sometimes it is difficult to co-ordinate the participants due to the difference in time. Hence delays result. Generally the teams cope and the regular submission of status reports is the project's control mechanism. Quark aims to employ high calibre people with good technical skills. Each candidate is thoroughly screened and must pass a technical test.

## 2.5.2 Incomplete Requirements

The problems of incomplete, ambiguous and unclear system's requirements were also experienced at Quark. Although they require time to resolve them the development cannot be delayed. An interesting observation is the manner in which the European and Indian developers handled this situation. The

European developers tend to wait until all required information is available. They experienced difficulties in dealing with inconsistencies, because in contrast to the Indian developers, the perfect situation is desired.

The Indian developers commenced development with whatever information (within reason) was available. For each inconsistency that was encountered, they created a stub or inserted dummy code, which is by-passed during execution. Once the situation has been clarified the stubs are systematically replaced with the correct code. With this development style they are faster in producing visual results. This programming approach is similar to the ISDR method, (Yoshioka, Suzuki and Katayama,1998). The author has had an informal discussion with the developers about this subject and their response was that in India they are confronted with inconsistencies in their daily life and perhaps this helps them to cope. It would appear that environmental characteristics and cultural habits have an impact on their working habits.

### 2.5.3 Documentation

Graham (1989) emphasized that the implication of incremental development is that user manuals and documentation are also created in increments. At Quark the documentation structure or the framework of its contents is extracted from the code's comments. Therefore the code must always be properly commented and updated. Sometimes this is difficult to control. However, the method is a time saver because it provides the technical writers a good framework to start with. When a release is shipped the latest documentation accompanies the software. Any updates to it are published on their home page.

### 2.5.4 Functionality Grouping

As with all the approaches discussed, shorter development cycles are implemented with the features being logically grouped. The logical grouping of features is not simple because it affects the entire system and the development schedule. To decide on the feature implementation set at Quark is not trivial and conflicts arise because of the dependencies among them. For example, product management may request certain features, however, they cannot be implemented because they depend on other features, which were planned for a later date.

The same situation arises when features must be omitted in order to meet the schedule. Although the user may agree to omit specific features, the number of features omitted may increase because of system dependencies. The biggest problem is that these dependencies are not obvious to the users and at the end they feel that the release has no added value. At Quark, through discussions and reviews of Use Case documents by independent groups, most dependencies become known and implementation is planned with minimal conflict.

### 2.5.5 User Feedback and Reviews

User feedback is an important software refinement instrument. Most of the SSA discussed place high

value on this information source. Customer views, their concerns and changing requirements are also taken seriously at Quark. This was a driving force that has contributed to QuarkXPress being successful.

As a standard practice Quark conducts periodic design reviews and the users are formally requested to attend. The objectives are to identify problems and strive for excellence. During these sessions the users get a chance to see visible development progress and clarify any misunderstandings. A typical outcome is that a number of usability issues are uncovered. For example, the user interfaces are confusing, human computer dialogues are tedious and specification deviations. Technical reviews are conducted separately to address issues like program design and performance. Since Quark has offshore development sites reviews are also conducted there.

### 2.5.6 Product Line Development

As Quark specialises in publishing software, the common functionality among its publishing applications are exploited. The product Digital Media System (DMS), a content management is integrated with its E-Commerce application. This simplifies the management of digital assets (graphic images, documents, sound files), which are used in an electronic catalogue that is displayed on the Internet in an E-Commerce application. This can be viewed as software reusability and software component development. In this manner Quark can offer a smart business solution fast. DMS is also sold as a standalone product. For example, a well known South African newspaper organization uses DMS as a standalone content management system. Although from an architectural point of view the publishing products share common software modules, the E-Commerce application's architecture is technically different. This tends to complicate the integration between two separate product lines, namely publishing and E-Commerce. In this case reuse is limited across product lines.

### 2.5.7 Teamwork

As with most software projects delays are common and Quark is no exception. Based on the fact that some parts of system have a lower workload than others, developers are assigned to assist the overloaded teams when necessary. This is not a problem because most developers have worked across the system. They also share their expertise during problem solving. This is important because it is necessary that all members understand the system holistically. The strategy is to spread the knowledge of the system and cope when resources are limited. The above average performance of the developers is recognized and they are generously rewarded with a spot bonus.

To improve the quality of the developer's knowledge and skills, SE literature is purchased and is accessible available to everyone. The developers also have an unlimited Internet access and use it extensively to research problems. The Intranet is well structured and is used as knowledge base to store SE documents, white papers and online books.

## 2.5.8 Components

In some application domains Quark does not have in-house expertise. Therefore in order to speed up development and use specialised knowledge, third party components are integrated into a system. For example, in the E-Commerce application, a third party component is integrated to compute American sales tax. Recently a third party warehouse component replaced its in-house developed warehouse system. The reason was that the external warehouse component offered additional functionality and it would be time consuming to develop them in-house.

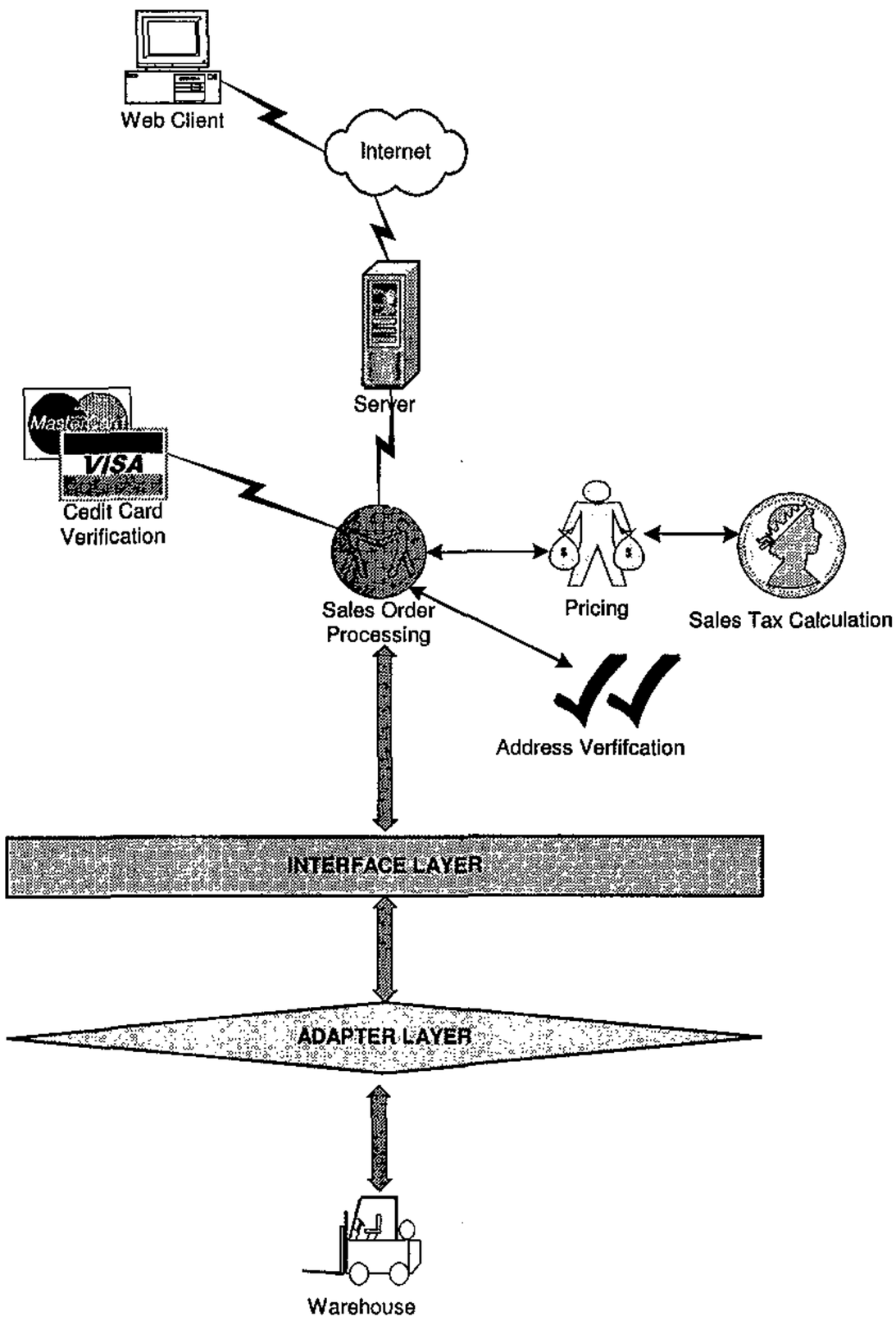


**Figure 2.1:** An E-Commerce application and component integration.
*Source*: Adapted from: Quark's Business Architecture Document.

In Figure 2.1 a high-level view of Quark's integrated E-Commerce application is illustrated. Full details of database access and other sub-systems, such as financial accounting and purchasing are omitted, as they are not relevant for the discussion. The objective is to illustrate system development with

components. The Sales Order Processing (SOP) system component is the main communicator with a customer over the Internet. The figure shows the SOP receiving an order through the Internet. The customer's credit card is verified by the services of an external credit card processor. The address is verified by the Address Verification System component (AVS), while the price calculation system component calls a third party component to calculate sales tax. The figure also shows the interface between SOP and the third party warehouse component.

Quark is quick to seize a business opportunity as it develops components for its own systems which may be sold. The Address Verification System (AVS) that is integrated with the E-Commerce application was developed internally with the intention to be sold. Since the AVS is owned, the licence costs are avoided and that helps to reduce the software cost. The interface layer is generic and is designed to exchange data with almost any warehouse system. The adapter layer is system specific and transfers the data from the interface layer to the warehouse component. The theoretical concept of the adapter is that a customer can use the E-Commerce application independent of the warehouse system that is installed. All that is required is a warehouse specific adapter to interface with the E-Commerce application.

### 2.5.9 Development Standards

Quark has its own SE development standards. They are defined as simple as possible in order not to be tedious to follow. There is little bureaucracy and the standards state the minimal compliance that is required. For example, major requirement changes must go through a change control procedure. The procedure states the minimum amount of information that is required to ensure a successful change control. Minor changes are documented and implemented. What may be regarded as a major or a minor change is discretionary

## 2.6 Conclusion

*Synchronize and Stabilize* efficiently and innovatively exploits the concepts of incremental development. It inherits its core development process from the waterfall model and it indicates that old techniques, when applied innovatively can bring about change. Although this is not Dyba's (2000) improvisation technique, it would indicate the important relevance of his discussion, namely to explore. Hence, out of an old idea, new ideas are created. As a result several methods have been found to address the needs of SMO to develop quality software rapidly. The software development problems confronting small organizations are gradually being identified and addressed. Many SSA have been developed to propose ways to manage these problems - software development using components and agile methods are the most recent approaches. Each approach deals with a different software problem domain and has its own strategy to resolve it. They also exhibit their own organizational properties and while solving common problems.

### 2.6.1 Organizational Effects of the SSA

In Table 2.1 the effects of the approaches on an organization are compared. Although some tools and

technique were specifically identified, it is presumed that additional development tools and techniques are also applicable. PLE claims that no special tools are necessary, while Aoyama admits that ASP is complex and a Software Engineering Environment (SEE) is necessary to avoid problems. The complexity levels of the SSA are generally high while some are medium.

| Property | ICM | EPM | XP | SCR | PLE | IMT | WSM | INP | ASP | SSC |
|---|---|---|---|---|---|---|---|---|---|---|
| **Application domain example.** | ERP | ERP | Payroll | switch-ing system | Desktop pub. | Diverse domains | Booking system | Mainly small Projects | Tele-Commu -nication | Diverse domains |
| **Complexity level.** | Medium -high | High | High | High | Medium -high | Low | Medium | Medium | High | High |
| **Learning curve.** | Medium | Medium -high | High | High | Medium | Medium | Medium | Low-Medium | High | High |
| **Skills level required.** | Medium | High | High | High | Medium -high | High | Medium | High | High | High |
| **Change in Environ-ment.** | Medium | Medium | High | High | Medium | Medium High | Medium | Medium | High | High |
| **Tools and techniques used.** | Proto-typing | Project Man. | Pairs Prog. | Time-Box | No tool support needed. | None | OO, UML, GUI tool | Structu-red dev. Style. | Has own SEE | Diverse Toolset |

**Table 2.1:** Organizational effects of the SSA.

For example, the development process of INP and ICM are by nature traditional; which is a reason why they are considered to be of medium complexity. The increase in the complexity level lies in their exploitation of the incremental development concepts. It is interesting to observe that the learning curve and the skills level synchronize with the complexity level. It stands to reason that the higher the complexity of an approach is, the higher the learning curve will be. Similarly these factors will influence the calibre of the development personnel that is required. The concepts of the SSA embody sophisticated software development procedures. It certainly demands a change in development strategy and insists on a skilled and committed team. The employment of specialists and the ability to work under minimum supervision to produce results is highly desirable with the SSA. For example, the SCR, XP and ASP approaches indicate this.

Software development does not take place in a vacuum, but within an environment. The three factors complexity, learning curve and skills level will influence the degree of changes that an organization must undergo should it adopt the SSA. A single SSA cannot be regarded as the best one to follow. The choice is relative and is influenced by the application domain. The SSA is not confined to development in small organizations, but it can also be used for large-scale software development as indicated by ASP. Furthermore, they maybe used in conjunction with one another. For example, XP techniques can be incorporated with PLE. Improvisation techniques may be used with any approach. This topic is discussed in detail in chapter 3. Therefore whatever approach is adopted; through careful planning it is possible to select "the best of the breeds". As a result SMO can attain optimal benefits.

Although there is a need to manage changes in the requirements and have a shorter cycle-time, the

SSA does not neglect basic SE principles. These principles are applied in various degrees of detail and complexity. For example, the Integrated Approach, Scrum, EPM, Wisdom and PLE would appear to be conservative and more methodical with regard to the development phases. That is, the phases are simplified, while XP is more radical with a "just do it" development style. The software engineering process of ASP is the most complex and may be difficult to implement. Thus a software engineering environment is recommended for this approach. The ASP model has its own Software Engineering Environment (SEE) that is compatible with its principles. The other approaches do not have their own SEE and the emphasis was on the development process. The development tasks are concurrent and the environment is volatile. Since development speed is an important factor errors must be avoided. The motto is: "get it right the first time". Therefore the adherence to good SE principles is important to maintain control and discipline throughout the development process.

### 2.6.2 Solving Common Problems

It is interesting to observe that each variation of the SSA attempts to solve problems in a unique way. Despite these differences, they generally converge to solve common problems in Software Engineering in the Small. In Table 2.2 examples of these common problems are illustrated. The common focus of the approaches is the system's requirements, disciplined development, management of complexity, quick reaction to changes and a reduction in risk and cost. It is particularly noteworthy to observe the importance of the human element the approaches, namely, improved teamwork and communication. These aspects cannot be overlooked because a professional team behaviour and communication is one of the major contributing success factors with this sophisticated development style.

| Problems | ICM | EPM | XP | SCR | PLE | IMT | WSM | INP | ASP | SSC |
|---|---|---|---|---|---|---|---|---|---|---|
| Manage incomplete Requirements. | X | X | X | X | X | | X | X | X | |
| Employ discipline during development. | X | X | X | X | X | | X | X | X | |
| Manage system complexity. | X | X | X | X | X | X | | X | X | X |
| Quick reaction to changes. | X | X | X | X | X | X | | X | X | |
| Reduce cost, risk, use resources effectively. | X | X | | X | X | X | X | X | X | X |
| Maintain competitive advantage. | X | | X | | X | | | | X | X |
| Software reuse. | | X | X | | X | | X | | | X |
| Improved planning and estimation. | X | X | | X | | X | X | | X | |
| Improved teamwork and communication. | X | X | X | X | X | | X | X | X | |

**Table 2.2:** Common problems solved.

Although the SSA is used in practice by organizations such as Quark, Microsoft, Netscape and Mozilla it is not mature. With the constant increasing demand to develop high quality software rapidly, this software development trend will increase. It is a challenge for organizations to breakaway from the

traditional development approaches. A breakaway will spawn a new software development culture in an organization's environment.

Although the SSA may solve some problems, they also introduce new ones. The following are some examples:

- It is difficult to define the increment feature sets.
- An architecture decision early is difficult and is a key development factor.
- It is difficult to make estimates.
- Defining an infrastructure is difficult.
- There are performance problems with each increment.
- There may be software integration difficulties.

The presence of the above problems should not be a reason not to use a SSA. There is no perfect development approach and therefore the advantages of the SSA should be exploited and the problems managed. A sound project management policy and the correct implementation of SE principles can contribute to success and manage problems. In essence the SSA attempts to achieve what has already been explained by the acronym IRACIS (Gane and Sarson,1979). This translates to Increase Revenue, Avoid Cost and Improve Service. The consequence of IRACIS for an organization its that it is likely to strengthen its position.

In this chapter the theory and concepts of SSA were studied in depth. As incremental development forms an important pillar of SSA it was discussed in order to enhance its understanding. Over time numerous variations of SSA have emerged and apply the concepts of the incremental model. In order to gain insight into SSA and how they address the Software Engineering in the Small problems a detail literature review on the SSA was conducted. Many observations and examples on software development were cited from Quark's development environment in order to provide a balanced discussion between theory and practice.

Software development in SMO is a challenge in the constantly changing IT industry. The diversified approaches promise to face these challenges and to cope with difficulties which may be encountered. For a deeper perspective the SSA are analysed in detail in the next chapter.

# 3 SYNCHRONIZE AND STABILIZE – A Critical Analysis

## 3.1 Introduction

As previously noted, specific approaches were developed for help Small and Medium size Organizations (SMO) to cope with their challenging operating conditions. In the previous chapter the underlying principles of *synchronize and stabilize* and the incremental model were discussed. The implementation of these principles by the various Synchronize and Stabilize Approaches (SSA) were also examined. In this chapter the SSA are analysed in more detail. An analysis is important because it will help to consolidate the understanding of the underlying theory and concepts of the SSA in a non-superficial manner. As a result myths and misunderstandings about synchronize and stabilize may be cleared. An analysis will provide information that may be useful to support organizations in their decisions to adopt the approaches.

The discussion is presented in textual format and where appropriate tables and diagrams are used. The chapter starts with a general comparison of the SAA and is followed by a discussion on the common characteristics among them. The ability to combine development concepts among the approaches is also analysed. The SSA has its strengths and weaknesses and offers SMO many software development advantages. These are discussed and cognisance of the disadvantages is taken. In addition there are business advantages which organizations may exploit. The need for discipline with this rapid development approach is essential and basic software engineering principles with respect to the SSA are discussed. In order to demonstrate the synchronize and stabilize process practically a simplified version of the workflow that is used at Quark is described. Finally, the factors influencing the successful implementation of the SSA are discussed.

The abbreviated names of the approaches that are defined in table 2.0 (chapter 2) will also be used throughout this chapter.

## 3.2 General Comparison of the SSA

Although different SSA models consist of several common properties, for example, incremental development, rapid delivery, management of complexity and reduction of development costs; they also differ in many aspects. The following are some differences:

- Each approach has its own software development procedure.
- They apply different techniques.
- They consist of different phases and sub-phases.
- They have their own development terminologies.

| SSA | Philosophical View | Problems Addressed | SSA Views on Waterfall Model |
|---|---|---|---|
| **ICM** | Develop system in series of increments over a time scale. Deliver system incrementally. | Poor estimations, requirements not clear, difficult to adapt to systems changes, integration and testing is problematic, costly maintenance. | Schedules difficult to estimate, high staff turnover causes delays, resources not used optimally. |
| **EPM** | Each increment is a mini project. Emphasis is on rapid learning and project management. Apply knowledge to make improvements. Formality is progressive. | Detail requirements and design are initially difficult. Too many unknowns. Project estimates poor. System rarely finished when delivered – because it evolves. | Lengthy procedure, can affect productivity and motivation. Continuous refinement of development not possible. |
| **XP** | Satisfy customer early with continuous delivery of valuable software. Development speed is important. | Problem domain not clear in the beginning. Software must be delivered fast. System changes are costly. | Too rigid and time consuming. Delves too much into the future. |
| **SCR** | Assumes analysis, design and development unpredictable and complicated. These are high risk and chaos. | Systems are complex and unpredictable. Avoid risks and falling into chaos. Faster adaptation to requirements and business changes. | Being a linear approach is its largest problem. Cannot respond to the unexpected. |
| **PLE** | Method is specifically for product line SE. Large-scale reuse of functionality and distribute the cost and effort. | Small organizations require fast reaction to customers needs and must be flexibility. Faster delivery of software. | No views were expressed. |
| **IMT** | Find a balance between exploiting existing skills and experiment with new ideas. | Small development environments are non deterministic and multidirectional. Must cope with unseen. Manage constant system changes. | No views were expressed. |
| **WSM** | Steers away from chaos. Leverages on characteristic of small organizations--speed, flexibility, and communication. Software reuse. | Requirements not clear. Financial and economical obstacles experienced. Improvements in development process required. | No views were expressed. |
| **INP** | Produce high quality software and timely results with low overheads for small projects. | Requirements are incomplete and the problem domain always changing. Process improvements and optimal use of resources. | No views were expressed. |
| **ASP** | Develop software quickly, while maintaining flexibility and respond to changing requirements. | Increased demand for software. Operating cost of development in global environment is high. Project delays. Quick adaptation to system changes and faster delivery. Manage requirements. | Process too slow. Based on volume of requirements which causes delays, reduces flexibility and productivity. Scheduling problems caused due to poor estimates. |
| **SSC** | Accelerate development, reduce cost and improve quality. | Increase software reuse and standardisation. Economical development and improved reliability and quality. | No views were expressed. |

**Table 3.0:** A general comparison of the SSA.

Apart from these differences, each approach is based upon its own philosophy. It also identifies the nature of the problems it intends to solve. Some of these problems are common among the approaches. The philosophy upon which an approach is based will influence the path that it takes to solve the problem statement. As a result the problem solving methods and techniques which are applied will vary. For example, PLE is more suitable for product line development and, while IMT has a stronger focus on rationalization of existing resources and experimenting with new ideas. Both these approaches apply different methods and techniques.

