Best Practices for Evolutionary Software Development

by
Darren Bronson

B. S. Computer Engineering, University of Illinois, 1992
M. S. Electrical Engineering, Stanford University, 1994

Submitted to the Sloan School of Management and the
Department of Electrical Engineering and Computer Science in partial fulfillment
of the requirements for the degrees of

Master of Business Administration
and
Master of Science in Electrical Engineering and Computer Science

in conjunction with the
Leaders for Manufacturing Program

At the
Massachusetts Institute of Technology
June 1999

©1999 Massachusetts Institute of Technology, All rights reserved

Signature of Author _____
Sloan School of Management
Department of Electrical Engineering and Computer Science

Certified by _____
Michael Cusumano, Thesis Advisor
Sloan Distinguished Professor of Management

Certified by _____
Daniel Jackson, Thesis Advisor
Associate Professor of Electrical Engineering and Computer Science

Accepted by _____
Arthur C. Smith, Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Accepted by _____
Lawrence S. Abeln, Director of the Masters Program
Sloan School of Management

Best Practices for
Evolutionary Software Development

by
Darren F. Bronson

Submitted to the MIT Sloan School of Management
and the Department of Electrical Engineering and Computer Science
on May 9, 1999
in partial fulfillment of the requirements for the degrees of
Master of Business Administration
and Master of Science in Electrical Engineering and Computer Science

Abstract

In the past few years, Evolutionary Software Development processes have been adopted by many development groups at Hewlett-Packard as an alternative to Waterfall Development. While there have been many successes with the new process, there also have been several groups that have attempted to adopt it, yet have decided to revert back to a simpler Waterfall process. Process consultants at HP believe that while the process has tremendous potential, it is more complicated to manage than previous processes and has some drawbacks when used inefficiently.

This project involves analyzing the efforts of Hewlett-Packard software teams that have adopted Evolutionary Development in order to determine which are the factors that most impact a group's success with the process. These efforts studied include management of project milestones, software integration, software architecture, and testing.

First, data on 30 previously completed evolutionary projects was collected. Case studies were performed for four of these projects. These cases were then used in the evaluation stage, where experts with the process drew conclusions about which factors have the most impact on success. Next, documents were created which list best practices for using the process. Finally, experts reviewed these documents and amended them until they become satisfactory tools for communicating best practices with teams.

Acknowledgements

Many people have contributed their knowledge, ideas, and encouragement toward the creation of this thesis. To those people I am truly grateful.

First and foremost, I would like to thank Bill Crandall, my manager at HP for the length of the internship. An alum himself (LFM'94), Bill is well familiar with the LFM program and recognized the benefit that an LFM internship could bring to this project. Bill originally conceived the conceived the structure and goals for the project. Yet gave me the freedom to run the project how I saw fit, so that I brought an outsider's view to the problem. Also, he gave a lot of his time, in terms of mentoring, lending his expertise, and encouraging me, which helped me learn a great deal.

My management advisor, Michael Cusumano, was also instrumental in creating the thesis. Although, he has been overloaded with a book release and two landmark trials, he still has been able to devote adequate time to advising me. Michael is truly a "guru" in the field of software development and gave me much insight into how to conduct research in this area.

Daniel Jackson, my engineering advisor, was a great resource. He provided a wealth of advice to help me better manage the research process and methods.

The "Evo team", consisting of several PGC consultants was also essential during the internship. Besides Bill, the team consisted of Nancy Near, Guy Cox, Ruth Malan, Derek Coleman, and Todd Cotton. Without their continual feedback and guidance, this project could never have gotten off the ground. Further help at HP was given by Elaine May, perhaps HP's leading expert with the process, who also gave valuable feedback on my work. Also, I am grateful to the teams that let me interview them for the case studies, particularly the project managers, which helped organize the process.

Tom Gilb, the originator of Evolutionary Development, also helped guide the project several times throughout its course. Tom's great skill as a visionary in the field helped us see the project in new light.

Lastly, I'd like to thank the LFM program, which organizes the internship process and provided a great education for me over the last two years. Thanks to all the directors, faculty, staff, and students.