In Table 3.0 the philosophy of the SSA is summarized in relation to the problems they intend to solve. Included is their point of view on the waterfall model. The philosophies vary, but they also share some

common beliefs. Some examples of them are, incremental development, rapid development, flexibility, software reuse and high quality software. The management of the system's requirements and fast reaction to changes during the development process is a major common concern. This is not surprising because it is a well-established fact that poor requirements have resulted in many software project failures.

Although some of the approaches have not expressed any views on the waterfall model, their philosophical views strongly imply its unsuitability. They do clearly emphasize that fast development and flexibility is required, which the waterfall model lacks. This is evident with the PLE IMT, WSM, INP and SSC approaches.

The flexibility of EPM is interesting as it can be applied to almost any type of project. According to Gilb [W5] it has been applied to small and large-scale software engineering tasks, such as aircraft engineering, telecommunications engineering and information system projects. The principles of EPM may be a suitable project management technique for SMO.

## 3.3 Common Characteristics of the SSA

Although there are different philosophical views, the SSA have many *common characteristics*. In other words the processes or procedures which are applied to solve the problem statement overlaps. However, the fact that there are common characteristics among them does not mean that an approach may be easily substituted for another. The implication is that overall an approach addresses similar problems, but according to its philosophical view. The choice of an approach might be influenced by several factors such as:

- The business philosophy
- Nature of the application
- The development environment
- Availability of resources.

The choice of a SSA is not simple. In Table 3.1 the common characteristics of the approaches are illustrated. The first column of the table indicates the common characteristics. The 'X' associates the characteristics with an approach. Although an approach did not explicitly mention a characteristic, it is presumed that it would be present. For example, it is presumed that XP would employ some form of project management procedures during development. IMT is unlike the other approaches as it focuses on exploiting the environment and exploring new ideas in the development environment. The results or side effects of these two factors are the catalyst for small organizations to cope with development problems. In comparison with the other approaches its proposals are innovative and may be effectively applied in conjunction with the other approaches.

| Characteristics | ICM | EPM | XP | SCR | PLE | IMT | WSM | INP | ASP | SSC |
|---|---|---|---|---|---|---|---|---|---|---|
| Use build Procedure | X | X | X | X | X | | X | X | X | X |
| User Participation | X | X | X | X | X | | X | X | X | |
| Incremental Development | X | X | X | X | X | X | X | X | X | X |
| Fast development, delivery | X | X | X | X | X | | X | X | X | X |
| Exploratory | X | X | | | | X | | | | X |
| Use Prototyping | X | X | | | X | | X | | | X |
| Team Oriented | X | X | X | X | X | X | X | X | X | |
| Uses some Formalism | X | X | | X | X | X | X | X | X | X |
| Emphasis on good Architecture | X | X | X | X | X | | X | X | X | X |
| Project Management | X | X | | X | X | | X | X | X | X |

**Table 3.1:** Common characteristics of the SSA.

Although one of the intentions is to develop software in a non-bureaucratic manner, most approaches apply some formalism during development. For example, EPM, SCR, PLE, INP ASP and SSC are structured and formal in their development processes. Formal procedures are of short durations at Quark and are a mechanism for control and structure during software engineering. Discussions are focused strictly on the subject matter and meetings are kept short. Any topic that is not directly related to the meeting's scope is discussed outside among the interested parties. Large volumes of written formal documentation are avoided because it is time consuming to write them and often they remain unread. They are also quickly outdated. The software specifications are as brief as possible. In some exceptional cases it is difficult to achieve brevity. As with the Use Case documents which are compiled by the product management department. It is not easy to identify the system's dependencies from them and these become known later during the implementation phase. Some rework is likely to cause project delays.

The SSA also considers good system architecture design an important characteristic. The goal is to design an open architecture, however, it is difficult to foresee every system changes that may have an impact on the architecture because the system evolves over time. In Quarks E-Commerce application, the system's architecture design was a neglected area and it was not long before architectural problems revealed their impact. The main problems encountered were module integration, system performance and stability. A task force was appointed to address these problems. A lesson learnt was that "repairing" a system's architecture is not trivial. If the problem is addressed early then the task may be less cumbersome, but there is no guarantee.

## 3.4 Combining Concepts among the SSA

Apart from the common characteristics the SSA possesses, it is possible to apply the concepts of an approach in conjunction with another in order to exploit its good techniques. The advantage is that the

deficits of a SSA may be overcome by applying a technique of another. This process will require careful planning in order not to conflict with the concepts and guidance offered by the approach that was adopted.

| SSA | ICM | EPM | XP | SCR | PLE | IMT | WSM | INP | ASP | SSC |
|---|---|---|---|---|---|---|---|---|---|---|
| **ICM** | | X | | | | X | | X | | X |
| **EPM** | X | | X | | | X | | | | X |
| **XP** | X | X | | X | X | X | | | | X |
| **SCR** | X | X | X | | X | X | | | | X |
| **PLE** | X | X | X | X | | X | X | X | X | X |
| **IMT** | | | | | | | | | | |
| **WSM** | X | X | X | | X | X | | | | X |
| **INP** | X | X | X | | X | X | X | | | X |
| **ASP** | X | X | X | | | X | | | | X |
| **SSC** | X | X | X | | | X | | X | X | |

**Table 3.2:** Combining concepts among the SSA.

In Table 3.2 the possibility to use the concepts of an approach in conjunction with another is illustrated. The first column indicates the SSA which may use techniques from other SSA. They are summarised as follows:

- The Incremental Model (ICM) may use EPM as a project management technique and apply the principles of IMT. The ICM may adopt techniques from SSC.
- Evolutionary Project Management (EPM) can apply the incremental techniques of ICM and pairs programming from XP. EPM may adopt techniques from SSC and the principles of IMT may be applied in its development environment.
- Extreme Programming (XP) can apply the incremental techniques of ICM and use EPM as a project management technique. Means to manage chaos may be adopted from SCR. If the application domain is a product line then some development strategy may be adopted from PLE. If components are used then XP can adopt techniques from SSC. The principles of IMT may also be applied.
- The Scrum Software Development Process (SCR) can apply the incremental techniques of ICM and use EPM as a project management technique. The technique of pairs programming from XP may also be applied. The concepts of PLE may be adopted if the application domain is a product line. If components are used then SCR may benefit from concepts of SSC. The principles of IMT may also be applied.
- Product Line Engineering (PLE) may apply the incremental techniques of ICM and use EPM as a project management technique. The technique of pairs programming from XP may also be incorporated. Means to manage chaos may be adopted from SCR. A PLE project may also adopt the concepts of WSM, INP and ASP during development. Since components are used to build products fast, PLE can benefit from the concepts of SSC.
- Improvisation Techniques (IMT) slants towards being a management technique to rationalize

an organization's resources and its theory may be applied to any SSA. It is based on strategic decision-making and a dedicated research team is required to identify what to explore and exploit. A highly skilled team is required for this task.

- Wisdom (WSM) can apply the incremental techniques of ICM and use EPM as a project management technique. The technique of pairs programming from XP may also be applied. The concepts of PLE may be adopted if the application domain is a product line. If components are used then WSM can benefit from the development approach of SSC. The principles of IMT may also be applied advantageously.
- The Integrated Approach (INP) can apply the incremental techniques of ICM and use EPM as a project management technique. The concepts of WSM blend with those of INP as they have much in common. The technique of pairs programming from XP can also be applied. If components are used then INP can benefit from the development approach of SSC. The principles of IMT may also be applied.
- The Agile Software Process (ASP) can apply the incremental techniques of ICM and use EPM as a project management technique. The technique of pairs programming from XP can also be applied. If components are used then ASP can benefit from the development approach of SSC. The principles of IMT may also be applicable to the development environment.
- The Synchronization and Stabilization using Components (SSC) approach can apply the incremental techniques of ICM and use EPM as a project management technique. The technique of pairs programming from XP can be applied to develop components rapidly. The concepts of ASP and INP may also be adopted for the development of a system with components. The principles of IMT may also be applied.

The discussion above demonstrates that the SSA complements each other. It can be observed that ICM, EPM, IMT and SSC are common to most of the approaches. Whether the concepts of the SSA will be used in conjunction with one another in practice will depend on the development policies practised. Nevertheless the possibility exists. Approaches such as SCR, ASP, WSM and XP may be regarded as "methodologies" on their own and it is unlikely that they would be applied in conjunction with others.

At Quark, many concepts from EPM, ICM, PLE, XP, SSC have been successfully applied in its E-Commerce application. With regard to XP pair programming, code factoring, collective ownership and continuous integration are practised. Unfortunately it is difficult to maintain a 40 hours working week as suggested by XP and most developers in Research and Development (R&D) exceed it. The reason is that the volume of work is high and skilled human resources are limited. With reference to IMT Quark is always exploring new technologies in order to stay competitive. For example, Quark did not simply introduce the .NET technology into its development environment. A team was assigned to evaluate the technology before implementing it. However, once technology is considered to be outdated, no attempt is made to exploit it because in Quark's business environment it would not be economical.

## 3.5 Strengths and Weaknesses of the SSA

As with any methodology it is important to understand the strengths and weaknesses in order to exploit it and to plan a defence against potential problems respectively. In Table 3.3 the main strengths and weaknesses of the approaches are summarized and are self-explanatory. The likely improvements of the approaches are also noted. The effort that is required to adopt an approach is derived in conjunction with Table 2.2 (chapter 2). The strengths of the INP are interesting as it combines the strengths of other standard development processes (Russ and McGregor, 2000). Hence, since these are familiar processes, the usage of this approach may be easier. However, it is difficult to find skilled personnel to fill the specific roles which are required by the INP. Generally, well-trained personnel are essential with the SSA otherwise the advantage of speed in the development will be diminished.

Proper development tools and a Software Engineering Environment (SEE) among the SSA would be an improvement to them. For example, the PLE and WSM approaches do not really have a comprehensive tool support. Since software development is a complex process and the development activities increase with each increment, appropriate development tools are indispensable. They contribute to the development speed and software quality. Russ and McGregor (2000), also found that tools are helpful to automate the development process.

The effort to adopt a SSA ranges from medium to high. It is an organization's strategic IT decision that requires careful planning in order to avoid setbacks. A change in thinking is also necessary should an organization make a transition to this development approach. Organizational changes are costly and may be met with resistance. Therefore a sound change management process must be implemented to ensure a harmonious transition and acceptance. It is not trivial to make this transition and an organization must be prepared to accept that there is a learning curve to follow. The Integrated Approach, EPM, Wisdom and PLE may to be easier to adopt, while ASP, XP, Scrum and SSC are more complex. Nevertheless, when new development methods are adopted they must be learnt and proper training is essential.

| SSA | Strengths | Weaknesses | Improvements | Effort to Adopt |
|---|---|---|---|---|
| **ICM** | Simple to use. Based on sound engineering principles and is mature. | Unsuitable for modern software development. Can easily degenerate into a build and fix approach. | A development toolset and SEE may be useful. A process model would be useful. | The effort to adopt is medium. |
| **EPM** | Strong in project management. Focus on learning system. Better estimates.<br>Suitable for diverse project types. | The process can become long drawn. May be difficult to get management commitment. Change in environment needed. | A EPM tool could be integrated into a SEE environment. | The effort to adopt is high. A commitment to project management is required from top management. |
| **XP** | Development is rapid. Encourages team work. Quick reaction to change. | Pair programming can cause conflict-programmers are individualist. Highly skilled people - are difficult to find. | The hybrid of SE practices used could be consolidated into a process model. A SEE may be useful. | The effort to adopt is high. Viewed as a new development culture for the organization. |
| **SCR** | Has good planning strategy. Considers unknown, risk and complexity. Designed to cope with changes. | User sets release date – potential for conflict high. Can easily fall into chaos if process not well managed. | Reduce Scrum meetings – people could feel being "policed". | The effort to adopt is high. Viewed as a new development culture for the organization. |
| **PLE** | Structured and encourages reuse and good system architecture. Fast delivery with long-term vision. | Reuse can be limited- products in range must be compatible for optimal reuse. No tool support. | Development can be improved through a suitable tool support such as a SEE and process model. | The effort to adopt is medium to high. |
| **IMT** | Good theory for resource rationalization. Economical to adapt where resources are scarce. | Requires a special team for exploration – this resource is expensive. Process can be slow for small organizations. | The method could suggest a model for exploitation and exploration. | The effort to adopt is low to medium. A skilled team is needed to do research and make recommendations. |
| **WSM** | A well structured and formal development process. Strong focus on usability (GUI design). | No high-end tools used - restrictive and can slow development. | Development can be improved through suitable tool support such as a SEE. | The effort to adopt is medium. A change in development habits is required. |
| **INP** | Mixes strengths of other standard process models. Process simple to understand, to use and well structured. | The eight steps may slow development.<br>Highly skilled people required are hard to find to fit the roles allocated. | Development can be improved with support of a SEE and automation in the development process. | The effort to adopt is medium to high. A change in development habits is required. |
| **ASP** | Structured and highly disciplined. Has its own SEE. Successful in global development environment. | Highly skilled people are hard to find. Complex, requires radical environmental changes. Is costly. | The process could be simplified and complexity reduced. | The effort to adopt is high. |
| **SSC** | High focus on reuse and quality. Saves development time – use what is already available. Save on development personnel. | Incompatibility and integration problems. Weak components weakens a system. Hidden costs. Highly skilled people needed. | A SEE and a process model could help to manage the development. | The effort to adopt is high. |

**Table 3.3:** The strengths and weaknesses of the SSA.

## 3.6 Software Development Advantages

SSA as a development philosophy has many advantages. It helps to cope with typical development problems, which are experienced with other development models. The following advantages are discussed in detail:

- Improved Requirements Elicitation
- Early Feedback
- Knowledge Gained is "Re-invested"
- Ability to Cope with Changes in the Environment
- Impact of error corrections can be reduced
- Improved Management of System Scope
- Manage Complexity
- Motivation.

### 3.6.1 Improved Requirements Elicitation

A decision on what must be developed and the scope of the requirements is a problem of uncertainty. Normally product management decides on the scope of the system. The decision is usually based on a business case, product research and product vision. However, these are inadequate to write a complete system requirement specification. The application domain and customer's preferences change fast and the product must accommodate changes. Hence, writing a complete requirement specification in the beginning is wasteful. Another problem is that there is never enough time to write complete requirements since the time to market is accelerated. A solution to this problem is to specify and developed *only* the features which are required for an increment. With this method the requirements are clearly focused and rework is reduced. In practice the requirements of a feature can be specified in ten pages. Should the functionality of a feature change in subsequent increments, then only those changes are specified and implemented. Similarly, if new features are added, then the requirements for them are specified.

Theoretically the requirements are likely to be complete because instead of creating volumes of documents, the task is divided into manageable increments. Thus the requirements are managed in a flexible manner. In this manner it is easier to incorporate changes and they are resolved iteratively. These advantages are not easy to achieve with the waterfall model. Although an effort is made to freeze the requirements, it rarely happens. With incremental requirements, the scope of the system's functionality would evolve cumulatively.

### 3.6.2 Early Feedback

After an increment has been shipped, *customer feedback* is received. The following are some common sources of feedback:

- Fairs and trade shows, for example, Cebit, COMDEX
- Directly from the customer

- Software distributors
- Analysis of calls made to an organization's Call Centre.

Early feedback is valuable to improve the product and influence the development of future increments. Customer feedback will be transformed into a requirements document in order to be specified for implementation. Therefore the initial specifications can be constantly adjusted to accommodate changes in market trends and remain competitive. As a result, only when the project is complete, are the specifications considered to be complete. This is different from the waterfall model, where development commences *after* the specification is complete (Cusumano and Yoffie, 1999). The continuous tweaking of the system through the SSA can improve its quality and usability. It is likely that the system will be up to date because of the constant adjustments made to it.

### 3.6.3 Knowledge Gained is "Re-Invested"

During the development of each increment, the system goes through several formal design review cycles, presentations (to users and management) and informal discussions. Workshops and brainstorming sessions are also conducted to discuss specific topics and problems. During these sessions there is a refreshing exchange of views, ideas, suggestions and experiences since people have different educational backgrounds. The system is viewed from a multi-dimensional perspective and is valuable to initiate suggestions for improvements. For example, during a design review in 1999 in which the author participated, the Graphical User Interface (GUI) was reduced from nine screens to one. In the alternative design the screens were replaced with a tabbed user interface in order to simplify the interaction. Quark strives for a state of the art user interfaces and strongly emphasizes on usability. It has prescribed Human Computer Interaction (HCI) and GUI guidelines which must be followed. A specialised team is assigned to design them.

As a result each increment has a learning curve and knowledge gained can be "re-invested" into the development process. Any knowledge that is gained is cumulative throughout the development and is useful to reduce errors, improve quality and improve usability. The team also learns to avoid similar mistakes and problems in the future. This results in reduced rework and faster development. Apart from the general improvements experienced, good communication and co-operation are also fostered among the teams.

### 3.6.4 Ability to Cope with Changes in the Environment

The continuous growth in the software industry and rapid changes in trends and technology are indisputable. During the development process unforeseen technical changes can occur in hardware and software. As a result the system must also support new technologies. With the introduction of Windows XP Quark had to reschedule its development to support this platform. The task was not simple because the new technology must be understood and is time consuming. The numerous system compatibility issues must also be carefully analysed. Despite the problems encountered, these changes may be accommodated without waiting until the system is completed. Since each increment

is a "complete system entity", it helps to cope with these difficulties. Therefore throughout the development cycle the probability that the system is technologically up to date is high and is a competitive advantage.

### 3.6.5 Impact of error corrections can be reduced

Errors will certainly be introduced throughout the development and will reveal themselves at anytime in various forms. These could be product errors where the expectations of the system have not been met, or process errors where the system is not developed according to the requirements. Errors can be identified under the following circumstances and must be corrected:

- Testing
- Document reviews
- Code review
- Design reviews
- User acceptance audits
- Quality assurance.

The early detection and correction of errors contributes to reducing the rippling effects they have in a system. It is also economical. With SSA, errors *are* detected and corrected early in the development. The daily build and the smoke tests (discussed earlier) are important mechanisms for early error detection and to monitor system stability. The side effects of system fixes may be reduced and simultaneously stability is increased through the SSA. Should unusual system behaviour occur, the latest changes are the most likely causes.

This situation was often experienced at Quark where most problems were caused by the latest system changes. An extreme method to debug the system was to back out the changes and re-introduce them incrementally. This method was faster than using a debugger. Apart from incremental software development, error detection may also be incremental in extreme situations. The developers will always make errors and is difficult to prevent. Therefore with the concepts of incremental techniques, continuous corrections and improvements are effectively managed.

### 3.6.6 Improved Management of System Scope

An accurate system specification is difficult to define, as it is inevitable that changes will be required. There is also an unbound enthusiasm and the potential danger to deviate from the scope by being over ambitious. With constant *synchronizing and stabilizing* of the system is easy to detect a deviation from the scope and take remedial action. On the contrary, early customer feedback may make it necessary to deviate from or extend the scope in order to remain competitive. An early recognition of these signs in manageable quantities enables improved management over them. Since the system is developed in increments, deviations will also be in increments and are more manageable. This situation is unlike the waterfall model where many changes are discovered much later in the

development. Therefore should there be any rework, the volume is likely to be reduced. Rework and scope deviations are cost drivers which can be reduced if they are addressed early.

### 3.6.7 Manage Complexity

Complexity is one of the many daunting inherent properties of software and will always remain (Brooks, 1987). New techniques, languages and paradigms will not eradicate the situation. Although they may solve some problems, however, others are introduced. Hence, the need to manage complexity will always be required. The interdependence and complexity among the modules of a system is difficult to understand and is difficult to communicate them to the project members. However, by decomposing a system into manageable increments and limiting the focus on subsets of the total functionality, complexity is reduced and information overload is controlled. In other words the scope of an increment is limited and the focus is temporarily narrowed to that increment. Theoretically, overall system complexity is reduced to increments in order to simplify its understanding. Although the growth of complexity *will increase* in an unpredictable magnitude with each increment, the developers will be able to learn the system in "manageable sections" and progressively increase their knowledge.

### 3.6.8 Motivation

The following are some echoes from the hallways of a typical development department:

- "Will this system ever be completed?"
- "If they keep on changing the requirement how can we finish..."
- "I feel that they do not know what they want..."

Even a top developer will be easily de-motivated when the development drags endlessly. Although IT personnel can be creative and churn up the most nifty ideas and solutions, they can become easily de-motivated, tired and bored when they get the impression that their efforts are being wasted. With the SSA each incremental cycle produces visual results of the developer's efforts. They are proud when the software is transferred on to the golden master CD ROM and shipped because their creation is on it for the marketplace to use. They identify well with their contribution towards the development of a product and are eager to enhance it in the next increment. Such a spirit can only have a positive effect on a project and is difficult to experience with the waterfall approach.

The *cumulative property* of the SSA is briefly examined below as it helps to understand the discussion of the previous sections. In Figure 3.0 the *cumulative process* of the synchronize and stabilize cycles is illustrated. The total number of increments of the development cycle will depend on the application. The *first increment* is a "bare bones" version of the system with limited functionality.