Table of Contents

1 INTRODUCTION 9

1.1 CONTEXT 9
The internship 9
Software development processes 9
Current software development trends 9
Software development at HP 10
1.2 LIFECYCLE PROCESSES MODELS 10
1.3 EVOLUTIONARY SOFTWARE DEVELOPMENT AT HP 11
1.4 PROBLEM DEFINITION 11
1.5 STRUCTURE OF THE THESIS 12

2 DESCRIPTION OF EVOLUTIONARY DEVELOPMENT 13

2.1 BENEFITS OF EVOLUTIONARY DEVELOPMENT 13
Increased visibility of progress 13
Reduced schedule risk 13
Reduced risk of customer acceptance 14
Reduced risk of integration 14
Early introduction to market 14
Increased motivation of development team 15
2.2 COMPARISON TO OTHER LIFECYCLE MODELS 15
Waterfall Development 16
Rapid Prototyping 16
Incremental Development 17
Evolutionary Development 17
RELATIONSHIP OF MODEL BENEFITS AND COSTS 17

3 RESEARCH METHODOLOGY 19

3.1 CREATION OF A PAST PROJECT MATRIX 19
3.2 CREATION OF CONSULTING MODELS 19
3.3 CASE STUDIES 20
3.4 WEEKLY MEETINGS 21
3.5 CONCLUSION OF BEST PRACTICES 21
3.6 PILOT TEST OF BEST PRACTICES 22

4 OVERVIEW OF CASE STUDIES 23

4.1 DEEP FREEZE 23
4.2 REDWOOD FIRMWARE 23
4.3 CASPER 24
4.4 BISTRO 24

5 CONCLUDED BEST PRACTICES 26

5.1 PHASE I: ATTEND GROUND SCHOOL 26
Understand Evo philosophy 26
Develop product concept 26
Set project goals 27
Identify risks and dependencies of project 28
Create plan for "Flight Plans" stage 28
5.2 PHASE II: MAKE FLIGHT PLANS 29

Develop initial requirements *29*
Develop high-level architecture *29*
Create development environment *30*
Create Evo plan for "Missions" stage *31*
5.3 PHASE III: FLY MISSIONS 33
Takeoff and land frequently--design, develop, integrate, and test incrementally *33*
Evolve requirements and architecture *34*
Keep your eye out the window--get user feedback early and often *34*
Build frequently *35*
Debrief after each mission *36*

6 GENERAL CONCLUSIONS 37

6.1 PROCESS PREPARATION 37
6.2 PROCESS PHILOSOPHY 37
6.3 TEAM COMPETENCE 37
6.4 CREATING THE "RIGHT PRODUCT" 37
6.5 MOTIVATIONAL IMPLICATIONS 38

7 RECOMMENDATIONS FOR FUTURE RESEARCH 39

7.1 QUANTITATIVE STUDY 39
7.2 IMPORTANCE OF FLEXIBLE ARCHITECTURES 39
7.3 TAILORING OF THE MODEL TO SPECIFIC NEEDS (LARGE TEAMS, NEW MARKETS, ETC.) 39
7.4 RELATION TO OPEN SOURCE SOFTWARE (OSS) METHOD 40

BIBLIOGRAPHY 41

APPENDIX I: PAST PROJECT MATRIX DESCRIPTION 43

ROWS AND COLUMNS OF THE MATRIX 43
NOTES ON THE DATA 43
CONCLUSIONS FROM AGGREGATE DATA 43

APPENDIX II: FULL TEXTS OF CASE STUDIES 44

DEEP FREEZE 41
REDWOOD FIRMWARE 44
CASPER 48
BISTRO 51

1 Introduction

1.1 Context

The internship

Research for this thesis was conducted during a six-month on-site internship as a part of MIT's Leaders for Manufacturing (LFM) Program. LFM is a partnership between 14 major U.S. manufacturing companies. Fellows of the LFM pursue both a master in management and a masters in engineering during the 24 month program. Internships are generally setup so that the student can both perform valuable research worthy of an MIT thesis and also make a significant impact on the sponsoring company.

Software development processes

In the 50 years that computers have existed, most of the focus has been on hardware. Creating hardware has always been orders of magnitude more difficult than creating software. Money was made from selling hardware, not software. Hence, many processes were created to better manage the creation of hardware, while software was usually slapped together quickly by a small team and updated later if bugs were found.