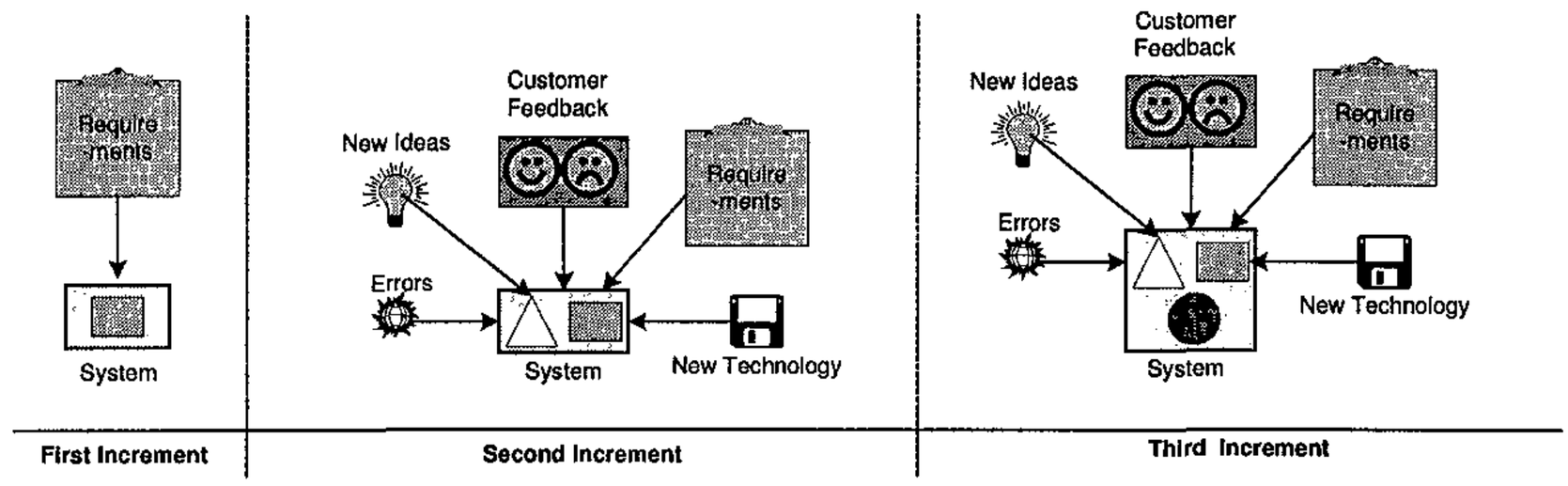


**Figure 3.0:** The cumulative process of the synchronize and stabilize cycles.

From the *second increment* onwards the development activities are more dynamic as compared to the first increment. Customer feedback is expected to have been received and may be incorporated into this increment. Thus the system is incremented to the next level of functionality. Knowledge gained and suggestions from the development team are also incorporated. Errors which were detected are also corrected. If there is a requirement to adapt the system to new technology, then it will also be included. The figure also illustrates the *phased increase* of the system's functionality and complexity during each increment. In this manner the management of complexity is improved. All these activities occur concurrently. In reality the development team is under increased pressure to complete the development as scheduled. The *third increment* will also proceed in a similar manner and result in increased growth of the system. Therefore it is evident that the SSA promotes continuous system improvements throughout the development cycle which is difficult to achieve with the waterfall model.

## 3.7 Software Development Difficulties

Although there are many important advantages through the SSA, there are also disadvantages. It is important to understand them so that they may be managed. The following are some of the disadvantages which are encountered during practice:

- Architecture Decisions are difficult.
- Increment Sets are difficult to define.
- Defining an Infrastructure early is difficult.
- Performance Problems arise.
- Software Integration is difficult.
- Multiple Maintenance Syndrome.
- The process may degenerate into a build-and-fix approach.

### 3.7.1 Architecture Decisions are difficult

The choice of system design and architecture cannot be easily determined during the implementation of the first increment. Figure 3.0 clearly demonstrates a system's evolution with each increment and. changes may occur in complexity, functionality, performance, size and stability. These changes are

cumulative and the architecture is expected to seamlessly accommodate them. Changes may also result in the architecture to lose its original clarity and integrity because it may have to be restructured or partially recreated (Rajlich, Bennett, 2000). The architecture team will do its best to define an extensible architecture that can evolve with future increments. However, it is difficult to make early architectural decisions on which to base the future of the system. Therefore good architectural decisions are essential for successful system evolution; otherwise a negative impact can cause it to degenerate. In the worse case a damaged architecture cannot be "repaired" and may have to be re-engineered.

### 3.7.2 Increment Sets are difficult to define

Although the SSA endeavours to deliver software rapidly, it is difficult to decide which system features must be developed in an increment set. Generally product management decides on the systems features and the scope of the increment sets. Sometimes this is difficult because competition demands that some features be developed earlier than planned. It is also possible that a feature that was planned for an early increment may be exchanged with one that was planned for a later increment. Although an increment set is clearly defined, it is a common occurrence that one of its features cannot be developed because it is dependent on a feature that is scheduled in a later increment. Since there are interdependencies among the features, they cannot be treated as Lego playing blocks because they impact the whole system.

During the development of Quarks E-Commerce application, it was decided that a function to pass customer credit (credit memo) in the first increment was not required. However, four weeks before the end of the schedule it was decided that a credit memo function was necessary. Since resources were low to develop the new functionality, it ended as a "hack" in the system and the code was restructured *one year later*. However, after the credit memo function was completed, it was realized that a debit memo function was required to cancel a credit memo. Due to low resources this functionality was a second "hack" to the system. Although the net result of the functionality was correct, the solution was sub-optimal because it violated proper accounting procedures. Therefore careful planning of the feature set is vital.

### 3.7.3 Defining an Infrastructure early is difficult

In addition to the difficulties experienced to make early architectural decisions, it is also difficult to decide on a correct hardware infrastructure for the system early in the development cycle. Since the scope of the first increment is limited, the infrastructure is also limited. As the system evolves with increased functionality and complexity, the infrastructure must also evolve to accommodate change. As a result new hardware must be purchased with added costs. Therefore although the final system is not completed, the application must be migrated to a new technical infrastructure during the course of the development. The infrastructure also has an impact on the quality of the system. This activity must be well managed because the risk for an increase in errors is high. The importance of a good system infrastructure cannot be neglected, because in modern software development infrastructure has

become critical in the new global economy (Jones, 2000). SMO *are* very active in the global economy.

A typical problem at Quark was that as the development progressed, the application could not run on the existing hardware and still satisfy the specified performance requirements. For example the size of the application, database tables and complexity increased. It was necessary to purchase new servers with increased capacity. Although purchasing new equipment helps, there are often delays in the development. The following are some of the reasons:

- The added cost must be justified and authorized by senior management. This process may take several weeks.
- The equipment may not be immediately deliverable. Hence, there are increased delays.
- Once the equipment is received, it must be set up, the application must be migrated to it and tested. All these activities are time intensive.

Therefore while the above acquisition process occurs, the development team is busy working with slower hardware which causes delays.

### 3.7.4 Performance Problems arise

As the system evolves, its code size, integration, functionality and complexity will grow in an unpredictable magnitude. The affect of these factors on the system's performance is unpredictable and modules may have to be redesigned to improve performance. Furthermore, the systems architecture may have to be re-engineered and changes may introduce new errors. Depending on the severity of the performance, an organization may have to delay the shipping of a release, which may result in the delay of future releases. Optimization strategies are difficult and require skilled personnel. Therefore such tasks should be assigned to specialists. At Quark this practised, namely, system tuning and performance issues *are* assigned to specialists. The task of solving performance problems will also require research, experimenting and appropriate tools.

The potential danger is that customers can become dissatisfied with degraded performance and lose their confidence in the system. If the situation is not promptly rectified, a loss of business is envisaged as they may seek alternative solutions.

### 3.7.5 Software Integration is Difficult

The development of the first increment (sometimes including the second) may give a false impression that the process is simple, because the development commences as a system of limited scope. The difficulty is to increment the existing system in each cycle by *integrating* it with new modules and sub-systems. This is represented by code at a lower level. This task is also subtle because an increment is regarded as a functional self-contained entity of the system. Therefore incrementing what exists may be also viewed as a form of "maintenance" of the previous increment. Although the development team will not refer to it as "maintenance", the characteristics are latently present. This aspect is also transparent to the customer because it is simply regarded as a new version of the system. Historically

software integration and maintenance are known to be difficult with unpredictable impacts on the system. These aspects make the SSA challenging. Since as changes are continuous with each increment, it is difficult to seamlessly integrate them.

### 3.7.6 Multiple Maintenance Syndrome

Since an increment is a self-contained functional entity of the software, it must be *maintained* because there will be errors. The software fixes and patches to an existing release must also be included in the release that is being currently developed. That means that new functionality is added and faults are corrected. To aggravate the situation, each release or version must be maintained separately and in parallel, because customers do not always upgrade to the latest release. For example, not all Ouark's customers have upgraded from QuarkXPress 4.0 to 5.0. Many of them did not consider it necessary and were not prepared to disrupt their work rhythm. Therefore release 4.0 must be maintained and residual fault fixes must be incorporated into release 5.0. Currently QuarkXPress 6.0 is being developed and all fault corrections of previous releases must be incorporated in it as well.

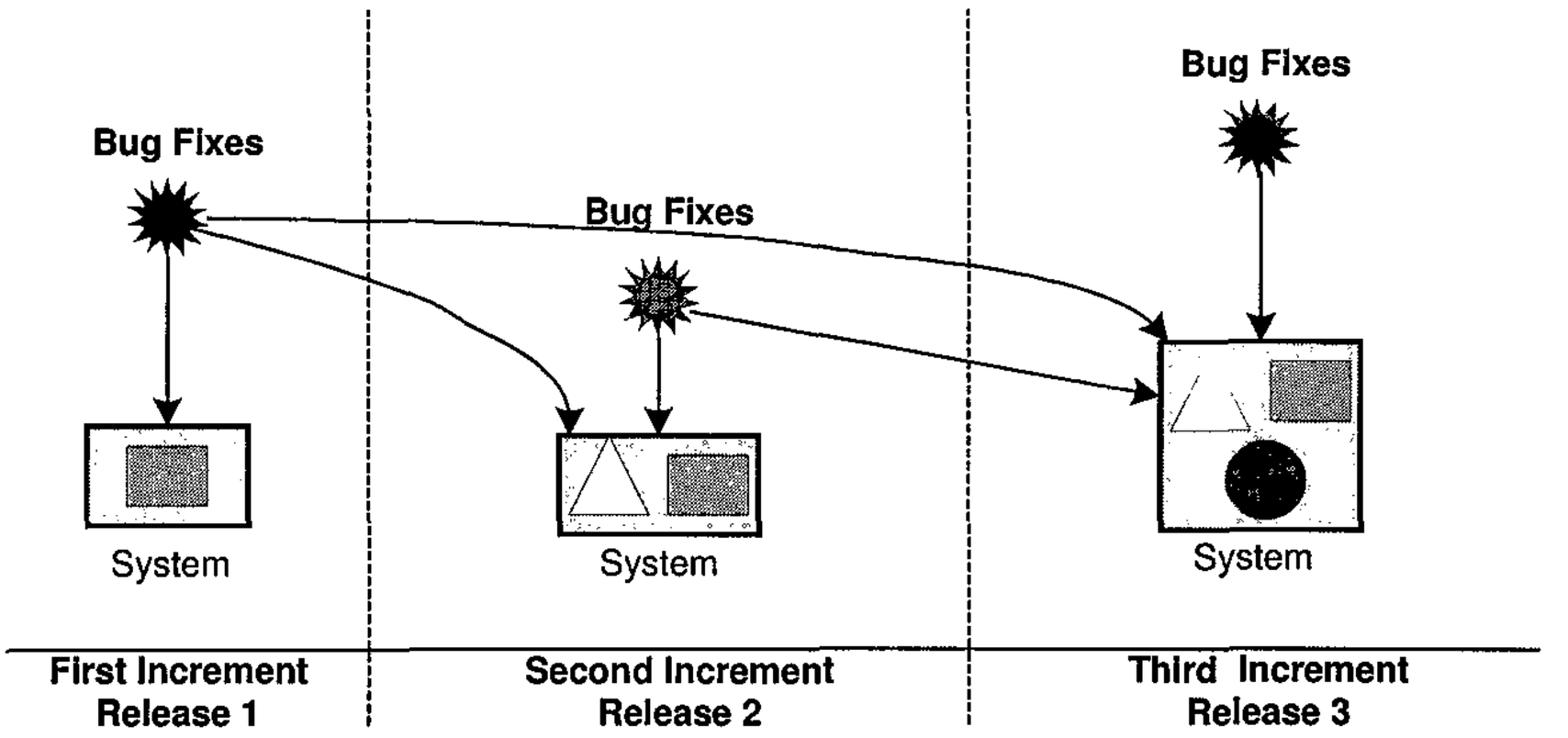


**Figure 3.1:** The complexity of maintenance of multiple software releases.

The complexity of maintenance of multiple releases is illustrated in Figure 3.1. The system is progressively aggregated with new increments. This will result in additional development resources being deployed. The multiple maintenance syndrome is further aggravated if the system supports multiple platforms. QuarkXPress supports Windows NT, Windows 2000, Windows XP and Apple Macintosh. Multiple releases must be maintained for these platforms.

A good co-ordinating and tracking mechanism is essential to ensure that all reported faults are corrected. The administration overheads are high and a good Software Configuration Management (SCM) team and version control tools are required. Conradi and Westfechtel, (1998) discuss this situation comprehensively and propose version models for SCM. They propose version models for both commercial systems and research prototypes. The fundamental SCM concepts such as

revisions, variants, configurations and changes are examined and an overview and classification of different versioning paradigms are provided.

### 3.7.7 The process may degenerate into a Build and Fix Approach

If the development process is not properly managed and control is lost, then the SSA can easily degenerate to a build-and-fix approach (Schach, 1993). Such a situation is not well suited for developing large products. A build-and-fix can result when staff is under pressure to meet deadlines. As a result software engineering principles are neglected and the development process is gradually transformed into a hacking of the system in order to make it function.

The author admits to this practice in order to meet a release deadline. However, a year later the patches were removed and the system was restructured. In practice it is necessary to deviate from the straight path, however, caution and discipline must be exercised. A senior team member should be informed so that the situation may be justified and monitored. Unfortunately when developers take shortcuts they rarely return to the problem area to rectify the situation. Their motto is simple: "never change a working program." If an organization implements code review procedures, then a hack in the code is likely to be discovered and a restructure can be requested. However, this practice should be avoided. Sound project management and realistic project schedules can help to avoid the build-and-fix approach.

The above difficulties are not trivial and good software management principles are required to overcome them. Despite the software development advantages and disadvantages, the SSA offers several business advantages.

## 3.8 Business Advantages

Small and Medium sized Organizations operate with limited financial and personnel resources. Their budget is limited and overheads must be kept as low. As the market is competitive, an organization cannot lag behind changes and trends if it wishes to improve its competitive advantage. One of the intentions of the SSA is to enable early delivery of a product to the market. Therefore an early Return on Investments (ROI) is experienced and helps to increase the confidence of the stakeholders and employees. Early ROI may be applied to finance future releases and hedge against financial risks. Since the development is phased, the capital outlay for the entire product is not required; only the part of the system that is being developed must be financed.

Shipping a product in increments is a useful way to monitor customer's behaviour and market movements. As a result the marketing department is provided with valuable feedback about the latest trends and can decide on what action is necessary in order to remain on course. For example, it may be necessary to adapt the marketing strategy in order to position the product. A constant monitoring of the business environment is important for an organization to apply its resources profitably. The hedge against market risks is improved because an organization can decide whether to continue with the

development or terminate it. Quark terminated its Site Midas application development after its first release because after a research it concluded that the product would not be profitable. Since the SSA adopts a phased development strategy, a ready version of the product is always available. It can be delivered when required.

The early delivery of a system is visual and tangible evidence of the team's efforts and increases their confidence. As a result they tend to stay longer with the organization. The advantage is that staff turnover is low and high replacement costs can be avoided. An organization will not welcome resignations of its key people right in the middle of its efforts to ship a release. Resignations are costly and can cause project delays.

Disciplined software development will benefit from good software engineering principles. In the next section some basic principles are discussed with respect to the SSA.

## 3.9 Applying General Software Engineering Principles

According to Moitra (1999) mainly large organizations and defence ministries have benefited from advances in Software Engineering (SE). Most organizations were unsuccessful in applying SE principles and practices, the main reason being their application in an incorrect context. Although the current SE principles do not integrate well into their business environment, it does not imply that they should be disregarded. The attempt of the SSA to produce results quickly does not mean that development must be haphazard and chaotic. The idea of dispensing with complete specifications up front does *not* suggest that they will not exist at all. It just means that they will be completed in phases.

Iterative development is more challenging than the waterfall approach and a disciplined software management and common sense are two important discriminators of SE success or failure (Royce, 2000). The application of good software engineering principles are indispensable for disciplined software development. In this section the following general software engineering principles, which may be applied with the approaches are discussed:

- Formality
- Reduction of Complexity
- Generalization of the Solution
- Incremental Development
- Re-usability and Maintainability.

### 3.9.1 Formality

Software Engineering endeavours to apply strict formality during software development. This includes applying formality in the analysis of the problem statement, system design and implementing a proper plan of what to do. Formality in this context means defining a clear structure, rules, procedures and techniques to execute the development tasks. Formalizing the development process includes:

- Setting Software Development Standards and Procedures

- Conventions
- Quality Control
- Project Management and Costing.

Realistic project goals and objectives must be defined. They are essential to monitor the progress of the development and to identify problems early. If there are deviations from the planned goals, then appropriate action can be taken. The result of proper plans and controls contributes to lowering the error rate to an understandable and controllable level and at the same time increasing productivity (Goldberg, 1986). There is little doubt of the impact that these have on the approaches in modern software development.

### 3.9.2 Reduction of Complexity

Software is complex, intricate and has many interdependent components, which interact together to produce results. Complexity affects the management and maintenance process. The greater the complexity, the greater is the difficulty to modify it or detect errors. Complexity is an inherent property of software that makes it difficult to construct (Brooks, 1987). According to Parnas (1985), the complexity of a software system is caused by a very large number of states contained in it. These prohibit the developer from obtaining a complete understanding of the behaviour of the system.

The reduction of complexity can cause two gains; the code is likely to have fewer errors; and should it be modified, it is easier to understand it in order to make the modifications (Jones, 1993). Software Engineering aims to deal with complexity methodically. A problem is first abstracted in its generality in order to understand the gist of it. Gradually it is decomposed into smaller problems and the fine points are analysed. Through abstraction, step-wise refinement and modularisation, the essence of the problem statement is easier understood and humanly manageable. The SSA attempt to manage complexity through a *divide and conquer* software development process.

### 3.9.3 Generalization of the Solution

Instead of searching for an exact solution to the problem statement a general solution is sought. Several solutions are critically examined and the pros and cons are evaluated while focusing on the problem statement. By initial generalization and then drilling down to an exact solution, an optimal solution is likely to be found, rather than a haphazard one. Since the SSA is a phased development process, finding a final solution to the problem statement applies an *exploration and learning* process. This principle is appropriate to the SSA since the initial system requirements are not clear. Therefore the requirements may be generalized in the initial phases and refined as the development proceeds.

### 3.9.4 Incremental Development

Through formalism the solution to the problem statement is solved incrementally, namely, in phases. In reality, the boundaries of these phases overlap and there is never a clear cut off point before proceeding to the next phase (although this is highly desired). It is only human to make mistakes and

they must be corrected before proceeding. By proceeding incrementally it is easier to backtrack and make corrections. The incremental approach makes it easier to manage inputs and outputs of each phase. It also helps to manage quality control, and to perform validation and verification of the development. This software engineering principle is one of the pillars on which the SSA is built upon and is indispensable.

### 3.9.5 Re-Usability and Maintainability

The environment in which the software operates might change and the system must be adapted accordingly. The principles of software engineering can help to overcome these problems by applying modern software development techniques. For example, the object oriented paradigm supports encapsulation and information hiding. This results in richer modules being developed because they are loosely coupled, which makes them reusable. The SSA also seeks to promote reusability, such as PLE, XP, WSM and SSC. The increased module independence and loose coupling are effective to reduce the impact of changes on the whole system.

Although small software development organizations require a tailored application of software engineering principles and practices, the above principles are not a hindrance. They are straightforward and aligned with the SSA development philosophy.

To accelerate the development process, the workflow must be organised in a logical sequence. In the next section, the workflow of a typical synchronize and stabilize process as experienced in practice is discussed.

## 3.10 A Typical Synchronize and Stabilize Workflow

Earlier the turbulent conditions in which SMO operate were mentioned. Namely they operate in a highly unpredictable and dynamic environment. To ensure minimum disruptions and high productivity during the development process, the activities must be performed in clearly defined *workflow*. This is essential for some of the following reasons:

- Helps to identify roles
- Implies a channel of command
- Indicates the path of information flow and backtracking
- Indicates the order in which the development proceeds
- Helps identify problem areas in the development procedure
- Suggests management control points
- Indicates the expected deliverables at each control point.

The above list is not exhaustive. The discussion that follows is based on an example that is used by Quark. To simplify the discussion, unnecessary details have been omitted and the emphasis is on the main activities. The following is a simplified version of the workflow activities:

**PRODUCT MANAGEMENT ACTIVITIES**

- Environment Scanning
- Competitor Analysis
- Product Scope (product master feature list)
- Define Increment Sets (increment scope)
- Specify Use Cases for Increment Set.

**DEVELOPMENT ACTIVITIES**

- System Design
- Detail Design
- Implementation and Test
- Build and Test (integration and test)
- System Reviews.

**RELEASE MANAGEMENT ACTIVITIES**

- Alpha release
- Alpha bug fixing and testing
- Beta release
- Final development (final bug fixing, testing and tuning)
- Product shipping.