Nowadays, thanks to the increase of CPU power, software code bases are much larger and more complicated than ever before. To create operating systems and major applications, teams of several hundred programmers are required. To manage teams today, large and small, developers must use stricter processes. However, existing software processes are much less developed than their hardware counterparts.

Because few software practices have become dominant standards, one group may use a vastly different process than another. Also, since groups have different interpretations of the processes, even two groups using the same process may be running it very differently.

Current software development trends

Just as with the development of computer hardware, the need for fast development cycles is great. Releasing a product a few months early or late can sometimes be the difference between success and failure. Groups often consider schedule to be their highest priority.

There is a wide variety of process competence between software groups today. Yet, the importance of process capabilities is being more realized lately. Much of the recent focus on process is thanks to the Capabilities Maturity Model (CMM), created by Carnegie Mellon's Software Engineering Institute. The CMM is a highly popularized tool, which helps a team benchmark its process capabilities against other software groups. Based on the responses to the CMM questionnaire, teams are assigned a number between 1 and 5, where a higher number means higher process competence. Knowledge of the CMM has been wide spread in the U.S., and teams often set explicit goals of transitioning to a higher CMM rating over periods of time. (Note that the CMM does not include a lifecycle model, and hence it is not compared directly to the models presented in section 2.2.)

Software development at HP

Hewlett Packard, traditionally known as a "hardware company", is finding itself developing more and more software. Like many companies, HP now employs more software engineers than hardware engineers. Still, since few of its end products are strictly software, the company still views itself as mainly a hardware company. Many of the goals set for software projects are derived from goals used for hardware teams. For example, teams sometimes tend to focus on quality more so than innovation and cycle time.

To assist the specific needs of the many software development groups, an internal software development consulting group was setup. Process Generation Consulting (PGC), formerly the Software Initiative, has been assisting various software teams with both technical and managerial issues for the last 12 years. Since this group has easy access to many HP software teams that use similar processes, it is an ideal group from which to conduct a research study. PGC hired me to study how its customers were using one particular process, Evolutionary Development.

In recent years, as an effort to combat the lack of revenue growth, HP upper management has been preaching the value of new product generation. Most of the company's revenues now come from new products and versions of products introduced in the last 2 years, which it proudly publicizes. Yet, there is a drive to be even more reliant on new products and product versions. Thus, demand for processes that assist with new products and allow faster cycle times is high. Many groups, when adopting a new software process, request the assistance of consultants from PGC to help them ease the transition to the new process.

1.2 Lifecycle processes models

Software lifecycle processes help manage the order and interrelationship between the various stages of development. The major development stages are investigation, design, implementation, and test. Several models have been used to base processes upon. The simplest and most traditionally used model has been the "Waterfall" model. In the waterfall model, the 4 stages occur serially, without overlapping or repeating. Before the Waterfall, groups often lumped the stages together. When this occurs, rework often results from tasks being performed without proper groundwork and planning. By helping teams avoid such rework, the Waterfall model has served developers well for many years.

Yet, there are many limitations to Waterfall Development. Creating hard transitions between stages makes it difficult to go back and change decisions that were made in previous stages[1]. Because of this major limitation, new lifecycle models have been introduced in recent years.

The Spiral Development model[2], introduced by Boehm in 1988, received much attention. The model allows teams to create a product in a handful of cycles of increasing magnitude. With the primary intention of reducing development risk, each cycle included checkpoints for risk assessment. Many teams have created processes based on the Spiral model, and many are satisfied with it. However, the Spiral model has often been criticized as being too complicated for most development teams.[3]

---

[1] Steve McConnell, Rapid Development, Microsoft Press, p. 138.

[2] Barry Boehm. "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988, pp. 61-72.

[3] McConnell, p. 143.

Several other models have been also introduced. The various models have much in common with each other. However, there are some aspects which distinguish one model from another. For example, cycles are present in many of the models, borrowing the concept of PDCA cycles from Total Quality Management. However, some models involve only 3-5 cycles, while others involve 20-50. Also, when groups implement a model into their software process, the implementations of the models often differ vastly. Developers often decide to add or leave out certain aspects that the models recommend, usually for purposes of simplification. Because of the similarity of the models and the variation of their implementations, there is much confusion over the differences between them. McConnell has compared 10 models for development in recent work, yet there is still much overlap between the models. Hence, it is often arguable which model a given development project is based on.