In Figure 3.2 a simplified workflow of the synchronize and stabilize activities is illustrated. The flow commences with Product Management's (PM) activities. The PM department scans the environment in order to identify a niche for a product. An analysis of competitors is also conducted in order to determine who is doing what and what may be improved. Generally, PM attends trade fairs and exhibitions to gather this information. After an extensive research a product and its scope are defined. The scope is defined in a product master feature list and describes the features of the product. The features are divided into three classes, namely base features, nice to have features and Unique Selling Points (USP). The USP is the most important and influences the product's competitive edge.

In the figure, PA indicates that the product master feature list is constantly revised in order to stay abreast with competition. PM determines the increment sets that must be developed. PC indicates that PM may change the scope of the increment set (add or remove features) with respect to the feature master list. Therefore the requirements are difficult to freeze because PM will always change their mind and a great degree of flexibility and co-operation is required. The requirements are specified as Use Cases (UC) and are sent to the development department. The arrow labelled PB shows that the UC may change because of changes to the increment set.

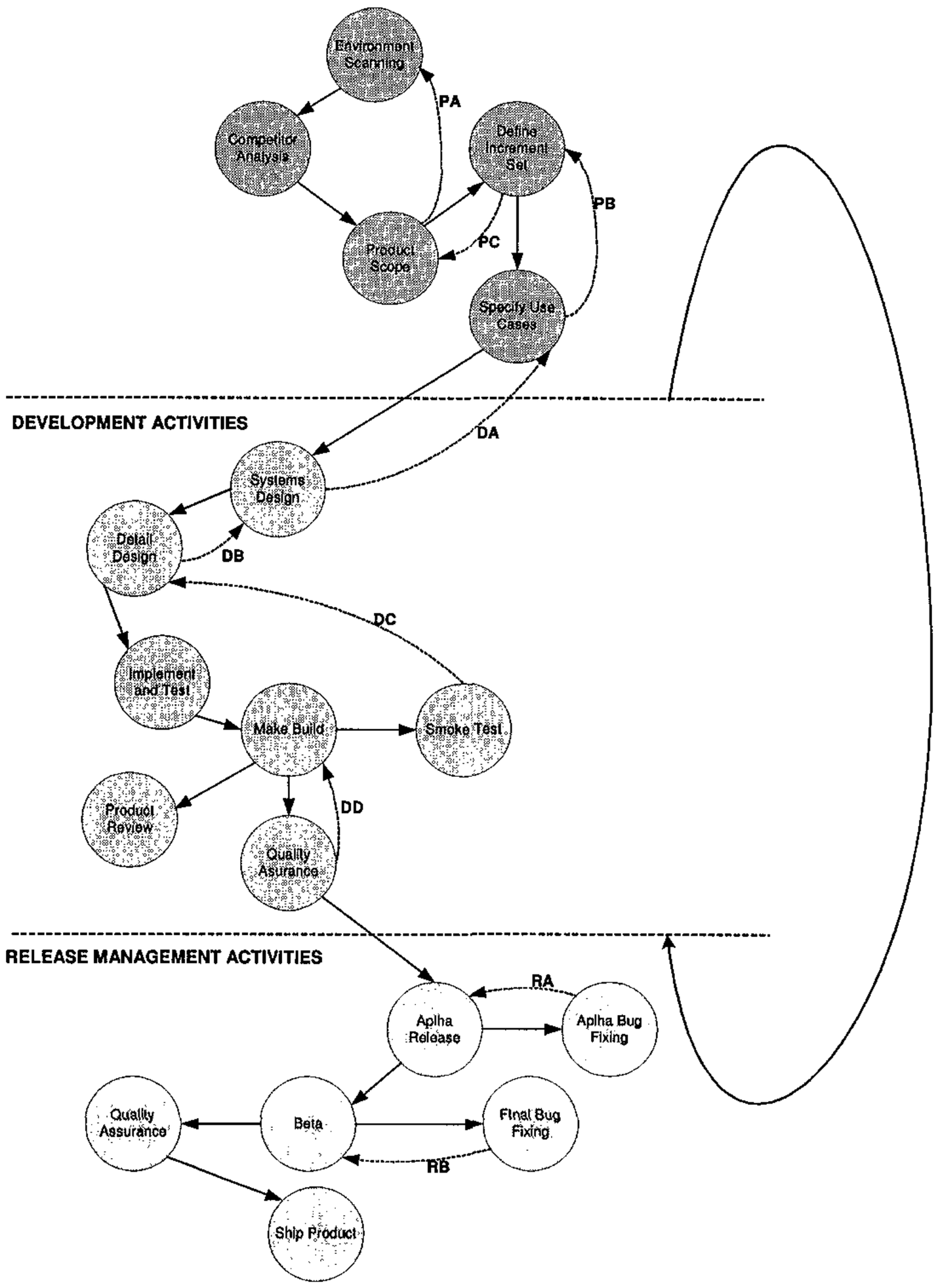


**Figure 3.2:** A synchronize and stabilize workflow.
*Source*: Adapted from: Quark's Business Architecture Document.

With respect to the Development Activities (DVA), a high level technical design is derived from the UC. The arrow labelled DA indicates the communication flow between DVA and PM. The communication is mainly in the form of queries and changes to the UC. Detail design follows and DB indicates that changes or problems must be rectified higher up. The detail design is in the form of a technical specification.

It is interesting to notice that the UC are transformed directly into technical specifications and the Analysis and Design (AD) phase is skipped. This was a decision taken by PM that the UC are comprehensive and that AD is not necessary. Unfortunately this method (still used) does not work well because the developers are technically oriented and have difficulties to comprehend the business requirements as stated in the UC. Previously AD formed the glue between PM and the developers. Therefore the communication traffic between development and PM (indicated by arrows DA and DB ) is extremely high. This problem was discussed with management; however, there was a reluctance to change. Based on the author's experience, AD should not be compromised.

The technical specifications are implemented and tested locally on the developer's computer. At a defined time of day the source code is checked-in with Software Configuration Management (SCM) team and the daily build is created. The build is smoke tested by the developers, domain experts and Quality Assurance (QA). The build and the smoke test combination are parallel to the integration and test phase of the waterfall model. The arrow labelled DC indicates the main iterative development cycle and is the *core of the synchronization and stabilization* process. Although it is not shown in the figure design and code reviews occur in this cycle. An important point is that QA will smoke test a build to determine whether they can accept it to perform detail product validation and verification procedures. The arrow labelled DD illustrates QA's rejection of the build. It has happened that a suitable build could not be produced for three weeks because of high rate of errors. However, once an increment is Engineering Complete (EC) and QA accepts a build that passes all checks, it will be released as an Alpha version. Quark has its own entry criteria to signoff an increment as EC. As an example the first criteria is stated below:

*"The requirements completely and accurately define the functionality that was intended to be developed. Product Management (PM) should ask Research and Development (R&D) the following question to determine EC, 'If a totally different R&D Engineer were to implement this feature from these requirements, would we have the same functionality that is currently implemented?' "* The list of criteria is comprehensive and a complete coverage of them is beyond the scope of the discussion.

During the Release Management Activities (RMA), the Alpha build continues to undergo intensive verification and validation procedures by QA. Fault fixing and fine-tuning will continue intensively as QA progressively reports defects. The focus is to reduce the level of severe faults, such as showstoppers. Cosmetic errors are allocated a low priority. The arrow labelled RA shows the Alpha fault fixing and testing cycle. If the requirements are satisfied, then the Alpha build is declared as a Beta version of the software and is installed and tested at appointed Beta sites. The PM department works closely with the Beta customers, whose feedback is valuable in the production of a highly useable and stable final release. The build is optimised for performance, stress tested and last minute errors are corrected. The arrow labelled RB shows the Beta fault fixing and testing cycle. QA also performs a final check on the build and if it is accepted, it is shipped. It should be noted that the final release is never fault free and is shipped with some residual faults. This is an accepted industry practice and is aligned to the notion of "sufficient" rather than "perfect" quality.

Although the activities are described sequentially, they are performed in parallel. The rapid development style of the SSA and the parallel performance of activities require good co-ordination during development. The success through the SSA is influenced by several critical success factors which are discussed next.

## 3.11 Critical Success Factors

With the adoption of any development approach a fair degree of success is expected and the same is true for the SSA. However, success will not happen without management's active role and proper steps must be taken. These steps must be well understood and be within the context of the organization's development philosophy. They must also be attainable and practical. The following are some critical success factors which management should consider:

- Commitment
- Quality Human Resources
- Keep up to date
- Planning
- Good Resource Management
- Clearly Defined Vision
- Clear Definition of Roles.

### 3.11.1 Commitment

Once a SSA has been adopted, its development processes must be well understood by *all* concerned and they must be committed to applying them correctly. Management must be convinced that the correct approach has been selected and must be the catalyst in achieving success. Apart from being committed, they must ensure enforcement.

### 3.11.2 Quality Human Resources

Since the SSA is a sophisticated development approach, a high calibre development team is indispensable. Namely, they must be well trained and possess good practical experience. The team must have a mature attitude and be able to work professionally under minimum supervision. Since the development environment is fast paced, it is not suitable for trainees. Most of the engineers at Quark are experienced graduates. This simplifies the task of assigning people to different tasks within short notice without or little training.

### 3.11.3 Keep up to date

The IT world is constantly changing with new methods and technologies are introduced constantly. Management must ensure that its organization is technically up to date and that its staff's skills are also updated. They must also be informed about the changes in the business environment and communicate this information within the organization. These changes have an impact on the systems developed by an organization and its vision.

### 3.11.4 Planning

Planning is not new to an organization. However, it is surprising how many organizations fail to plan effectively. Planning must be balanced and not focused only on project planning. For example, it is also essential to focus on resource planning, financial planning and business planning. All these plans influence each other. Therefore it is necessary to adopt a *holistic approach* to planning. The turbulence and complexities that small organizations experience, is a reason enough to be meticulous in this regard.

### 3.11.5 Good Resource Management

The limited resources of SMO must be well managed. For example, financial resources should be carefully managed so that capital is available to survive weak economic conditions, or to take advantage of opportunities. An organization can benefit in these situations. It is also important that budgets are allocated to the projects that will provide the optimal payback. Similarly, human resources should be well managed so that the work force is of high quality and stable. Human resource problems are a hindrance to an organization that is trying to get a system on to the market.

### 3.11.6 Clearly Defined Vision

With the approaches there are many unknowns and management must have a clear defined vision of their intentions. The product will evolve over time and management must look ahead to visualize what the final product will look like. They should be able to forecast whether the objectives are achievable or whether the resources should be applied elsewhere. Since environmental changes are frequent, a vision for new opportunities and innovative development is necessary. However, plans and visions do not always materialize as expected, therefore it is necessary to define fallback strategy.

### 3.11.7 Clear Definition of Roles

The importance of who does what, when and why cannot be over emphasized. In a fast paced development environment organizational clarity would reduce confusion and clarify the expected responsibility of the team. Furthermore, it helps to define the expected deliverables from the team members.

## 3.12 Conclusion

In this chapter the SSA were comprehensively analysed from various perspectives. As part of the analysis the following aspects were examined:

- Distinguishing the underlying philosophy of the SSA
- Understanding the benefits offered by the SSA
- Identifying problems and weaknesses
- Identifying business opportunities
- General characteristics of the SSA.

The discussion on the above aspects is important because it provides useful information in understanding the SSA. From a strategic point of view an appropriate decision will have an impact on the organization. Namely, whether its development approach is suitable for its business. Another impact is the effort that is required by an organization to adopt a SSA. For example, the effort required to adopt Wisdom is medium, while the effort to adopt the Agile Software Process is high. As previously noted, SMO requires flexible development techniques. An organization is not necessarily restricted to adhere *only* to the principles of the SSA that it has adopted. As the analysis indicates, techniques may be borrowed between the SSA. Therefore it is important to understand this flexibility of SSA for optimal benefits. This is a clear implication that the value of the synchronize and stabilize techniques cannot be ignored by SMO. There are shortcomings with the approaches; however, with good management principles their impact may be reduced. The classic waterfall model minimizes the concepts of incremental development and iteration. Unfortunately many projects of SMO do not lend themselves to the waterfall structure.

The application of software engineering principles as mentioned is clearly understood and easy to apply. The theory is supported with practical examples from the Quark development environment. These examples are important to illustrate real world software development and to enhance the understanding of the approaches. The workflow of the development activities, although simplified, is a clear indication of the complexity that is involved in developing a system and they rarely execute as smoothly as expected. The development activities are fast paced and are performed concurrently. Therefore a high degree of co-operation, maturity, independence, responsibility and professionalism is absolutely essential with this development approach.

As an organization will want to succeed through the SSA, the critical success factors may be used as a guideline. They are subjective and an organization must analyse them within the context of its own objectives and environment. Whatever the factors are, top management must be involved and enforce them in order to be workable. With this regard it is important for an organization to realize its ambitions and limitations.

A critical analysis does not only reveal the benefits and shortcomings of the SSA, but also helps to suggest the properties of an appropriate SEE and process model. In the next section a set of frameworks for that may be applied with the SSA is discussed.

# 4 A SET OF FRAMEWORKS

## 4.1 Introduction

In the previous chapter a critical analysis of the Synchronize and Stabilize Approaches (SSA) was discussed from multiple perspectives. *The intention was to provide an enriched analysis of the SSA that is supported by real world experience. Only through such an understanding it is possible to exploit the advantages which this development technique offers and implement the means to overcome its shortcomings.* It was also seen that an appropriate Software Engineering Environment (SEE) and a Software Process Model (SPM) for this sophisticated development style are usually neglected. Because the development pace is fast and many activities are performed concurrently, a SEE and a SPM are important supporting mechanisms for Software Engineering in the Small. The environment in which Small and Medium size Organizations (SMO) operates is dynamic and an appropriate SEE and SPM are vital to manage the development process in a disciplined manner throughout. The concept of best practices has often been reduced to a buzzword level. In order to avoid this a framework for best practices is proposed. This is another neglected area that needs to be addressed with respect to the SSA.

In this chapter three frameworks are proposed to support Software Engineering in the Small. The three frameworks discussed include:

- A proposal for a Software Engineering Environment (SEE)
- A proposal for a Software Process Model
- A proposed Framework for Best Practices.

## 4.2 A Proposal for a Software Engineering Environment (SEE)

One of the many objectives of software engineering as a discipline is the production and support of quality software. By improving the quality of the development process, the quality of the end product is positively influenced. The software industry is gradually witnessing the orientation towards quality and the maturing of software engineering as a discipline. For example, The Malcolm Baldrige National Quality Award3 has "Management of Process Quality" as one of its seven assessment categories. The Capability Maturity Model (CMM), 4-6 produced by the Software Engineering Institute (SEI) at Carnegie Mellon University, defines the maturity levels of an organization based on the quality of the processes it employs (Heineman et. al., 1994). The ability to implement a quality development processes to produce quality software is greatly influenced by the *environment* in which it is produced. Small and Medium sized Organizations will certainly experience a competitive edge if they can deliver optimal quality software under the difficult conditions in which they operate. In this section a SEE with respect to the SSA is proposed.

### 4.2.1 Environmental Influences and Application Characteristics

Software Engineering in the Small is a challenge in the constantly changing IT industry. Constant changes also means a change in the development environment and organizations must adapt to them.

Over the last four decades many changes have occurred in the development environment and in the nature of software applications. For example, twenty years ago, the development environment was mainly mainframe based and tools and methodologies were slow in being introduced. As a result the availability of tools was limited. The most common method (and also tedious) to debug a simple COBOL program was by reading its core dump. In the recent years many applications were scaled down and mainframe development has been reduced, while client/server and desktop development have increased. Most data base applications are now relational. The availability of tools and techniques has increased and most development environments have become highly visual. Techniques such as code re-factoring and design patterns have been introduced. Similarly applications are becoming more specialised, for example, data mining, data warehouse and Web portal development.

Many organizations develop several products with common characteristics among them. Product Line Engineering (PLE) supports this type of development and aims to simplify the effort by exploiting the concepts of standardisation and reusability. For example, Borland's C++ Builder wraps many of Delphi's Visual Components Library (VCL) objects internally. These products have the same look and feel and can share a common SEE. However, this is not always the case. Quark's DMS (Digital Media System) application and its E-Commerce application do not share any common objects. Furthermore, the development approach is different and it is difficult to completely standardize the development environment across the organization.

Software development has also become more complex and diversified. The introduction of component development is not a simple matter of "plug and play". An application that has a high usage of components will also have a high integration effort, namely, the "hand shake" between components must be clean. A SEE for component-based applications is experimental. The activities are component research, evaluation, selection, benchmarking and prototyping. In Web applications, such as E-Commerce and Web Portals, a specialised SEE is required as compared to an Enterprise Resource Planning (ERP) application. For example, in addition to supporting the typical analysis, design and implementation phases, a SEE for Web engineering must also have tools to support document integration into the site and to manage and prepare media for the Web (Powell, 1998).

Modern software development has become increasingly challenging. As a result the SEE that is required today is beyond the selection of a single CASE (Computer Aided Software Engineering) tool. However, a single CASE tool may be suitable for some environments, for example, financial institutions and state departments who still run their legacy systems. Smaller organizations may not find this to be optimal. They may have to *short circuit* the development process in order to gain speed and flexibility. For example, Rational Rose is the prescribed SEE at Quark, however, the developers mainly use it to perform object modelling and the software development cycle did not follow the complete Rational process. Rational's Requsite Pro (a document management system) was replaced by Quark's DMS application, while Visio was used to draw process flow and dataflow diagrams. It might be argued that there were no cross references between the class diagrams and the process flow

diagrams. Actually nobody missed them. As a clarification the author is only implying that the development approach is flexible and the team members did what was necessary to speed up the development. The documents and diagrams were communicated across the teams during multiple sessions and the ability to understand them in various formats was not a problem. Thus it is evident that a skilled team in diverse methodologies and techniques is required.

System changes are not linear and as a result the development environment must be dynamic. A system's transition from one state to another must be smooth and the process must be flexible. Therefore the SEE must support rapid changes and evolve with changes in the real world. In the next section a SEE that exhibits these characteristics is discussed.

### 4.2.2 The Metamorphic Software Engineering Environment

The theory of the Metamorphic Software Engineering Environment (MSEE) (developed by the author) is to constantly adjust the development environment to changes in order to maintain its efficiency and effectiveness. This is achieved by eliminating the agents, which instead of being an aid to solve problems, *becomes* a hindrance. Therefore the environment is constantly transformed over time. The term "agents" is referred to something that is used to aid in the production of software. For example, a modelling language such as Unified Modelling Language (UML), automated software-testing tools and Graphic User Interface (GUI) tools are agents. The terms agents and tools are used interchangeably during the discussion.

Eliminating an agent from the development environment means that the value it provides is no longer optimal and must be replaced. Software development tools are innovative and competitive. What used to be a state of the art is becomes outdated and cumbersome to use when compared to new ones on the market. For example, Micrografix's ABC Flowcharter was popular because flowcharts and models could be drawn with a computer. However, Microsoft's drawing tool (Visio Professional) was considered to be more sophisticated, easier to use and offered additional drawing capabilities. In an organization where the author once worked, ABC Flowcharter was the standard diagramming tool. However, over time it became obsolete and the development team proposed that there should be a change to Visio. Management made the change one year later. Development with the approaches cannot tolerate lengthy bureaucratic delays for a simple decision as the advantages of rapid development is diminished.

Tools are constantly improving and over time new ones are introduced. For example, initially the Object Modelling Technique (OMT) methodology used its own OMTool Editor to model a system. Today, Rational Rose's development environment provides the capability to do object modelling using OMT notation, Booch notation and UML. Hence, the users have a choice of widely accepted modelling techniques. The important question is: *What is the criteria to eliminate an agent?* An agent is eliminated based on the organization's experience and judgement that it is no longer supportive of changes in the development environment. The timing of elimination will vary from organization to organization. Being restricted to fixed toolset SEE in modern software development can be counter-

productive. By reducing environmental restrictions, increasing environmental flexibility and maintaining discipline, the concept of the MSEE ideally supports SMO. Therefore as the situation demands and over time, the SEE will undergo a change of characteristics and conditions. Based on the nature of the application it should be possible to choose the appropriate agents to reach a project's goals. For example, prototyping techniques may be used in one project while Joint Application Development (JAD) techniques in another. Thus the SEE must support a situational development approach. Similarly, de-bugging a procedural language application and an object oriented language application is different. An approach specific debugger will be more useful than a generic one. A modern software development environment must also be flexible and place extra emphasis on automated development procedures and administration tasks. This helps the developers to concentrate on what they do best. Furthermore, systems will always undergo changes and the development environment must provide the means to make them effectively.