This thesis is not attempting to redefine the various lifecycle models. Yet, for the purposes of distinguishing between them, we will use some characteristics of the models to create working definitions. The conclusions presented here are not only applicable toward users of Evolutionary and Incremental Development, but also for other processes that share the characteristics of having iterative cycles and/or utilizing user feedback.

1.3 Evolutionary Software Development at HP

Evolutionary Development, often called Evo, was first introduced by Tom Gilb, in the book Principles of Software Engineering Management.[4] Like many other lifecycle models, Evo was proposed as an alternative to the waterfall model. Gilb has further defined the process and created tools to aid it in later works.

Elaine May, as a project manager, was the first to formally use the process at HP in 1991. Gilb was hired as a consultant to help the group adopt the process. May's team discovered that it was possible to relax some of Gilb's instructions for implementing Evo. In particular, they used internal or "surrogate customers", to give much of the user feedback, because of the difficulty in using external customers[5]. After the project, May left the team to join PGC as a software process consultant. There, she helped other teams throughout the company similarly adopt the process. After her departure from the group in 1995, PGC continued to consult on the process.

Since 1991, PGC has assisted around 30 HP teams with the process over the last 8 years. Teams have varied the process dramatically, and some have had great success, while others have failed. Some of the failures have been attributed to early lack of confidence in the process, as well as increased overhead in running the process[6].

1.4 Problem Definition

The problem as defined by PGC managers was to determine why some groups found Evo to be greatly beneficial, while others have found it mediocre or even aborted the process midway through a project.

Because the process is regarded so highly, most all of the consultants in PGC would prescribe groups to shift from Waterfall Development to Evolutionary Development. However, recent failures to adopt the

---

[4] Tom Gilb, Principles of Software Engineering Management, Addison Wesley, 1988.

[5] Elaine May and Barbara Zimmer, "The Evolutionary Development Model for Software Development", *HP Journal*, August 1996, p.1.

[6] May and Zimmer, pp.3-4

process cast some doubt on Evo's image as the "miracle cure". What was most interesting was the disparity between some teams' classifications of the project as successful and other teams' claims that it is ineffective. PGC consultants hypothesized that the process had great potential, yet it would succeed or fail, based on how it was implemented. Perhaps teams were basing critical decisions such as their cycle lengths based on intuition, when there might be a better method to decide such parameters, based on studying the success of past projects.

Thus, my task was to determine how teams could better handle implementation decisions, in order to make their Evo adoption more effective.

1.5 Structure of the thesis

This thesis begins by defining and describing Evolutionary Development in Chapter 2. In order to make the definition more clear, the evolutionary model is compared to 3 other lifecycle models. Also, the relationships between the 4 models is discussed, as well as the benefits of transitioning from one model to another.

Next, Chapter 3 describes the research methodology I used in conducting research and generating best practices.

Chapter 4 is a summary of the four case studies that were performed on teams that had used Evolutionary or Incremental Development to produce a product. Full texts of the cases are included as appendices.

The next section, chapter 5, lists the best practices that were concluded from the research. These best practices are grouped into 3 stages, according to when, chronologically, they are applied during a project. For each best practice, reasoning behind using the practice is discussed, as well as heuristics for implementing the practice.

Chapter 6 then presents some general conclusions about the about the process and how HP software groups operate in general. Included here are discussions of the most critical best practices, barriers to success with the process, and the organizational impact of the process.

Finally, chapter 7 lists the topics that this study did not yet address and provides some insight into what research can be performed in the future, concerning software lifecycle processes.

2 Description of Evolutionary Development

Evolutionary Development (Evo) is a software development methodology in which a product is created via many iterative delivery cycles, while gathering of user feedback regularly during development. Within each delivery cycle, the software is designed, coded, tested and then delivered to users. The users give feedback on the product and the team responds, often by changing the product, plans, or process. These cycles continue until the product is shipped.

2.1 Benefits of Evolutionary Development

Groups decide to follow an Evolutionary Development model for various reasons. At HP, managers generally state 2 or 3 of the following benefits as their motivation for using the process.

Increased visibility of progress

Because the product is developed in cycles, each cycle boundary serves as a checkpoint for progress. The traditional method of estimating progress, where team members estimate what percent of their code is complete, is highly inaccurate. In the evolutionary approach, features are developed serially and are checked into a working code base upon each feature's completion. Progress monitoring becomes much more accurate, since features are deemed complete when they are both integrated and runable.

Reduced schedule risk

Developing features sequentially allows teams to save low priority features for last. Then, if the schedule slips, the latest cycles can simply be skipped to pull in the schedule. Compared to the waterfall approach, which usually requires a project to build all desired features in parallel during a single development phase, this approach makes the project much more likely to hit its market window.

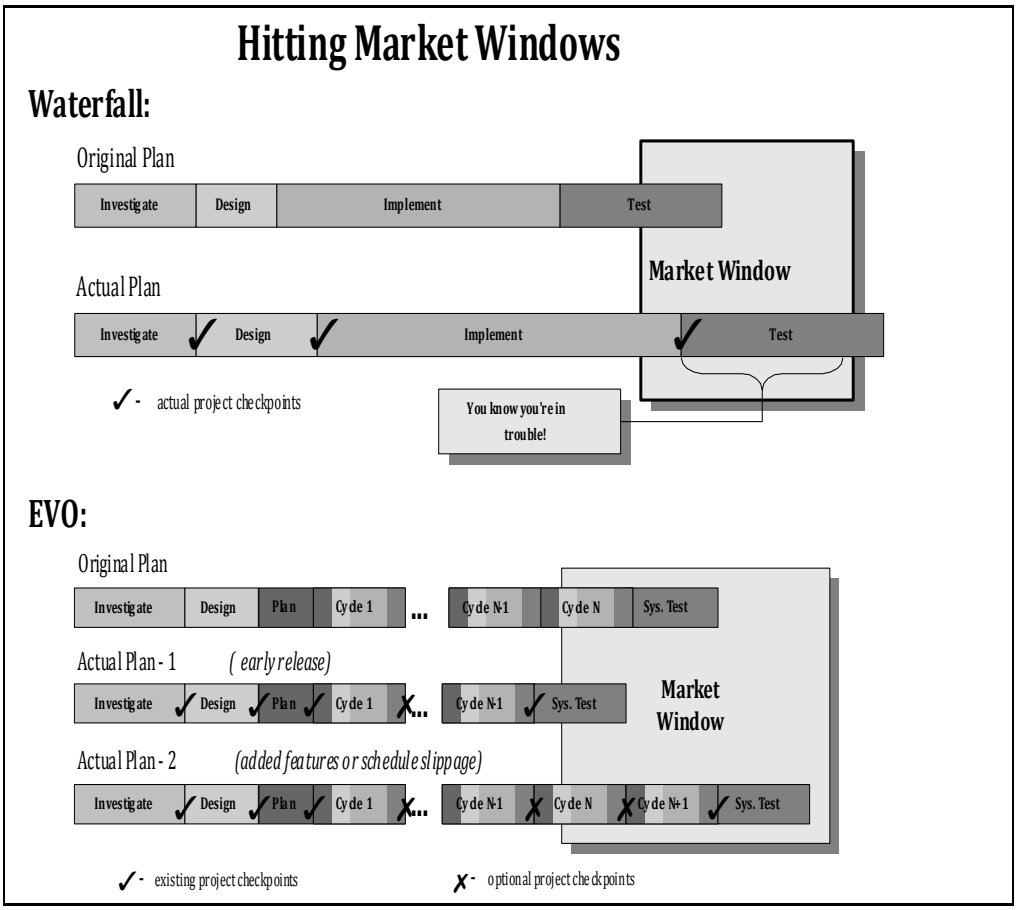


Figure 1: Ability of the lifecycle process to adapt to schedule changes

Reduced risk of customer acceptance

Evolutionary Development involves getting feedback from customers (and those who can act as customers) throughout the development process. This feedback often helps the team make critical decisions about the interface and feature set, so as to increase the odds of customer acceptance.

Reduced risk of integration

Big bang approaches, in which all subsystems are integrated at the end, can become a huge time sink at the end of a project. Several projects at HP have spent months or even failed because their integration has been overly complex. (The rationale is as follows. If 1 bug caused by 1 new code section takes 1 minute to fix, then 10 bugs caused by 10 new code sections will take >10 minutes to fix, because there is more new code to sift through when searching for each bug.) Incremental integration at every cycle boundary reduces this risk.