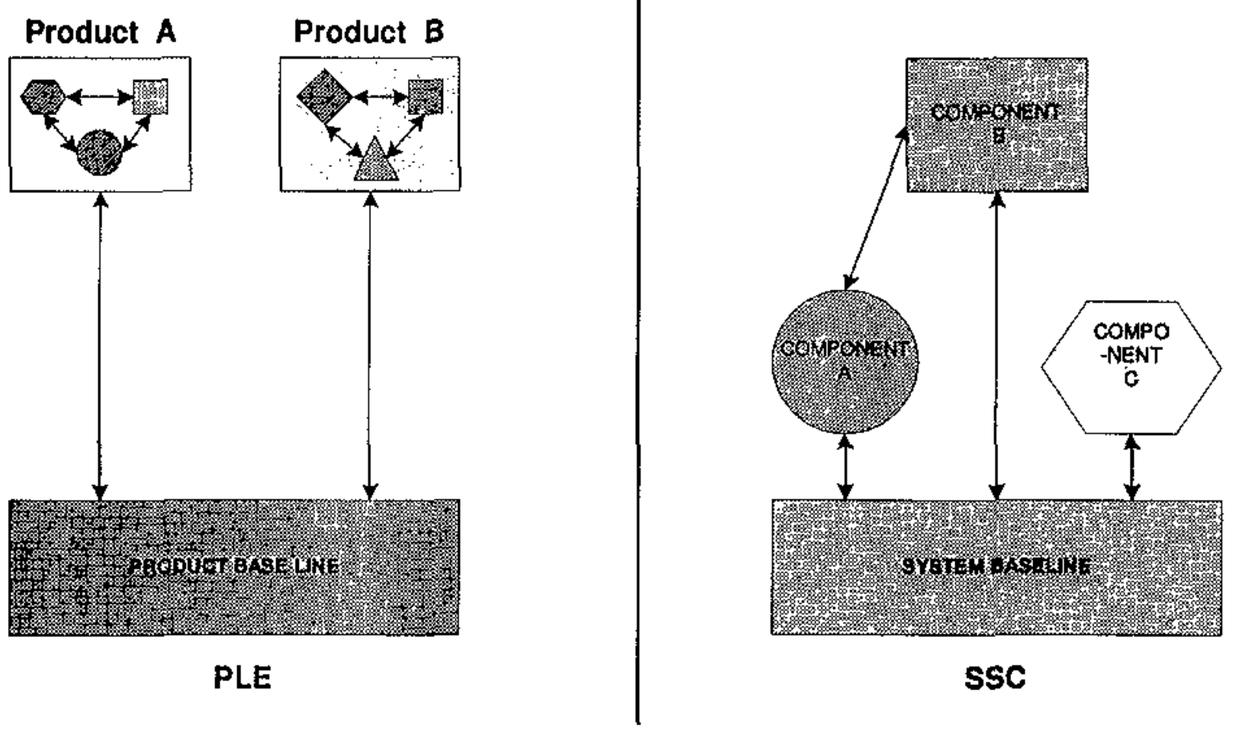


**Figure 4.0:** The application structures of PLE and SSC development.

In Figure 4.0 the simplified application structures of systems development using the PLE and the Synchronize and Stabilize with Components (SSC) approaches are illustrated. Depending on the type of systems that an organization develops, it is possible that both approaches may be used. The PLE approach results in several self contained systems in a product family which runs on top of a common base system. The systems are highly integrated and share common software modules. However, with the SSC approach, apart from the component evaluation and selection tasks, a high percentage of the development task is focused on designing and implementing interfaces between the components and the base system. The SSC approach is of a highly technical nature and mainly involves data exchange and task delegation among components. Therefore the direction of development cannot always be prescribed. During the development the usage of tools, techniques and methods will differ. The SEE must be able to support such diversity and the MSEE is conceptually designed to address this situation.

| Development Types | Description |
|---|---|
| **Conceptual Development** | Explore new business concepts and the application of new technology. Research projects. |
| **New Systems Development** | Development of new applications. |
| **System Enhancement** | Major system enhancement. Namely, adding new functionality, interfaces, tune performance. Adjust to customer's changing demands. |
| **System Maintenance** | Make corrective, preventive, and adaptive changes to the system as required. |
| **System Re-engineering** | Redesign, restructure a part or the whole system. Incorporates backwards and forwards engineering. |

**Table 4.0:** Basic types of systems development.

In Table 4.0 some of the basic types of systems development that an organization will undertake are summarised. They are traditional in nature and most software organizations are bound to have already encountered them. However, there are other related development activities as illustrated in Figure 3.2 (chapter 3). For example, software configuration management, quality assurance and system reviews. Nevertheless the MSEE should provide the essential support for the basic development types and the other related activities.

Furthermore, an organization must rely on its own professional judgement as to how it will architect its version of a MSEE. Most of the choices made would be influenced by many factors. In the next section an example of implementing a MSEE is examined.

### 4.2.3 Implementing a Metamorphic Software Engineering Environment

Since organizations are unique and have their own goals and objectives, their development environments are also influenced accordingly. Therefore the appropriate agents must be selected because software engineering is complex and the complete process must be optimally supported. The following basic key activities are considered necessary in order to establish a MSEE:

- Strategic Management Decision
- Team Building
- Identifying the Development Activities
- Selection and Evaluation Process
- Continuous Monitoring.

This list is not exhaustive and depending on the organization other key activities may be necessary. These points are briefly discussed below.

#### 4.2.3.1 Strategic Management Decision

The type of development that an organization will undertake is a strategic management decision that

must be carefully planned. Some visionary planning is required with regard to the type of software projects it will undertake presently and in the future. Its development goals and objectives must be well defined and verified whether they will be achieved in the development environment. The following factors may influence management's decision:

- Its line of business.
- Its market share and competition.
- The volatility of the business environment.
- The size of the organization.
- The availability of finance.

### 4.2.3.2 Team Building

Once the strategy for a development environment has been decided, the actual task of creating it must be set in motion. The ability to select tools, techniques and methodologies requires specialized skills and management must select the right team for this purpose. An appointed team should have the following skills:

- Ability to conduct research
- Good bench marking and evaluation techniques
- Good technical skills
- Be objective and have sound judgement.

### 4.2.3.3 Identifying the Development Activities

| Development Activities | Team Roles | Example of Tools |
|---|---|---|
| **Environment Scanning, Scope Competitor Analysis** | Product Management | Market Database, Customer database, Ad-hoc information, statistic analysers, |
| **Planning** | Project Management | Project management software |
| **Requirements Definition** | Product Management, Users, Analysts, Domain Experts | Methodology, e.g. OMT or UML CASE Tool |
| **Systems Analysis and Design** | Analysts, Domain Experts, Systems Architects | Methodology, e.g. OMT or UML CASE Tool, |
| **Technical Design** | Software Engineers, Software Architects | CASE Tool, Software specifications tools |
| **Implementation and Test Build and Test (integration and test)** | Software Engineers, Developers Users | Development environment e.g. Delphi, Powerbuilder. repository manager, test data generator, testing tools |
| **Maintenance** | Software Engineers, Analysts, Product Management, Domain Experts | Methodology, e.g. OMT or UML CASE Tools |
| **Quality Assurance** | Quality Assurance Team | Test data generator, Automated test tool, Bug tracking software |
| **Software Configuration Management Release Management** | SCM Team (Technical Analysts and Software Engineers | Build generation tools, Version control tools |
| **Documentation** | Whole Team, Technical Writer | Word processor, Document manager |

**Table 4.1:** Typical development activities.

In order to implement a MSEE it is important to identify the basic development activities. The

identification of these activities will indicate the tools techniques and methodologies that are necessary for a MSEE. With reference to Figure 3.2 (chapter 3) it is simple to identify some these typical system development activities. In Table 4.1 these activities are summarised together with the team roles and examples of development tools. The table is a generalization and individual organizations will have specific activities, roles and tools. A development team may consist of a database administrator, GUI developer, and business-logic manager. Each team member utilizes different tools in the development environments, and will share information such as database table information, user-interface components, and business-logic rules (Shimmin, 1995).

### 4.2.3.4 Selection and Evaluation Process

The technical team must scan the market for appropriate tools, techniques and methodologies. The tools, which are selected and evaluated, must be in sync with management's strategic decisions. The nature of this task is exploratory and constant experimentation is undertaken before a final selection is made. Table 4.1 also indicates that the development environment must support the tasks of a variety of people. Thus the selection and evaluation process is not trivial and the team must have a good overall understanding of the organization's scope. O'Brien (1995) provides some valuable hints on tool selection in his article.

### 4.2.3.5 Continuous Monitoring

Once the environment is defined the team must continuously monitor its effectiveness. Any sub-optimal agents must be replaced. Furthermore, the market environment must be constantly scanned for new tools and techniques with regard to their usefulness in the development environment. As a result of continuous monitoring the MSEE will undergo changes in its characteristics and conditions.

The basic key activities discussed serve as general guidelines for the effort that is required to establish a MSEE. In the next section a comprehensive example of a MSEE is discussed.

### 4.2.3.6 A MSEE Example

As previously noted, the MSEE is designed to cope with changes. The degree of changes or the desire to change will depend on an organization. In this section, an example of a MSEE is discussed from a practical point of view. Based on the author's experience, the entire SEE will not change, but parts of it will. Therefore a MSEE will have a *base environment* that will remain fairly stable. Generally it consists of a methodology that has been adopted. It is possible that an organization may adopt more than one methodology. For example, at 3M Deutschland GmbH (the author consulted there) structured analysis and design was used as a standard. However, the Jackson System Development (JSD) methodology was also used because the earlier systems which were developed with it were still in operation. Normally the CASE tool used supports the underlying methodology. Should an organization decide to undertake object-oriented development, then a CASE tool that supports this paradigm may also be introduced into the *base environment*. It can be seen that nowadays the software development is diverse and the MSEE allows itself to be tailored to support this. Although a CASE environment

may incorporate a range of tools, the author had observed that organizations generally find that they are not always suitable and hardly use them. They tend to use the methodology mainly for system modelling. A full evaluation of CASE environments is a specialised topic and is beyond the scope of the discussion.

The MSEE will also have a *bucket* of tools known as the *variable environment.* In this environment most changes will occur and within a short time frame. The team mentioned in the previous section will spend most of their efforts in keeping the *variable environment* in a state of the art condition. As an example, a typical environment will include agents such as:

- Integrated Development Environment (IDE), for example, Delphi, Visual Studio 6, PowerBuilder, Oracle 2000
- Assemblers
- Code editor, for example, Codewright
- Object browser
- Debugger and code profiler, for example, Nu Mega bounds checker
- Version control software, for example, Visual Source Safe
- File Conversion Tools
- Bug tracking software, for example, Radar
- Test data generators and Automated Tester
- Graphic Tools
- Web Design Tools, for example Hot Metal, Web Sphere
- Report Writers, for example, Crystal Reports.

The above list of development tools and utilities is endless. However, it should provide a clear idea of the *variable environment* of a MSEE. As a matter of fact, Web design can be a unique development environment of its own. An organization may also develop its own tools. An engineer at Quark developed a System Stresser tool that progressively stress tests a system and examines its behaviour. With this regard, memory consumption, bottlenecks and optimisation problems are resolved. Similarly, Quark uses its own content management system, DMS, to manage all system documentation.

The *variable environment* tends to have a greater number of users than the *base environment.* Thus it should be the state of the art, but practical, simple to use and extensible. An environment with these qualities would contribute to speed and quality during the development. Although the *variable environment* may appear to be a collection of ad-hoc tools and utilities, it is not really the case. By modern standards most tools integrate well because they have open interfaces. However, they must be well researched, evaluated and selected. It should also be remembered that a change in an *agent* might depend on another *agent.* For example, the Extended Hyper Text Markup Language (XHTML) was intended to replace the Hyper Text Markup Language (HTML). However, upgrading a Web application from version XHTML 1.0 to XHTML 1.1 is not simple because the Internet Explorer 5.5 does not support the full set of XHTML 1.1 features (Deitel, Deitel and Nieto, 2002). Therefore the

application will not function correctly until future versions of the Explorer provide this support.

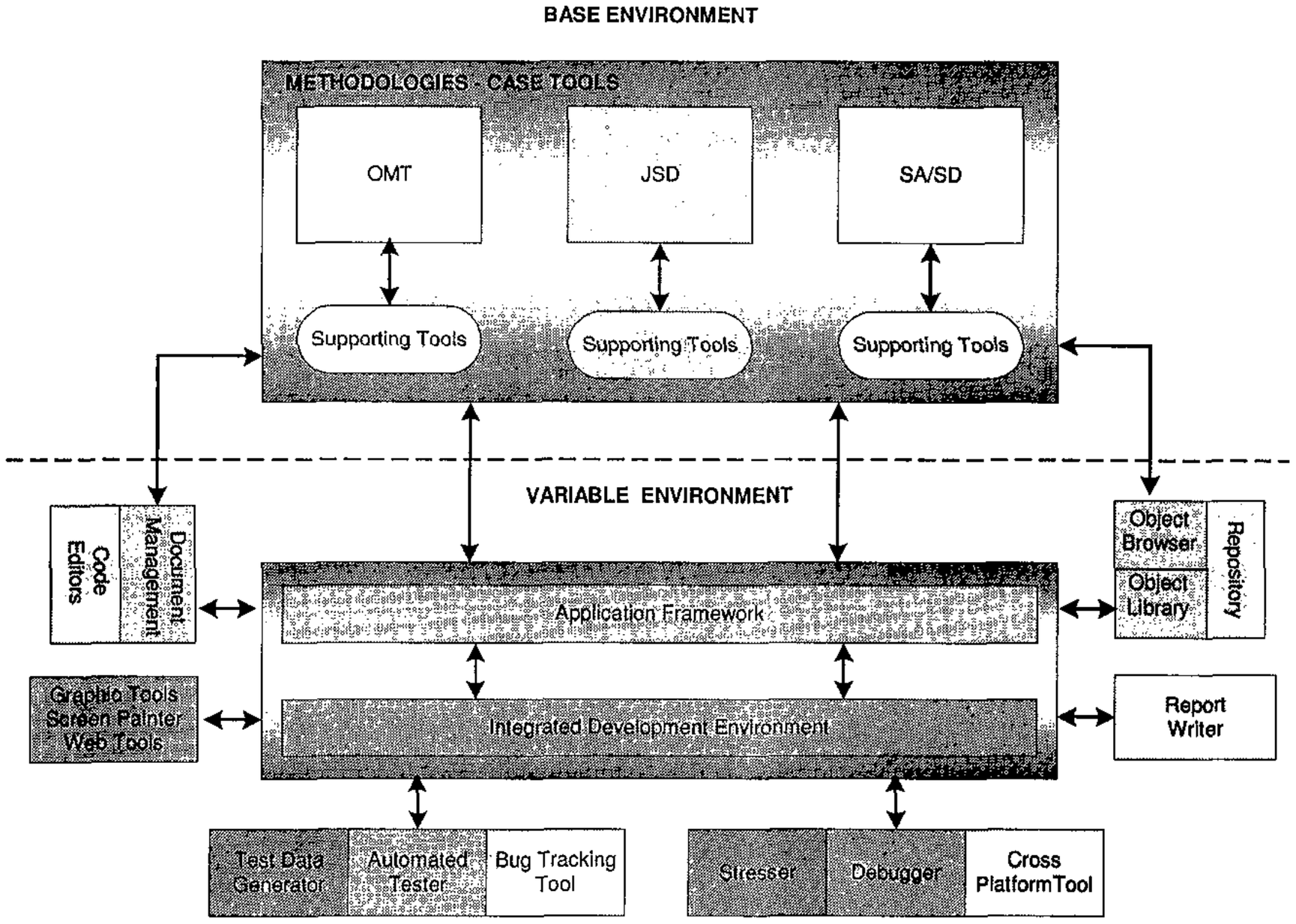


**Figure 4.1:** A model of the MSEE.

In Figure 4.1, a model of the MSEE is illustrated with its *base and variable environments*. It is a general representation (and simplified) as there can be many variations of it. The boundary between the environments is blurred and there is much communication between them. The arrows indicate the communication paths. Depending on the integration capability of the tools the communication may be manual or automated.

Quark recently switched to the .NET technology without any major interruptions in its development process. The *application framework* for building client/server applications constantly changes as new modules are added or removed. Similarly, the object library is also constantly updated. During the previous years a number of Web design tools were also added to the *bucket* and were used as required. However, over the past five years the Rational development environment was *mainly* used for object modelling and design. The usage of UML and Use Cases had increased. Therefore over time Quark's *variable environment* had undergone the many changes to remain highly productive.

In some cases the *base environment* can be comprehensive. The CASE tool such as Information Engineering Facility (IEF) has five toolsets, namely, planning toolset, analysis toolset, design toolset, construction toolset and implementation toolset (Avison and Fitzgerald, 1995). As a result it can also

possess the capability to perform tasks of the *variable environment* because of the overlap in functionality. Ultimately the concepts of the *base and variable environments* of the MSEE aims to create a robust and flexible development environment that is well equipped to face software engineering challenges. This concept also blends well with the philosophy of the SSA.

The MSEE does have its advantages and disadvantages and are discussed in the following sections.

### 4.2.4 Advantages of the MSEE

The MSEE has many important advantages for an organization that adopts the SSA. The following advantages are discussed:

- Promotes a Technically Up to Date Environment
- Improved Productivity
- Improved Quality
- Provides Choice and Flexibility.

#### 4.2.4.1 Promotes a Technically Up to Date Environment

Since the MSEE progressively changes over time, an organization is able to maintain a technically up to date development environment. If the tools, methods and techniques are properly evaluated and selected, then it is possible to achieve a modern, effective and efficient development environment. A technically updated environment is a motivational factor because the development staff values the importance to work with the latest technology. Being technically up to date is an important evaluation criteria for a system's success. For example, Quark adopted the industry standard Extensible Markup Language (XML) format into its development environment. This made it possible for QuarkXPress users to free their content data with the product Avenue.quark for reuse. Avenue.quark leverages the rich pool of content data in QuarkXPress documents for re-use in multiple formats and delivery media. Thus media independent publishing is supported.

#### 4.2.4.2 Improved Productivity

Over time tools, techniques and methods also improve and have become more innovative. The concept of the MSEE allows an organization to exploit these properties. With new and improved agents being added to the development environment progressively, much of the work can be simplified and automated. As a result an increase in productivity and an optimal use of time is influenced. A simple drawing tool such as Visio Professional allows a developer to create diagrams effortlessly as compared to similar tools of the past five to ten years. Similarly, an automatic software-testing tool can also simplify and speed up the testing process.

#### 4.2.4.3 Improved Quality

By adjusting and re-evaluating the development environment progressively improves its quality. A quality development environment will contribute to the quality of the development process and system.

An appropriate CASE environment can assist in improving the analysis and formalising the process thus improving the front-end development activities. Nowadays it is insufficient to rely only on the debugger supplied with an IDE. With additional code analysis tools, faults such as dangling pointers and memory leaks in the code can be detected and corrected early. As a result the quality of the system is improved.

#### 4.2.4.4 Provides Choice and Flexibility

The MSEE encourages an open development environment since nowadays tools have open interfaces and tend to integrate well. Although integration problems still exist, there has been a great amount of improvement in this area over the past years. Instead of being restricted to a "fixed" environment the MSEE evolves and provides greater flexibility. Therefore the development team has a choice of agents and can exercise their preference to meet their objectives. Generally people are more efficient when they are not completely restricted.

### 4.2.5 Disadvantages of the MSEE

Although there are advantages with the MSEE, there are also some difficulties and it is important to realize them. The following are some of the difficulties with this environment:

- Costly to Maintain
- Requires Skilled Personnel
- Chasing the Latest Flavours
- Problem to Cope with Changes.

#### 4.2.5.1 Costly to Maintain

A MSEE can become costly to maintain, because it must be constantly updated. It is common for software development tools to be outdated within a few months. This aspect can have an impact on organizations with a very limited budget. A possible solution is to opt for simpler tools as practised by the Wisdom approach. They are cheaper and high training costs can be avoided.

#### 4.2.5.2 Requires Skilled Personnel

To maintain a MSEE in an organization is a specialised task. It requires someone with good technical skills who can understand the development environment in its entirety. Well-trained personnel are not easy to find and often recruitment is expensive.

#### 4.2.5.3 Chasing the Latest Flavours

Through naïve carelessness, possibly driven by well-intentioned enthusiasm, the selection of tools may end up being a chase for the latest flavour. This can be caused by trying to satisfy an ego or a misguided attempt to increase organizational maturity. Therefore management must be committed and proper research must be conducted to avoid this pitfall.

### 4.2.5.4 Problem to cope with changes

Since the environment constantly changes, the developers may encounter difficulties in coping with them. This is especially true if the differences between the old and new versions of the tools are vast. Furthermore, the new version must be learnt and can be time consuming. If possible, changes should be introduced during quiet periods, as a learning curve may be required.

The benefits of the MSEE outweigh the disadvantages. Through good management the negative effects may be reduced or overcome. However, the scope of implementing a MSEE is organization specific and will vary. Software development consists of a number of inter-dependant and complicated activities. In order for the development to proceed in an orderly manner a software process model that is suitable for the SSA is necessary. In the next section this software process model is discussed.

## 4.3 A Proposal for a Software Process Model

Organizations which are confronted with the Software Engineering in the Small problems develop smaller, but complex systems with rapidly evolving requirements and operate in an unpredictable environment. Apart from an appropriate SEE, an appropriate SPM is also necessary as it has a direct influence on the quality of the system. The general nature of the development with the SSA is a rapid system evolution with concurrent activities. The SPM must be flexible and adaptable to cope with these conditions. As illustrated in Figure 4.0, the life cycle of each system can be different and the SPM should support these differences.

In chapter 2 the following concerns with respect to the SSA were identified:

- Rapid changes in the environment and software requirements
- Development is against time
- Good co-ordination and control is required
- Cope with system diversity
- Remain competitive
- Good communication
- Manage complexity
- Good resource management (financial and human)
- Use uncomplicated standards and apply good SE principles
- Good systems architecture and high quality software.

The above are real world concerns and the development process must be capable of addressing them. During the development of a system many *questions* are posed around the problem statement. The following are some examples of them:

- What must be developed?
- Why is it required?
- How will it be developed?
- Who will use it?
- When will it be available?
- Who will pay for it?

These important key questions have a strong impact on Software Engineering in the Small. Actually they should be answered clearly before commencing any software development project. Comaford (1995) examines nine interesting *problems* of software development and provides suggestions to avoid them. Out of the nine problems, the following are of particular interest:

- Better understanding of the business
- Better understanding the users
- Better management of the development staff
- Start testing.

The questions and problems are posed from different perspectives but are related. The SPM should address them by guiding the development process in order to provide solutions. The objective of the SPM is to support the development of quality software through a quality process. Since the approaches exploit the concepts of incremental and iterative development, the SPM should also embody these concepts. With these aspects to consider and with regard to the SSA, a SPM is proposed in the next section.