Early introduction to market

Early versions of the product can be easily released with only the major features available. Only a brief Q/A phase is needed to complete the product after the team concludes that the current cycle should be the last.

Increased motivation of development team

Seeing a working product evolve every cycle is usually a rewarding experience for developers. One team member's check-ins become immediately available and usable by others.

2.2 Comparison to other lifecycle models

There are two major characteristics of Evolutionary Development that make it distinct from other models: (1) delivery of the product in short discrete cycles, and (2) early and frequent user feedback. Most lifecycle models can be differentiated based on how strictly they adhere to each of these characteristics.

A delivery cycle is defined as a portion of the project's implementation phase, which concludes with a working product (or the software portion of a product). Some software projects do not integrate the code together into a single executable until the end of the project, thus creating only one delivery cycle. Other projects are divided up into multiple deliveries for various purposes. The upper bound observed at Hewlett Packard is around 50 deliveries, in which about 2% of the project is developed in each cycle.

User feedback can be any sort of feedback the team receives from an end user or someone emulating an end user. The primary purpose of this feedback is to help the team decide which features are important in the product and how those features should be implemented to best meet the user's needs. The latest when feedback can begin is at the product's release to the market. (Most products receive their first user feedback at their Beta release, at which time most of the feedback is used for bug fixes.) The earliest that a product can receive feedback is at the beginning of the implementation phase of the project, by creating a prototype or patching together a few key features to deliver to users.

For simplicity, we will consider only 4 development models: Waterfall, Incremental Development, Rapid Prototyping and Evolutionary Development. These definitions shall be working definitions for the purposes of this analysis, and are not meant to undermine definitions that are currently used by others.

The figure below shows how the four models relate to the two characteristics:

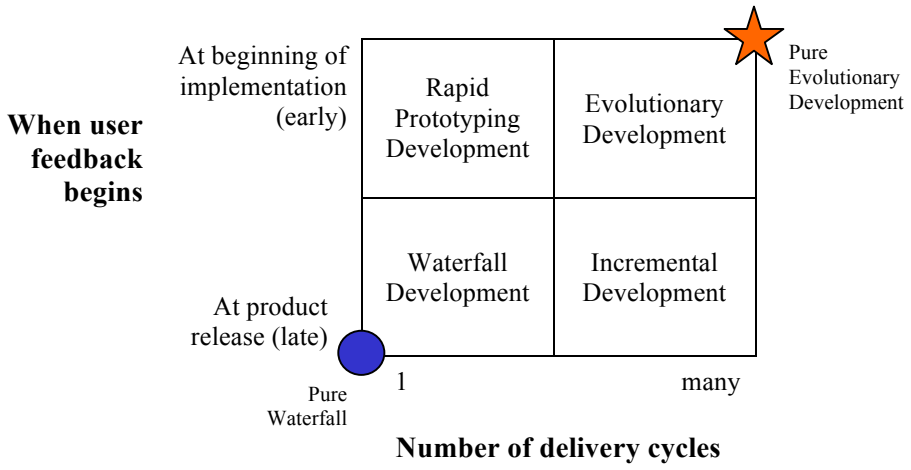


Figure 2: Model comparison matrix

Timeline diagrams are now shown to further illustrate the differences between the models:

Waterfall Development

Figure 3: Waterfall Development Lifecycle (1 cycle, no user feedback)

The traditional Waterfall model strings together investigation, design, implementation, and test phases sequentially and there is usually no delivery of a working product until the end of the project. Thus, there is only one delivery. There is also typically little or no user feedback involved in the process until late, often at beta release time or later.

The Waterfall model is strong in creating explicit checkpoints between phases and allowing groups to move forward with confidence. However, it is often criticized for making it difficult to backtrack to previous phases in order to make changes.