### 4.3.1 An Iterated Meta Process Model

Software aims to solve real world problems. Before software can be engineered, the 'system' in which it resides must be understood. To accomplish this, the overall objectives of it must be determined. Aspects such as the role of hardware, software, people, technology, procedures, and other system elements must be identified. The operational requirements must be elicited, analysed, specified, modelled, validated, and managed. It is these activities and relationships, which form the foundation of system engineering. This is a *holistic* view of a system and implies *synergism* (Powers, Cheney and Crow, 1990). Information systems do not initiate from a vacuum. Factors such as the organization and general infrastructure will influence the target system. *The SPM model should reflect reality by modelling real world development processes to provide real world solutions.*

The proposed SPM is referred to as the Iterated Meta Process Model (IMPM) and has been developed by the author. The concept of the IMPM is based on exploiting the use of models which are assigned distinct roles. The models represent the problem statement visually at various levels of abstraction and focuses on addressing the concerns and questions mentioned in the previous section. The outputs of the models aggregate to create a *holistic* view of the problem statement.

In a turbulent development environment it is difficult to immediately obtain full knowledge to solve the problem statement. However, the IMPM *forces* management, analysts and designers to focus their attention on the key aspects of systems development by *formally* including them as an imperative part of the software development process, for example, organizational needs, system ownership and human roles. In this manner knowledge that is required to develop the target system may gradually extracted and documented.

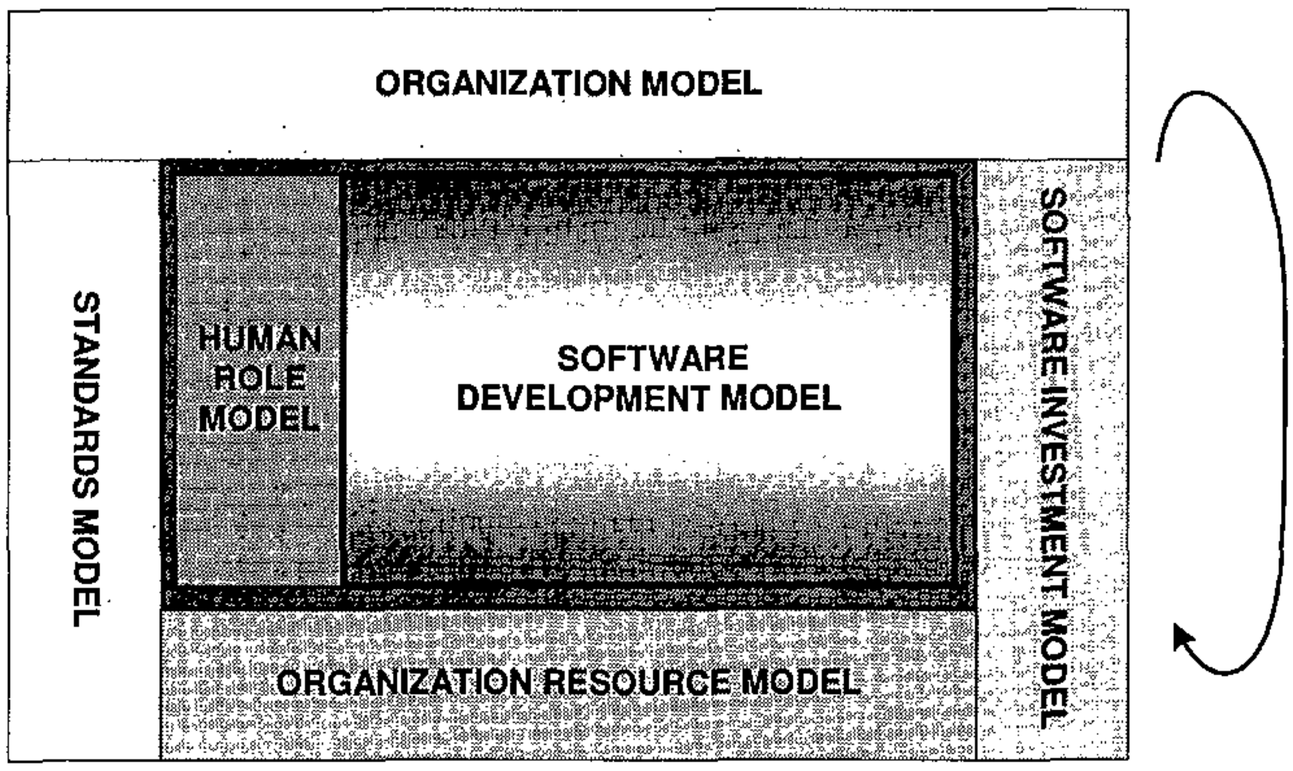


**Figure 4.2:** The Iterated Meta Process Model (IMPM).

In Figure 4.2 the IMPM is illustrated and is comprised of the following six key imodels:

- Organization Model
- Standards Model
- Human Role Model
- Software Investment Model
- Organization Resource Model
- Software Development Model.

Each model has it own goals and objectives and its own set of processes to achieve them. It is tempting to consider one model to be more important over an other, but it would be unwise to do that because they influence each other. With regard to the key role they play in the software development process a *holistic* and *synergistic* perspective is essential. However, in order to simplify their understanding it would make sense to study them independently. What follows is a brief description of the role of the models.

The *organization model* relates to the business analysis school of thought and focuses on *what* must be developed. Its objective is to view the organization with respect to its business architecture and the environments (micro and macro) in which it operates. By modelling the organization it its possible to identify the data, information and systems that it requires to support its business. In other words, is the correct system being built for the organization to support its objectives and further its aims?

The *standards model* is the organization's mechanism to create and maintain order in its functioning. The model helps to define guidelines and procedures for its diverse business functions across the organization. Standards help to improve the channel of communication in the organization and ensure that there is consistency and uniformity in doing things. It must be ensured that they do not conflict across business functions. With respect to software engineering principles the standards model

supports *formalism* during the development.

It is the *human role model* that really gives an organization its life and motion. The people that it employs are the cogs that drive the organization forward either into prosperity or backwards into doom. The *human role model* represents the total human resource pool. It defines their objectives within the framework of their co-operating roles, which must aggregate to meet the organization's objectives. These objectives must not conflict with each other. Humans require a varying degree of information to make decisions and must have the appropriate information systems to provide them this. Furthermore the *human role model* also specifically defines the human resource pool of expertise that is required to realize software projects. The Human Computer Interface school of thought influences this model with regard to the usability of information systems.

Software costs money and the *software investment model* represents the various cost parameters of software development. It identifies the owners and the stakeholders of the system and supports them to make decisions as to whether a project is feasible. It addresses issues such as the development budget, return on investment and value received. Therefore it supports the management of systems development costs.

The *organization resource model* defines the technical resource infrastructure that is required to support the software development process. These include hardware, software tools, techniques, network structure and the development environments (local and global). The model also defines the management policies of the infrastructure.

The *software development model* focuses on *how* the problem statement must be solved to meet the business objectives and the user's needs. It examines possible solutions to the problem statement. The five models which were already described above strongly influence the scope of this model. Which in turn influences the final solution to the problem statement. This model is also influenced by the software engineering school of thought, namely, *is the system being built right and is the right system being built (that is validation and verification).*

The following examples summarises the practical significance of the models:

- The *organization model* determines *what* the system must do to support its business.
- The *standards model* will guide the development with the essential development standards.
- The system must be developed from the user's perspective and is influenced by the *human role model.* For example, a high degree of usability is required.
- The solution to the problem statement will certainly be influenced by the budget. Therefore the *software investment model* plays a major role in allocating financial resources.
- Depending on the type of solution sought, the demand on *organization resource model* will vary. For example, if a Web-based solution is required, then additional hardware and software such as Web servers and firewall are required. Hence the infrastructure changes.
- The *software development model* focuses on implementing the solution.

It can be seen that the IMPM provides a useful framework to define the development policies which must be adhered to.

### 4.3.2 Applying the Iterated Meta Process Model

Although each model has it own goals and objectives they work co-operatively towards a common goal namely, to solve the problem statement. During the development, the analyst will examine the problem statement and *pose vital questions* around the target system. Each model will be used in conjunction with another to provide answers. The common systems analysis and design question is the *what* and *how* of the development process.

| General Questions | Org. Model | Standards Model | Human Role Model | Investment Model | Resource Model | Dev. Model |
|---|---|---|---|---|---|---|
| What is the problem? | X | | X | | | |
| Is it feasible, Make or buy? | X | | | X | | |
| Stakeholders and owner? | X | | X | | | |
| Who are the users? | X | | X | | | |
| Development guidelines? | | X | | | | |
| Skills and Expertise? | | | X | X | | |
| What technology is required? | | X | | X | X | X |
| How to Implement? | | X | X | | X | X |

**Table 4.2:** Examples of basic development questions in relationship to the IMPM.

In Table 4.2, the first column indicates examples of basic questions (stated in point form) which may be asked during development. The columns two to seven indicates the models of the IMPM which are needed to answer them. For example, a question on what technology to use may be addressed through by the standards model, software investment model, organization resource model and software development model. The standards model is relevant because an organization normally defines its development standards to be followed. As an example one of Quarks technical standards is to use Microsoft's technology and the development team must adhere to it. The resource model is used to determine whether the technology already exists or whether it must be procured. If it must be procured, then for obvious reasons it will have an impact on the software investment model. The software development model will provide a final specification of the technology that is required once the technical analysis has been completed.

In the above example it is clear that the models of the IMPM collaborate to address fundamental questions around the problem statement. As a result the analysis is improved by a multi-dimensional analysis of the problem statement, which helps to improve planning, reduce risks and cope with uncertainty. The common problems that are listed in Table 2.2 (chapter 2, section 2.6.2) can also be

addressed by the IMPM.

The models of the IMPM are common to most types of software development and form a process framework that supports the analysis, design and implementation activities. The specific procedures and workflows for each model are defined as required by an organization. For example, the organization model will define the integrated business processes and external interfaces. Thus development process can be tailored and made flexible, which is a goal of the SSA. It is essential that the development process occur within the framework of the IMPM because the inter-dependence of the models is the control mechanism. Therefore the meta-model approach of the IMPM encourages a *holistic* development process and contributes towards a re-formulated software development strategy for Software Engineering in the Small.

Since the models are goals oriented, each will produce specific output that will be used during the development. For example, the standards model will provide an appropriate development standards document. The organization model will provide a clear definition of the problem statement, whereas the human role model will indicate their system requirements. The human role model has an important function in identifying the *creator* and the *owner* of the output produced. The stakeholder and the project manager may *create* the development budget together, but the project manager will be the *owner* of it and must ensure that the development is completed within it. The creator and owner can be the same person. Since the SSA is incremental and iterative, the output of the models will also be incremental and are aggregated to form the *blueprint* for the development process. The documents (output) would be accumulated for each increment and collectively form the complete system documentation at the end of the development. Any changes to them must be made through the respective models.

The concept of a creator and an owner of output are useful to enforce commitment and accountability of the development team. The output is a visible result of work completed and can be used to evaluate the ability of the team. That is assessing the reliability of individuals who are capable of getting the job done. Software development with the SSA depends highly on the commitment, accountability, task visibility and reliability of the development team.

Many changes occur from one increment to the next. For example, the standards might change or new technology must be adopted. The iterative property of the IMPM allows changes to be incorporated by iterating them during each incremental cycle. The new increment of the system will be developed according to the updated state of the IMPM. In this manner the development will always apply the latest processes. However, management must ensure that the IMPM is updated.

The IMPM aims to maintain formalism and good software development principles with reduced bureaucracy and delays. The processes are simple and easy to adopt and fits in well with the SSA development concepts. The IMPM may be extended to include other models if required. For example, *a project management model* can be added.

A careful observation will indicate that the first five models of IMPM discussed above, lay the foundation for the development. These tasks are executed in parallel; however, their output must converge. The activities supported by *software development model* cannot commence without this foundation. Thus the IMPM provides a control to check that the development takes place on a firm foundation.

In Figure 4.3 the *software development model* is illustrated in detail and is based on Figure 3.2 (chapter 3, section3.10). A closer study shows that it represents the "software manufacturing centre" of the IMPM. The organization's mission is the key catalyst to the software development model. It gives the "order" to the "software manufacturing centre". The model's activities are iterative and are divided into three areas of responsibility; however, an organization is free to make its own divisions. The flows of tasks within the activities are also dependent on the organization. The diagram illustrates the deliverables (output) which are produced at each stage. The deliverables are used for control and verification purposes and provide visibility of what has been achieved. An important deliverable (not shown in the figure) after an increment has been completed is the *post-mortem* analysis document. It records, mistakes made, suggestions for improvements and problems which were encountered. This document is used as input for future improvement during development.

The iterative nature of the model supports the rapid reaction that is required by the SSA, as adjustments are possible at numerous points during the development of an increment. For example, the changes may be required in the Beta release and must be programmed by the development team. The feedback loop from the release management activities to the system development activities indicates this backward flow. The *feature trimming* activity is important as it manages the omission of features in order to meet the schedule. The omission of features affects the scope of an increment and at Quark omissions are managed through a formal change control process.

With reference to the MSEE in Figure 4.1, the tasks of design and the creation of the Use Cases can be associated with the *base environment*. Other tasks like coding, testing and build generation are related to the *variable environment*. Carmel and Becker (1995), identifies the following special needs which characterises a packaged software process:

- Addressing multiple user types
- Product differentiation
- Identify remote customer
- Involving remote customer
- Speed in development
- Create market interface
- Develop in iterative mode
- Deliver near defect-free release.

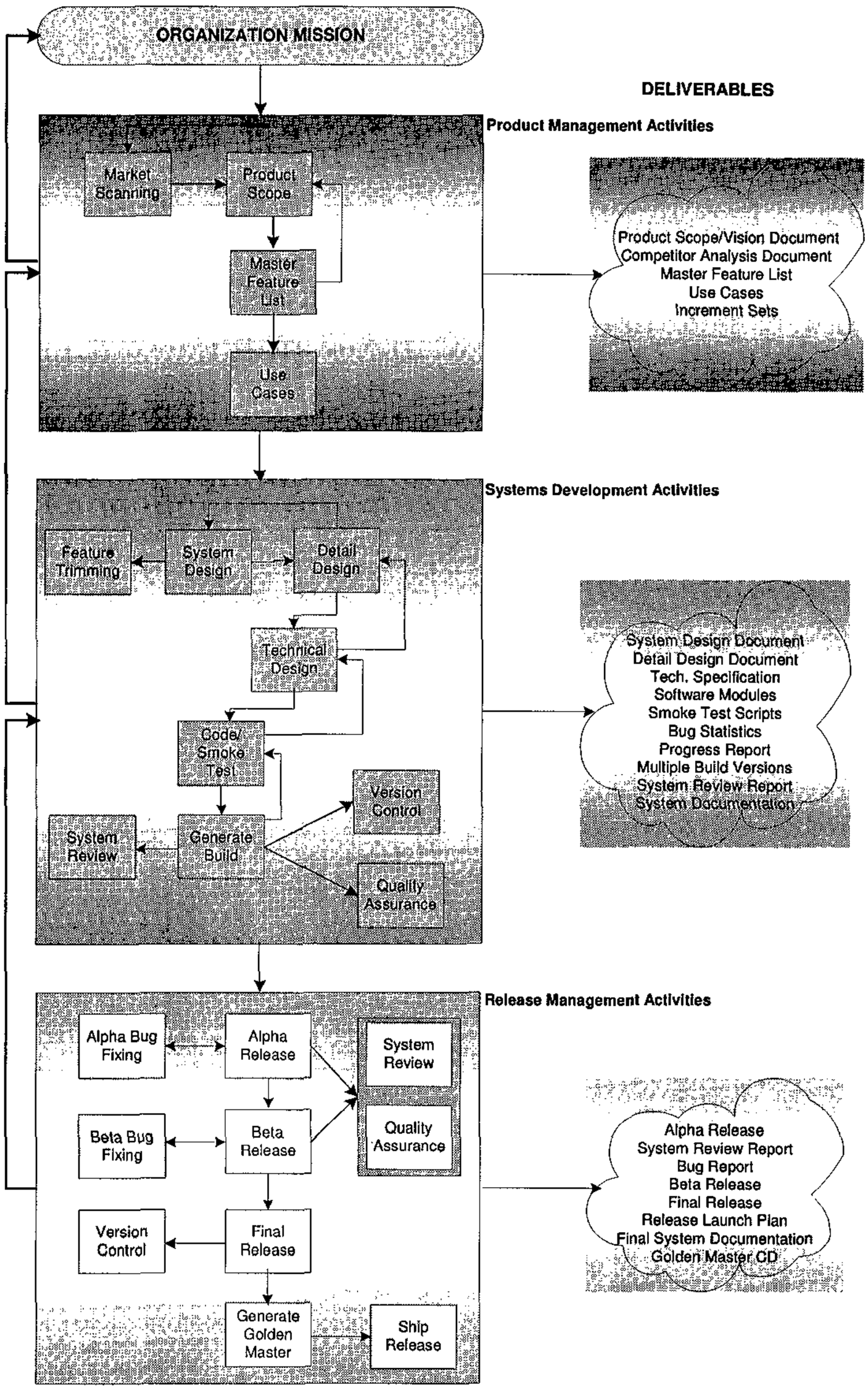


**Figure 4.3:** A detail view of the Software Development Model.

The IMPM supports packaged software development and the workflow of the *software development model* is illustrated in Figure 4.3 and does map effectively to the above process. Quark develops packaged software and follows similar development processes. In the case of a in-house development

environment the workflow of the software development model can be re-structured. The product management and release management activities are not required and may be replaced with a *version migration procedure.*

### 4.3.3 Motivation for the Iterated Meta Process Model

The motivation for the IMPM hinges on the fact that it is supported by sound theoretical concepts and aims to solve real world problems. In this section the following aspects are considered to be influential for the motivation of the IMPM:

- Fundamental System Concepts
- Churchman's Theory
- Supports General System Architecture
- Views Real World Problems.

#### 4.3.3.1 Fundamental System Concepts

An interesting property of the IMPM is that it supports *fundamental system concepts.* Ahituv and Neumann (1990) identifies the following eight basic characteristics of a system:

- Goals and purposes
- Inputs
- Outputs
- Boundaries and environments
- Components and interrelations
- Constraints.

The individual models of the IMPM have clearly defined goals and objectives, which aggregate to construct the target system. The clear defined goals and objectives of the models, simplifies the analysis and design tasks because each model has a precise role during the development. They accept specific inputs and produce specific outputs to facilitate system construction.

Similar to a system, the IMPM has its own boundaries and environments, that is the development framework is within the model's boundaries. The models and their interactions may be compared to the components and their interrelations of a system. The models behave like specialised *black boxes* whose concept is to reduce the overall complexity of the system construction process. Since an information system possess these characteristics, the process model that is used to build it should also contain these characteristics in order to uniformly support this type of thinking. Thus the characteristics of the IMPM help to maintain this thread of thought.

#### 4.3.3.2 Churchman's Theory

According to Churchman's theory (Churchman,1971) an entity must meet nine conditions in order to be called a system. Although the theory dates back over three decades, the system fundamentals it

The concept of *business architecture* is to define a business strategy based on a strategic vision (current and future capabilities) as defined by *strategic capabilities architecture*. The business strategy is defined in terms of goals and objectives (short-term, medium-term and long-term), technological environment, and external environment.

The business strategy is translated through a process of information systems planning. The information needs, which must be provided by systems to meet the business objectives, are identified. The *information architecture* includes application level aspects, for example, sales and accounting. Any constraints which may be imposed by the business strategy are considered.

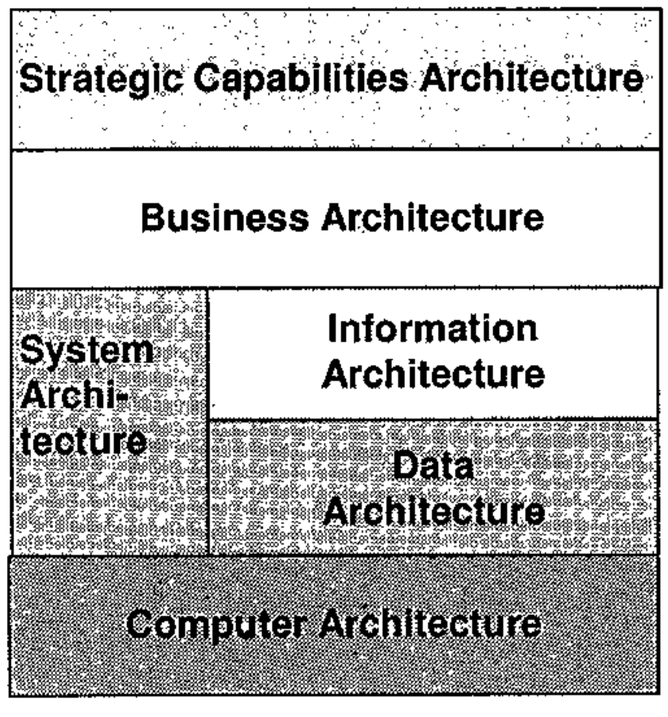


**Figure 4.4:** The general concepts of system architecture.
*Source*: Adapted from: Quark's Business Architecture Document.

*Data architecture* defines the data which will serve the information needs. It defines the current and future needs of data, technology and data movement across system boundaries.

*System architecture* relates to *information architecture*, *data architecture* and *computer architecture*. It involves specific systems, which must be deployed with regard to business applications, data requirements and hard- and software needs.

*Computer architecture* is comprised mainly of specific hardware and software, which will form the technological base for the above architectures. Therefore the choices and decisions made will be influenced by the above *and* also influence the above.