Rapid Prototyping Development

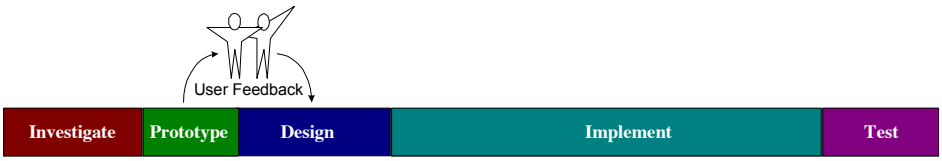


Figure 4: Rapid Prototyping Development Lifecycle (1 cycle, early user feedback)

For Rapid Prototyping Development, we assume a basic Waterfall model with the added measure of prototype creation early in the product's implementation phase. Prototyping is usually performed either to (1) assess customer acceptance or to (2) assess feasibility of a new technology. Both are considered forms of user feedback, even though developers may be emulating the user when evaluating the prototype. It is conventional wisdom to throw away the prototype and start from scratch since the prototype is not based on a thoroughly planned architecture. (Note that user feedback here is slightly earlier than that of Evolutionary Development, since the feedback occurs before the design phase. Evolutionary Development can also be managed with a throwaway prototype up front, in order to push user feedback even earlier.)

The benefit of prototyping is the knowledge gained from early feedback. Adjustments can be made or the project cancelled if deemed appropriate. The downside is that prototyping efforts consume valuable resources and early feedback is never entirely accurate. Also, managers and customers may mistakenly assume that the whole product is "almost ready" after seeing the prototype.

Incremental Development

Figure 5: Incremental Development Lifecycle (many cycles, no user feedback)

Incremental projects generally contain 3 or more delivery cycles, where a working executable is ready at the end of each cycle. Some projects contain 3-5 long cycles are usually described as "milestone approaches" or "phased development." Other projects contain around 50 short cycles, around 1-3 weeks each. No user feedback is collected until product release or upon release of the Beta version at the earliest.

Products are usually generated via delivery cycles to allow the team visibility of progress at the end of each cycle. If progress is greater or less than what was predicted, the schedule may be adjusted by adding or skipping features in the product. Also, continual integration makes a project more manageable by reducing risk from big-bang integration and helping teams spot problems earlier. The cost of the process is usually the extra overhead of management tasks at each cycle boundary.

Evolutionary Development

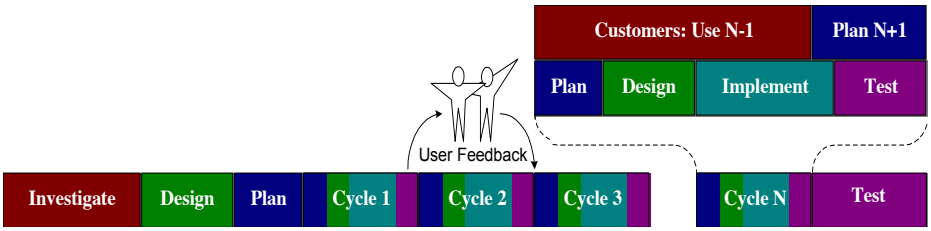


Figure 6: Evolutionary Development Lifecycle (many cycles, early user feedback)

Evolutionary projects contain both many cycles and early feedback and take advantage of the synergy between the two characteristics. In products with many delivery cycles, since a working product is ready at the end of each cycle, this product may be used to get feedback from users, often with little overhead.

The benefits, therefore, of Evolutionary Development are the sum of the benefits of both Rapid Prototyping and Incremental Development, without consuming resources normally required for prototyping. However, the downside is that the effort required in project management tasks is increased somewhat over Incremental Development, due to managing the relationship with and the delivery to the users.

2.3 Relationship of Model Benefits and Costs

Few development experts will claim that one of the four models is best in all cases. Rather, each project's needs and resources will determine the optimal approach. Evolutionary Development, the most complex of the models, yields the most benefit, but also the most cost. In moving to a more complex model, one must tradeoff the benefit with the associated cost.

When moving from the bottom to the top of the chart, earlier feedback allows teams to make more changes to better meet customer needs. Also, feedback helps teams respond more quickly to changes in the technology or the marketplace, such as moves by competitors. Therefore, moving up the chart gives a project *higher probability of building "the right" product.* However, this benefit comes at the cost of *higher effort in managing the user relationship*. To serve this effort, groups usually designate a user liaison to manage such relationships.