During an incremental development process the system's architecture will evolve. Thus it must be developed on a sound architectural framework that can satisfy general architect requirements. The IMPM aids in meeting this objective during development. For example, the *organization model* relates to the *strategic capabilities architecture* as they both address common problems. Since there is a close relationship between them, the IMPM contributes to analysing architectural requirements *naturally* because the models drive it. As a result stable architectural requirements are possible. The

output of the models can also be used to generate and verify general architecture requirements.

| General System Architecture | Iterated Meta Process Model |
|---|---|
| **Strategic Capabilities Architecture** | Organization model |
| **Business Architecture** | Organization model, Human Role Model |
| **Information Architecture** | Organization model, Human Role Model, Standards model |
| **Data Architecture** | Organization model, Human Role Model, Standards model |
| **System Architecture** | Software development model, Standards model |
| **Computer Architecture** | Organization resource model |

**Table 4.4:** The relationship between general system architecture and the IMPM.

Since the IMPM is iterative and complements the general architecture concepts, the development team can constantly review the architecture through a *natural* iterative development process. The relationship between general system architecture and the IMPM respective models are summarized in Table 4.4.

### 4.3.3.4 Views Real World Problems

In section 4.3 numerous real world concerns, questions and problems were identified and are widely experienced by many SMO. The models of the IMPM contribute to a real worldview development process. In other words it enforces the asking of questions about the real world actors of the system. For example, the organisation model will help the developers to better understand the business and identify what must be developed. The human role model helps to understand the users and their needs for the system.

The IMPM addresses the complete development life cycle. Its workflows can be tailored as required and new models may be added. As a result it is flexible and suitable for Software Engineering in the Small. It is also based on fundamental software engineering principles, which are easy to apply in order to develop quality software.

Although there are many suggested best practices for software development, these suggestions are always subjective. A framework for best practices is considered to be more useful and is the topic of the next section.

## 4.4 A Proposed Framework for Best Practices

The notion of Best Practices (BP) is becoming increasingly popular. Unfortunately, in order to increase their sales, many software vendors claim that their software supports the industry's BP. This would also appear to be a logical strategy to accelerate an organization's performance (Petersen, 1999). It is quite common to hear developers and managers debating over the latest BP, which are eventually reduced to buzzwords. Some job vacancies require candidates to implement BP in an organization. In this situation the potential for conflict has already been created because a candidate's experience of BP may differ from the organization's point of view. This would imply that best practices are generic and one size fits all. *This is not the case.*

McConnell (1996), *recommends* the spiral life-cycle model as one of the best practices for rapid software development. However, McConnell (2000) made the following criticism about Boehm's spiral life-cycle, "*....the model is so complicated that only experts can use it. I'd give it the prize for the most complicated and least understood software engineering diagram of the 20th century.*" The criticism is strong and implies that the spiral life-cycle should be *avoided*. The author is of the opinion that the suggested BP is confusing and contradictory. In one instance the spiral life-cycle is recommended as a BP, while in another it is cited to be very complex and least understood.

Best practices are a desirable concept. Unfortunately many software practitioners do not study them in depth and in context with their objectives. They are viewed according to the success that has been experienced by others. However, the situations in which others have applied a specific BP may be quite different. To consider a best practice to be generic may not yield its optimal value. A drawback with BP data is that it is seldom cross-domain or longitudinal. Few professionals compile them in context with a specific problem and its solution (Andriole, 1995). Thus the process is a combination of experimental, creative and problem solving activities with a subjective view. What may be good for the creator of a BP may not be appropriate for someone else. There is no unanimous opinion on what constitutes BP, and thus there is no unanimous opinion on how to choose them (Fenton, 1996). Therefore caution must be exercised when adopting them. The important question is: W*ho knows best?*

There are also no absolute definitions for the following concepts:

- Best Practice
- Basic Practice
- Good Practice
- Best Influence
- Recommended Practice
- Current Practice or Trends.

The distinctions are blurred and their definitions overlap. However, they do have a common intention, namely, to communicate to the software community improved ways of doing things. Opinions and

experiences from many independent sources are important because they encourage debate and innovative thinking. Organisations should certainly be cognisant of them and *evaluate* the relevancy to their own environment. Comaford (1995), emphasizes that best practices are required to overcome software development problems and avoid repeating mistakes. She discusses several software problems and recommends general BP to solve them. However, it is more important that organizations are able to define their *own* BP because it will influence the overall productivity of the environment and the quality of the system.

Sometimes it is hard to determine where myths end and where reality begins; the same applies to BP. As a result it is difficult to determine what to believe and it is easy to be confronted with a *best practice overload*. To overcome the above situations, organizations must *reform* their approach in adopting or defining best practices. An approach is to define best practices based on a sound framework; this is discussed in detail in the next section.

## 4.4.1 A Best Practice Framework

The general intention of best practices is to show how to apply specific development strategies and how to avoid problems, thus improving the development process and productivity. Depending on their rate of success and acceptance they may be adopted as an industry standard. The outline of this section is as follows:

- The Motivation for a Best Practice Framework
- The Framework
- Applying the Framework.

### 4.4.1.1 The Motivation for a Best Practice Framework

As noted in the previous section, it is difficult to define generic best practices because organizations are diverse. Furthermore, BP must meet the organization's specific goals and add value to the way it operates. It is also important that best practices are viewed *holistically* in terms of systems development. The author has observed that most discussions on BP concentrate on system implementation activities. Unfortunately systems development does *not* only comprise of implementation.

*A framework is necessary for BP because it provides a conceptual structure that supports their definition in a relational context.* The framework is comprised of elements which indicate the areas of focus and the structural relationship of these areas. It is important to understand this relationship because it is useful to identify side effects when best practices are implemented. For example, best practices that are implemented for resource management will affect productivity and the system's quality. The framework also supports the development infrastructure and encourages a *holistic* approach in implementing BP.

The framework is also a high level of abstraction of BP that helps to guide their definition by indicating

their scope and boundaries. The elements of the framework can also be viewed as *containers* for the BP in a specific problem area. Thus the definition of the goals and objectives of BP is more effective because the focus is concentrated. The problem of *best practice overload* is avoided because an organization can define them within the context of its own environment. Since the framework is a high level of abstraction, it helps to simplify the understanding of the focus of BP to the team. When a concept is well understood it is easier to gain acceptance and commitment from those who are involved.

Although a framework forms a foundation, it is not restrictive. New elements may be added to extend it to meet an organization's changing needs. The rapid changes in the Information Technology (IT) industry makes it necessary for an organization to constantly revise the framework.

### 4.4.1.2 The Framework

Software development is comprised of multiple interdependent phases which are followed in a logical sequence. The phases classify the nature of the effort that is required into specialised areas of responsibility and conveys the concept of "division of labour" and specialization. As a result, the roles and skills of the development team can be clearly assigned. Although the phases are separate, their boundaries are blurred. The following are typical development phases:

- Requirements Analysis
- Specification
- Project Management
- Design
- Implementation
- Maintenance
- Quality Assurance
- Change Management.

The above is a generalization and other phases may also be included. Each phase has a set of processes, activities and tasks which produces tangible output (deliverables) as evidence of their execution. The procedures employed by a phase to do the prescribed work would seriously reflect on the quality of final system. Therefore it would be reasonable to define best practices for each *phase* of software development. The framework's elements will support the definition of BP for the respective phases. As a result the best practices are "specialised" and helps to improve problem solving. Another advantage of a *phase-oriented* approach is that the domain and scope of a BP is clearly focused and its effect may be is measured.

Although a phase guides the development it is influenced by other factors. For example, the calibre of the *systems analyst* will influence the quality of the requirements analysis phase. The requirements analysis phase will influence development problems such as complexity reduction and quality. These factors are not always obvious.

The following are some typical development problems which may be confronted:

- Skilled personnel
- High quality
- Good management
- Fast reaction to changes
- Limited resources
- Reduce complexity
- Reduce cost
- Good estimations
- Innovation
- Low risk
- High productivity.

The organization's BP framework should support the definition of best practices to cope with the *phase-problem association.* Therefore the proposed framework is *phase and problem oriented,* as there is a direct relationship between them. It is referred to as the Organization Best Practice Framework (OBPF) and has been developed by the author. The objective of OBPF is to support the definition of BP across the organization. This approach aims at creating an *organizational best practice blue print* on a sound framework that can be authoritative and binding by management.

The following questions posed illustrate the initiation of BP in an environment:

- What BP will improve the requirement analysis phase?
- What BP will help to reduce complexity?
- What BP can improve quality?

| Best Practice Framework | Development Problems | Typical Development Phases |
|---|---|---|
| **Organization Culture** | Good management | The whole development |
| **Software Development Management** | Good management, Low risk | Programming, Change management |
| **Project Management** | Good estimations, Low risk | Project Management |
| **Human Role** | Skilled personnel, High Productivity | The whole development |
| **Quality Assurance** | High Quality | Quality assurance |
| **Method** | High productivity, Fast reaction to changes | Requirements Analysis, Specification, Design, Implementation, Maintenance |
| **Cost Control** | Reduce Cost | The whole development |
| **Standards** | High Quality, High productivity | The whole development |
| **Resource Management** | High productivity, Limited resources | The whole development |
| **Research and Development** | Innovation | The whole development |

**Table 4.5:** The Elements of the Organizational Best Practice Framework.

The first column of Table 4.5 lists the elements of the Organizational Best Practice Framework. It is the high level structure that defines the BP. The development problems in the second column are associated with the respective elements of the OBPF. The association indicates which elements of the framework will influence the BP for these problems. For example, the best practice that is defined by

the Cost Control element will help the organization to manage its development costs. Similarly, an organization's BP on Research and Development will influence innovation. High productivity is influenced by several elements; therefore it follows then that the overall BP of an organization will influence productivity.

The typical development phases are presented in the third column. Similar to column two, they are associated with the elements of the *framework* which will influence them. For example, the best practices which are defined under the Organisation's Culture will influence the whole development. Likewise the BP for Method will have an impact on the requirements analysis, specification, design, and implementation. In the table it is seen that the Cost Control element influences the whole development. There may be a Cost Control best practice for requirements analysis and implementation respectively. From a cost aspect, a BP may be to perform the requirements analysis in-house, while it may be a BP to outsource the implementation because it is economical.

The OBPF supports the definition of BP for the development phases and problems. As table 4.5 illustrates, these aspects form a close relationship and should not be viewed in isolation. The elements of the framework are briefly discussed below.

The *Organization Culture* defines those shared beliefs and values which unite people in an organization towards a common goal. It is a set of basic assumptions that are deemed valid and are accepted within the organization. These assumptions are regarded as the correct way to do things or understand problems in an organization (Smit and Cronje, 1992). Senior management generally drives an organization's culture and studies also show that this culture influences productivity (Dion, 1993).

*Software Development Management* is the driving force behind the development and is based on four basic management functions: planning, organizing, leading and controlling. It helps to ensure that the planned objectives are executed accordingly and achieved. It is also the basic framework to define general management principles and policies. At Quark the software development manager is responsible for all programming activities.

*Project Management* is a specialised management function that defines, plans, organizes, leads and monitors the activities of the project. It is the driving force behind the success of a project. Good project management is necessary to complete a project on time, within budget and of the prescribed quality and functionality. At Quark project management receives an operating budget from the Cost Control element (discussed below).

*Human Role* represents the significance of humans in the development. It represents their number, calibre and role. A range of roles exist, such as management, developers and users. Humans are the live wire of the organization and are influenced by its culture. A best practice may relate to deriving the maximum benefit from them.

*Quality Assurance* is also a specialised management function whose prime task is to *assure* that the system satisfies the prescribed quality. The outcome is to reduce defects and increase customer satisfaction. Good quality software will contribute to its success.

*Method* defines the procedures, tools and techniques, which are used to construct the system. It helps to guide the development based on the method's philosophy. Philosophy is the principle or a set of principles on which the method is based. An improper choice of a method will have a negative impact on the system.

Software is costly and an organization does not have unlimited resources. *Cost Control* defines the organization's BP to manage its development cost and maintain a positive flow of funds. It addresses problems such as the most effective ways to deploy capital, select projects and overall cost management.

In the framework, the *Standards* element defines the organization's BP in adopting development standards. A BP may be to adopt Projects in Controlled Environments (PRINCE) as a project management standard. Standards help to ensure good communication and uniformity in doing things.

*Resource Management* refers to the tools and technical resource infrastructure that is required to support the software development process. It defines the practices on adopting new resources and how to manage them.

*Research and Development (R&D)* is the knowledge element of the framework. It defines the practices, which an organization can implement to gain and share knowledge.

There is no one BP framework as there are many possibilities to create them. An organization can create its own framework, based on its specific needs. From the author's experience the elements of the framework are based on these key areas where most development problems originate. Although the above phases and problems can have different names they must be associated with the respective elements (as shown in Table 4.5). As a result the organizational development infrastructure is strengthened. The OBPF acts as a reference and a guide to the elements and their relationship with the phases and problems. The framework can be regarded to be generic since the elements are common to many types of development. Therefore best practices which are specific to an organization can be defined within a generic framework across development boundaries, thus adopting a holistic approach.

The elements of the framework must be considered in conjunction with each other, because a BP defined in an element can influence another. For example, R&D best practices can influence the availability of knowledge to employees (Human Role). On the other hand, BP for Cost Control will influence the amount of funds available for R&D.

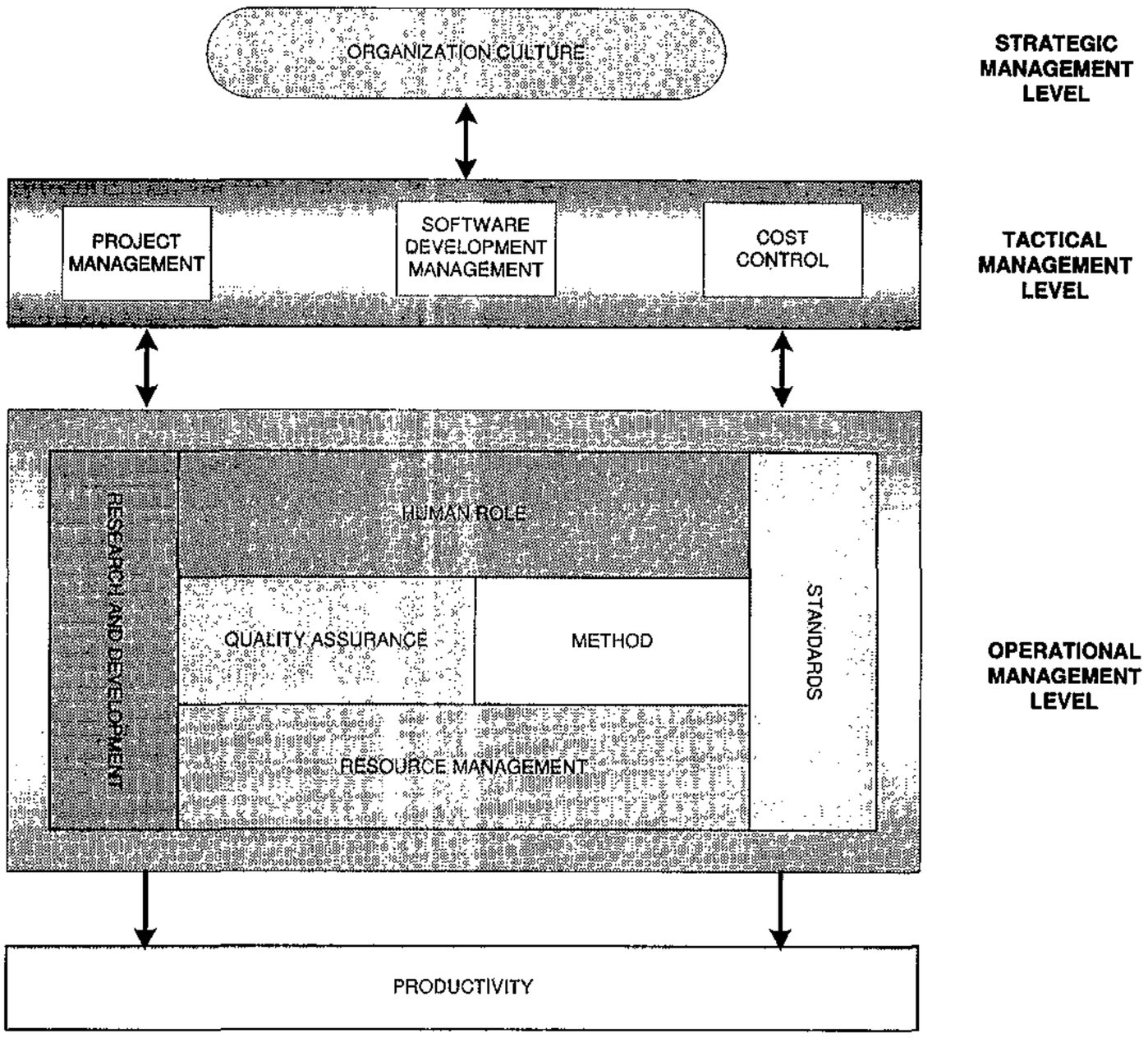


**Figure 4.5:** The Relational Structure of the Organizational Best Practice Framework.

In Figure 4.5 the structure of the framework is illustrated with its elements and their relationships. Some elements can be mapped to the processes of a typical software process model, such as quality assurance. However, in this case the motive is different. A high interaction among the elements is indicated at the operational level. Therefore any best practices which are defined for an element at this level will have a strong impact on other elements. The framework helps to expose this property, so that BP may be defined in harmony across the development environment. These interactions also act as a prompter and can encourage management to adopt a holistic view in implementing BP.

An important feature of the framework is that it supports the three basic management levels, namely, strategic, tactical and operational. In order to introduce changes it is necessary that all levels are involved and communicate with each other. The framework is their common frame of reference and helps to ensure that an alignment is established among the levels of management.

### 4.4.1.3 Applying the Framework

In this section the practical application of the OBPF is discussed. The framework *cannot* forcefully prevent a weak decision on best practices, but it can help to place it in a proper perspective in the development environment. In other words it will be associated with the most suitable element. Due to the structural relationship of the framework, an inappropriate BP will reveal its impact. The framework

is strong in providing feedback on an implementation of a BP.

It is assumed that an organization adopts or defines a BP because it experiences problems or wishes to make improvements in its environment. Normally once the situation is identified, it is discussed at senior management level and a decision is taken. It is also assumed that management is actively involved in its implementation and will follow it up. The application of the framework is case based and several BP alternatives are possible. The case examples which are described below are based on typical development problems which the author has encountered. In order to provide clarity, a suitable best practice is discussed per case.

*Case:* The users experience difficulties to use the tool Widget Builder.
*Best Practice proposals:*

**A.** Ensure that people are able to use methods and tools to their fullest.
**B.** Purchase easy to use development tools in the future.

*Decision:* Option A.
*Suitable Best Practice:* Ensure that people are able to use methods and tools to their fullest.
*Motivation:*
A larger amount of an IT department's budget is spent on software tools and utilities. Unfortunately they are not used optimally because users lack the necessary training. The author has experienced that an IT department installs the tools on a server, and then sends the installation instructions by e-mail to the users. They are left alone to read examples, take an example tour and consult help files in order to learn their usage. As a result these tools are not used effectively and efficiently, and are criticised to be unfit for use. An effort must be made to train people to ensure that they have the necessary knowledge to use them. Training may be time consuming and costly, but it is cheaper than ignorance. A good approach is to appoint a super user who can provide training and first level support. In this manner an organization can form its own assessment of the tools it uses to improve productivity and solve problems.
*Responsible Framework Element:* Resource Management, Research and Development. Resource management must ensure that when a new tool is introduced; the difficulties of its usage are also assessed. Training must be negotiated with the vendor. Research and Development must collaborate with Resource Management during its research and recommendation of new tools.

*Case:* Improve the co-operation among the team members.
*Best Practice proposals:*

**A.** Introduce interpersonal skills training.
**B.** Redefine the concept of a user.

*Decision:* Option B, as a *start* as it is economical and can be performed in-house.
*Suitable Best Practice:* Redefine the concept of a user.
*Motivation:*

Generally users are considered to be the end users of a system. A development department is always trying to satisfy their end users by accommodating their demands and striving to meet deadlines. This is a good attitude. However, the concept of a user is narrow and should be viewed from a broader perspective. The members of a development team are also users among themselves. For example, if a developer is required to create an Application Programmers Interface (API) that will be used by other developers, then they are actually the "users" of the API's creator. The API's creator must ensure that his "users" needs are satisfied and the service that he provides is acceptable. By viewing the concept of a user from a broader perspective, it tends to increase the awareness of a member's commitment to others. When this concept is implemented, those involved in the project will realize the interdependence among them and work in an improved co-operative manner. The psychological aspect is that the failure to "satisfy the user" will always be noticed. As a result people tend to become results or goals oriented and be more responsible; this benefits the project. This concept is simple, easy to implement and it works well.

*Responsible Framework Element:* Organization Culture, Human Role.

*Case:* The development team must be kept up to date on trends and technology because it is important to improve productivity.

*Best Practice proposals:*

**A.** Send team members regularly on training courses.

**B.** Conduct regular workshops.

**C.** Create a centralized knowledge base that can be accessed by everybody.

*Decision:* All options are acceptable and are dealt with as follows:

Option A will be implemented if specialised knowledge is required and is not available in-house.

Option B will be implemented as the situation demands.

Option C will be implemented because the knowledge base can be accessed across the organization.

*Suitable Best Practice:* Maintain a centralized knowledge base.

*Motivation:*

In the context of this discussion, a centralized knowledge base is a collection of knowledge from various sources and on various mediums. The following are some simple ways to achieve this goal:

- Subscription to selected magazines.
- Membership of selected professional institutes.
- An in-house technical library.
- Maintaining an Intranet with articles, documents and E-Books.
- A collection of useful links in the Internet.

Since books and magazines are expensive and many people cannot afford them. The above concept is simple and effective and it can encourage people to read and be up to date with information. From a cost point of view, it is an investment in its skills pool (with reference to Figure 4.5 - Human Role). E-

Books are also economical because they can be accessed simultaneously. It is also possible for an organization to gain membership in professional institutes and take advantage of their research publications. The knowledge base may be extended with a collection of video lectures and development *tips and tricks.* Developers are creative and often discover very smart methods to solve problems. By maintaining them on a database, knowledge can be shared in the organization.
*Responsible Framework Element:* Research and Development.

The three cases should provide a good understanding on applying the framework. By assigning the implementation of a BP to a responsible element management can exercise better control. The responsible elements are specialists in their field; for example, research and development and resource management are specialised functional areas. The advice on BP from a "specialist" can be relied upon and is in the best interest of the organization. The above cases also demonstrate the interaction between the elements, for example, Research and Development and Resource Management. This acts as an internal control and ensures integrity in BP implementation.

## 4.5 Conclusion

In this chapter the following three frameworks were proposed:

- The Metamorphic Software Engineering Environment (MSEE).
- The Iterative Meta Process Model (IMPM).
- A Framework for Best Practices.

There will always be changes and the direction and magnitude of these changes are variable and uncertain. SMO are confronted with constant changes during software development and supports the SSA as a solution to cope with them. Although the SSA guides the development processes, the development environment must also accommodate changes. The concepts of the MSEE support such a development environment because it can adjust to changes to maintain its efficiency and effectiveness. The MSEE adopts a real worldview of the development environment by differentiating between a *base environment* and a *variable environment.* The *variable environment* undergoes most changes and is conceptualised to adjust easily, thus ensuring high productivity. The MSEE helps to maintain formalism and discipline during the development, while allowing much scalability and flexibility in the environment. It attempts to eliminate restrictions in the development environment. These properties blend well with the concepts of the SSA because an organization can tailor its environment to cope with changes. The management of changes with the SSA and an adaptable SEE contributes to a balanced strategy for SMO to remain competitive. Only change will remain stable, and the MSEE encourages a *best fit* practice for a project. The creation of a MSEE is not trivial and requires strategic management planning. This effort also requires skilled personnel to create a MSEE. The advantages which a MSEE offers to SMO outweigh the disadvantages and should be exploited. Although the MSEE has been discussed with regard to SMO and SSA, it is not restricted this to domain.

Many important development issues, which should be addressed by a process model, were identified.

Since the SSA exploits the concepts of increments and iteration to cope with changes, a process model should also possess these properties. The Iterated Meta Process Model (IMPM) has this property. It exploits the concept of a model by abstracting key development areas which are common to most software development types. This development approach views the development process as individual areas of responsibility and as the *big picture.* The model exploits the concepts of *holism and synergism*, and ultimately aids the development of quality software through a quality process. This is an advantage for SMO because they are required to maintain good quality while reacting to changes. The models of the IMPM are simple to use and they identify *who* is responsible to provide answers to fundamental development questions. The model supports fundamental system concepts and satisfies Churchman's nine conditions for a system (Churchman, 1971). The SSA strongly emphasizes the importance of architecture and the IMPM assists the development process to identify basic architecture requirements.

Currently there are a variety of development best practices which may be adopted. This is not a simple task because organizations are diverse and best practice is not a one size fits all solution. In order to move away from this situation, it is more useful to define best practices within a framework. The OBPF framework forms the foundation for an organization's best practices and can be tailored according to its development environment. At a high level abstraction it helps to guide their definition by outlining their scope. A major advantage is that it prevents best practices from being defined in isolation, but rather in its used context. The nature of the framework leads to specialization in respective areas and best practices from a "specialist" are more reliable. Since the framework is a visible model it provides a common basis for communication at all levels. As a result problem analysis and decision-making are improved. However, the framework *cannot* prevent weak decision-making. The relationship among the elements of the framework enforces the impact of a best practice to be considered. This acts as an internal control and ensures integrity in BP implementation.

# 5 CONCLUSION

During this study the *core* Software Engineering in the Small problems (chapter 1, section 1.1), which confront Small and Medium sized Organizations (SMO) were identified. As these problems support the Synchronize and Stabilize Approaches (SSA) as a solution, a comprehensive understanding of them is necessary in order to exploit their application. The study was conducted as described below:

- In chapter 2 the *conceptual theory* of the SSA and *concepts of the incremental model* were discussed in depth in order to understand the *essential principles* of this sophisticated software development technique. The various approaches in the literature were reviewed in order to understand their concepts and principles. The review was also used to identify gaps and trends in the literature, which formed the basis for the chapter's recommendations.

- Chapter three is a comprehensive *critical analysis* of the SSA. The approaches were compared and their common characteristics were identified. The analysis also highlighted their strengths and weaknesses, advantages, disadvantages, business advantages and critical success factors. Basic Software Engineering (SE) principles were discussed within this context.

- Based on the *critical analysis* of the approaches it was noted that an appropriate Software Engineering Environment (SEE) and Software Process Model (SPM) are generally neglected. In order to avoid being trapped in a web of Best Practices (BP) buzzwords, a framework for best practices was considered to be more pragmatic. This is another neglected area. Therefore three frameworks, namely, *a Software Engineering Environment, a Software Process Model and a Framework for Best Practices* were developed in chapter 4.

The discussion was strengthened by examples and observations made from the Quark development environment. This helped to enrich the study context by providing a balanced discussion between theory and practice.

Some constraints were imposed on the study (Chapter 1, section 1.6). The selected subset of the approaches (chapter 2, section 2.4) surveyed were considered to be adequate as a larger subset would not have influenced the understanding of the essential principles, properties and impacts of the SSA. One of the objectives of the study was to *explicitly* shift from a subjective view on best practices (chapter 1, section 1.3). Therefore an extendable framework for best practices (chapter 4, section 4.4) was deemed to be more useful; a discussion of specific bests practices would not have been appropriate. A full case study of Quark is a *separate* area of study and with a different focus. The intention was to study the approaches in depth and support the discussion with real world examples extensively possible and in a generic manner.

## 5.1 Discussion

The constant metamorphosis of computer technology is directly related to its adaptability and flexibility. This has been observed over the decades and has a great impact on SMO. Software engineering through SSA, as indicated by the study, is a provably rapid process. Therefore in modern software development it is imperative that organizations recognize and strive to create a development environment that is more adaptive and collaborative. In order to apply the SSA as an effective solution for SMO specific objectives were defined in chapter 1, section 1.3 and these were addressed as follows in the research:

- The study sought to consolidate the understanding of the underlying theory and concepts of the SSA in a non-superficial manner. Understanding this theoretical foundation is a pre-requisite to stimulate a cumulative exploration of the subject. In chapter 2, section 2.2, the *conceptual theory of the SSA* was discussed. The *literature review* (chapter 2, section 2.4) provided an insight and depth of the various approaches in practice and how they addressed the Software Engineering in the Small problems. The review was also useful to identify trends and gaps in the literature, in order to propose solutions to address these problems. In chapter 3 the approaches were analysed and the *critical analysis report* produced provided a multi-perspective of the SSA. As a result the information is balanced and may be useful to support those organizations which may decide to adopt the SSA.

- The study also aimed to propose a SEE and a software process model that is appropriate for the SSA. In the *critical analysis* report (chapter 3), the *strengths and weaknesses of the SSA* (chapter 3, section 3.5) were discussed and summarized in table 3.3. The discussion revealed that proper development tools and a SEE appear to be neglected among the approaches. In chapter 4, section 4.2, the Metamorphic Software Engineering Environment (MSEE) was proposed and motivated in detail in order to address this problem. The *critical analysis* report also indicated that an appropriate software process model appeared to be lacking in the approaches discussed. In chapter 4, section 4.3, the Iterated Meta Process Model (IMPM) was developed and motivated in detail as an apt software process model for the approaches.

- In chapter 2, section 2.4, the various other approaches were discussed. It was clearly demonstrated that these approaches neglected the subject of current best practices. The study also indicated that there was no unanimous opinion on what constituted best practice, and there was no universal method on how to choose them. This also led to the important question: *"Who knows best?"* Since best practices is subject to opinion, an important objective of the study was to avoid the best practice hype, which was accomplished by the proposal of the Organizational Best Practice Framework (OBPF) discussed in chapter 4, section 4.4.1.

- Throughout the discussion many practical examples were cited from Quark's development environment. This was in keeping with the study's objectives to make use of practical examples from the author's work environment to demonstrate the conceptual theory of the

SSA. Many important observations made at Quark were discussed in chapter 2, section 2.5, which helped to enrich the discussion of the literature review from an implementation point of view. The discussion on Quark's synchronizes and stabilize workflow (chapter 3, section 3.10) provides an insight to a practical software development process flow.

The study strongly emphasizes that embracing changes and employing adaptable and flexible development techniques influence an organization's continuance in business. The proposed frameworks are conceptualised to cope with changes by being adaptable and flexible. These properties are also inherent to the Software Engineering in the Small problems, which make the frameworks suitable for SMO. Furthermore, the frameworks adopt a *holistic view* of the development environment and process and imply a great degree of *synergism.* Small and Medium sized Organizations *cannot* greatly benefit from a fragmented development approach. The "big picture" must always be in focus. Since these frameworks are iterative, they support these approaches which are also inherently iterative.

The software development problems with SSA are many-fold. Although these approaches endeavour to solve some of them, new problems are introduced, which are discussed below.

### 5.1.1 Development Challenges

Some of the main problems experienced included high risk, limited resources, strong competition and innovation. As there is a *no silver bullet* solution to these problems, the approaches emphasized that SMO are challenged to *re-formulate* their development strategy, by shifting away from a restrictive sequential development process (e.g. the waterfall model) to incremental development techniques. For example, Agile Software Process, Extreme Programming and Scrum apply incremental techniques to address problems such as incomplete requirements, cost, complexity and fast delivery (chapter 2, section 2.4). Good SE principles are essential to ensure disciplined development and were discussed in chapter 3, section 3.9.

Although the SSA helps to cope with the above problems, other significant impediments are introduced, such as the impact on the organization structure, the need for skilled personnel and technical challenges. The implementation of an approach will result in a change in an organization's structure. For example, Extreme Programming, Scrum and The Integrated Approach require different organization structures. A new way of thinking that is inspired by the SSA may be met with resistance to change and a proper change management process must be implemented to make a smooth transition. Furthermore, changes are costly, time consuming and there is often a steep learning curve.

Software engineering through these approaches requires experienced personnel. Since the development activities are fast paced and concurrent, a high degree of co-operation, maturity, independence, responsibility and professionalism is absolutely essential. The problem is that there are not many people of the same calibre as Kent Beck and Mikio Aoyama to lead teams in this type of development environment. Statistics indicate that about 50 percent of the world's software developers

are below average (Boehm, 2002). Hence, it is difficult and costly to recruit a proper balance of skills.

There are many technical challenges confronting development through the SSA. It is difficult to make early architectural decisions because the system evolves with each increment and the architecture must evolve accordingly. Therefore weak decisions will cause the system to decay and will be costly to re-engineer. The integration of new increments is also not as simple as the SSA may imply and the complexity increases progressively and is non-deterministic. However, from a positive note the techniques of the SSA help to reduce complexity to manageable increments. Technical difficulties are also aggravated by the current lack of skills in these approaches.

SSA Software development requires that management have a clear vision of its direction and is able to adapt to changes rapidly. It is important that a clear statement of its goals and objectives are defined and communicated to all parties involved. The potential for conflict with the SSA is high because interests differ at various organizational levels. For example, product management may want to include certain features in a version of the software, but development management cannot implement them due to technical reasons. Therefore management must work co-operatively towards a common goal, practise priority management and implement conflict resolution management

Although the above factors may be generally considered and practised in many development environments, SSA requires a stronger measure of them. Therefore competent software development management must be practised.

### 5.1.2 The Future

With the increasing demand to produce high quality software rapidly, the use of SSA is likely to increase in the future. In addition many SMO are scaling down their operations. The approaches are beginning to receive increased attention in the industry; this is especially true for Agile Software Process, Extreme Programming and software development with components. The software industry will continue to introduce new tools and utilities to support rapid application development. However, they are not mature and caution must be exercised in their selection. Many major organizations such as Quark, Netscape, Alcatel, Microsoft and Mozilla successfully apply the SSA during development and could serve as a good reference for *newcomers*.

The value of the study was previously defined and is summarized below:

- The material covered by the objectives of the study (section 1.3) can be seen as a general contribution to the software engineering approach of *synchronized and stabilized* development.
- The *critical analysis* report (chapter 3), discusses the strengths and weaknesses, advantages, disadvantages, business advantages and critical success factors of these approaches, which may be useful when managing problems within the SSA. A typical *synchronize and stabilize workflow* (section 3.10) that reflects reality may also be considered useful.

- Although the approaches were comprehensively examined, it is not the final word. Information systems students may find the discussion on the *conceptual theory of the SSA* (chapter 2, section 2.1), the *critical analysis report* (chapter 3) and the *proposed frameworks* (chapter 4) stimulating. As a result through debate and constructive dialogue the attention of the software community may be drawn to this often neglected area of software development.

Further research is necessary to discover other similar approaches and to improve the existing ones. In chapter 3, section 3.4, it was noted that the techniques associated with these approaches may be applied in conjunction to one another. Studies on how to optimally combine best techniques from these approaches and an analysis of their impact can be used to define a "best of the breed" approach. As computer technology will continue to change rapidly, the requirements for adaptability and flexibility in the SEE and development process will increase. Therefore further research on adaptable software engineering environments and software process models will benefit the continued adoption of the synchronize and stabilize development approach.

# BIBLIOGRAPHY

## References

Ahituv, N. & Neumann, S. 1990. *Principles of Management Information Systems.* Wm: C. Brown Publishers. Third edition.

Andriole, S.J. !995. Debatable Development: What Should We Believe?, *IEEE Software.*

Aoyama, M. 1998a. Agile Software Process and Its Experience, *IEEE, The 20th International Conference on Software Engineering.*

Aoyama, M. 1998b. Web-Based Agile Software development, *IEEE Software,* 58-65.

Avison, D.E. & Fitzgerald, G. 1995. *Information Systems Development: Methodologies, Techniques and Tools.* McGraw-Hill Companies. Second edition.

Beck, K. 1999. Embracing Change with Extreme Programming, *Computer,* 60-66.

Boehm, B. 2002. Get Ready for Agile Methods, with Care, *Computer*, 64-69.

Brooks, F.P. Jr. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering, *Computer*, 10-19.

Brusilovsky, P. & Maybury, M. T. May 2002. From Adaptive Hypermedia to The Adaptive Web, *Communications of the ACM. 45*(5).

Carmel, E. & Becker, S. 1995. A process model for packaged software development, *IEEE Transactions on Engineering Management.* 41(5). 50-61.

Churchman, C.W. 1971. *The Design of Inquiring Systems: Basic Concepts of Systems and Organization.* New York: Basic Books.

Comaford, C. 1995. What's Wrong with Software Development?, *Software Development.*

Conradi, R. & Westfechtel, B. 1998. Version Models for Software Configuration Management, *Communications of the ACM. 3*0(2).

Cox, B.J. 1990. There *I*s a Silver Bullet, *BYTE*, 209-218.

Cusumano, M.A. & Yoffie, D.B. 1999. Software Development on Internet Time, *Computer,* 60-66.

Cusumano, M.A. & Selby, R.W. 1997. How Microsoft Builds Software, *Communications of the ACM.* 40(6): 53-61.

Cusumano, M.A. & Selby, R.W. 1995. *Microsoft Secrets*. The Free Press.

Deitel, H.M., Deitel, P.J. & Nieto, T.R. 2002. *Internet and World Wide Web-How to Program*, Prentice-Hall Inc. Second edition.

Dion, R. 1993. Process Improvement and the Corporate Balance, *IEEE Software*, 28-35.

Dunn, R.H. & Ullman, R.S. 1994. *TQM for Computer Software.* McGraw-Hill Companies. Second edition.

Dyba, T. 2000. Improvisation in Small Software Organisations, *IEEE Software,* 17(5): 82-87.

Fayad, M.E., Laitinen M. & Ward R.P. 2000a. Software Engineering in the Small, *Communications of the ACM*, 43(3): 115-118.

Fayad, M.E., Laitinen M. & Ward R.P. 2000b. Management in the Small, *Communications of the ACM*, 43(11): 113-116.

Fenton, N. 1996. Point – Counterpoint, *IEEE Software.*

Gane, C. & Sarson, T. 1979. *Structured Systems Analysis: tools and techniques.* Prentice Hall.

Gilb, T. 1988. *Principles of Software Engineering Management Information Systems.* Workingham: Addison Wesley. UK.

Graham, D.R. 1989. Incremental Development: review of non-monolithic life-cycle development models, *Information and Software Technology*, 31(1): 7-20.

Goldberg, R. 1986. Software Engineering an Emerging Discipline, *IBM SYS Inc.* 25(3/4).

Griss, M.L. & Pour, G. 2001. Accelerating Development with Agent Components, *Computer,* 37-43.

Heineman, G.T., Botsford, J.E., Caldiera, G., Kaiser, G.E., Kellner, M.I. & Madhavji, N.H. 1994. Emerging technologies that support a software process life cycle, *IBM Systems Journa*l,33(3).

Jones, A. 2000. The Challenge of Building Survival Information Intensive Systems, *Computer,* 39-43.

Jones, R. 1993. Complexity Made Simple, *EXE: The Software Developers Magazine.*

Knauber, P., Muthig, D., Schmid, K. & Widen, T. 2000. Applying Product Line Concepts in Small and Medium-Sized Companies, *IEEE Software,* 17(5): 88-95.

Lutz, M.J. 2001. Software Engineering on Internet Time, *Computer,* 36.

Macala, R.R., Stuckey, Jr., L.D. & Gross, D.C. 1996. Managing Domain-Specific, Product-Line Development, *IEEE Software,* 13(3): 57-67.

McConnell, S. 2000. The Best Influences on Software Engineering, *IEEE Software,* 17(1): 10-17.

McConnell, S. 1996. *Rapid Development: Taming Wild Software Schedules.* Microsoft Press.

Moitra, D. October 1999. Software Engineering in the Small, *Computer,* 39-40.

Motoyoshi, Y., Otsuki, S. 1998. An Incremental Project Plan: Introducing Cleanroom Method and Object-Oriented Development Method, *IEEE, The 20th International Conference on Software Engineering.*

Nunes, N.J. & Cunha, J.F. 2000. Wisdom: A Software Engineering Method for Small Software Development Companies, *IEEE Software,* 17(5): 113-119.

O'Brien, L. 1995. The Ten Commandments of Tool Selection, *Software Development.*

Parnas, D.L. 1985. Software Aspects of Strategic Defense Systems, *Communications of the ACM,* 28(12): 1326-1335.

Petersen, G. 1999. Who Knows Best? *Crosstalk: The Journal of Defence Software Engineering.*

Powell, T.A. 1998. *Web Site Engineering – beyond web page design.* Prentice Hall.

Powers, M. J., Cheney, P.H. & Crow, G. 1990. *Structured Systems Development – Analysis, Design, Implementation.* Boyd and Fraser. Second edition.

Rajlich, V.T. & Bennett, K.H. 2000. A Staged Model for the Software Life Cycle, *Computer, 66-71.*

Rauterberg, M. 1992, An Iterative-Cyclic Software Process Model, *Work and Organisational Psychology Unit*, Swiss Federal Institute of Technology (ETH).

Repenning, A, et. al. 2001. Using Components for Rapid Distributed Software Development, *IEEE*

*Software, 38-45.*

Rising, L. & Janoff, N. S. 2000. The Scrum Software Development Process for Small Teams, *IEEE Software,* 17(4): 28-32.

Royce, W. 2000. Software Management Renaissance, *IEEE Software,* 17(4): 116-118, 121.

Royce, W. 1970. Managing the Development of Large Software Systems: Concepts and Techniques, *WESCON Technical Papers, Western Electronic Show and Convention,* 1-9.

Russ, M.L. & McGregor, J.D. 2000. A Software Development Process for Small Projects, *IEEE Software,* 17(5): 96-101.

Schach, S.R., 1993. *Software Engineering.* Irwin and Aksen Associates. Second edition.

Shimmin, B. F. November 1995. IBM Software Suite Lets Developers Collaborate, *LAN Times.*

Smit, P. J. & Cronje, G.J. de. J. 1992. *Management Principles,* Juta & Co., Ltd. First edition.

Strohm, O. 1991. Projektmanagement bei der Softwareentwicklung. In: Ackermann, D. und Ulrich, E. (Ed.) Software-Ergonomie 1991. (*Reports of the German Chapter of ACM,* Vol. 33). Stuttgart: Teubner. 46-48.

Voas, J.M. 1998. Certifying Off-the-Shelf Software Components, *Computer,* 53-59.

Woodward S. 1999. Evolutionary Project Management, *Computer,* 49-57.

Yoshioka, N., Suzuki, M. & Katayama, T. 2000. Incremental Software Development Method Based on Abstract Interpretation, IEEE 9th International Workshop on Software Specification and Design.

## Web References

[W1] Kruchten, P. September 2001. Going Over the Waterfall with the RUP, *The Rational Edge,* http://www.therationaledge.com/content/sep_01/t_waterfall_pk.html January 2002

[W2] Software Development, *Software Research*
http://www.softwareresearch.com/SoftwareDevelopment/SoftwareDevelopment.htm
June 2002

[W3] SCRUM Software Development Process, http://www.controlchaos.com/scrumwp.htm
June 2002

[W4] Refactorings in Alphabetic Order, http://www.refactoring.com/catalog/index.html
July 2002

[W5] Gilb, T. 1997. Evo Manuscript MINI Version, http://www.result-planning.com
August 2002