When moving from left to right on the chart, we find that increasing the number of delivery cycles yields a greater *visibility of progress*. It also gives a project more *manageability* in terms of reducing integration risk and making breakages visible sooner. Yet, the cost incurred is in the increased *effort in managing cycle boundaries*. This means that someone, often the project manager, must spend many extra hours each cycle in planning and rescheduling.[7]

The figure below graphically displays these benefits:

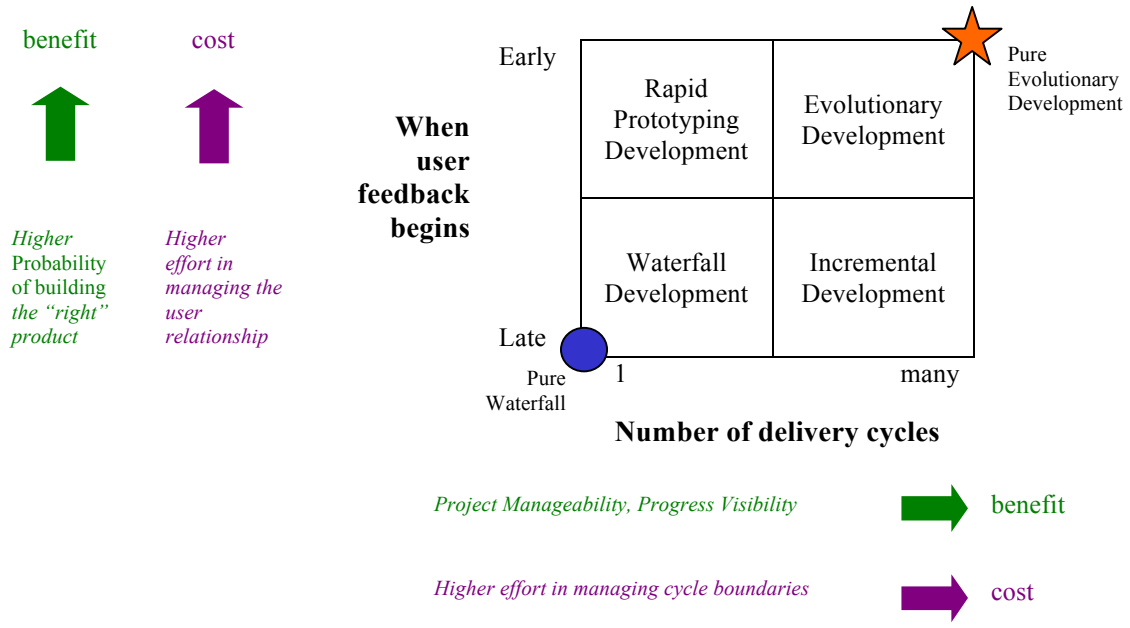


Figure 7: Model comparison matrix with benefits

[7] May and Zimmer, p.6

3 Research methodology

3.1 Creation of a past project matrix

By working as an intern in PGC, I had access to data from all of the projects that hired PGC to assist with their Evolutionary Development process. Most of the knowledge about the process still existed in the heads of the consultants and the project managers they assisted. However, some of the consultants did compile a matrix of past projects in 1995, when PGC last examined the process for critical success factors. The matrix contained only around 10 projects as row headings and a handful of column headings, such as *cycle length, factors that occurred during the project,* and *key learnings*. My first task was to expand this matrix to contain more projects and enough additional fields, such that conclusions could be drawn.

Counting the projects between 1995 and 1998, the total number jumped to 30. I also added around 20 more fields to hold data about the projects that might be relevant. The first fields that were added reflected success. Because success of the project was different than success of the Evo process, we used two separate fields to track success. The rest of the fields were chosen, based on which variances between projects had a likely chance of influencing the success of the Evo process. (See *Appendix I: past project matrix description* for a list of column headings.)

Next, filling in the blank matrix proved to be a challenge. Many of the PGC consultants were still around and could be questioned about the projects they assisted. However, it was often the case that certain data about the project was forgotten, since it happened many years before, or that the data was never known. Also, the project managers themselves were somewhat difficult to contact. Several had left the company. Others, I was instructed by PGC not to contact, since they were potential future customers that should not be troubled.

Teams also did not often perform retrospective analyses of Evo's effectiveness. Instead, they would deem the process successful or unsuccessful and then move on. Evo is seen by most teams as either "good" or "evil", instead of being a model, whose implementations are the cause of success or failure. Hence, few written analyses of the projects were made, and it was somewhat common for project managers to forget why the process was effective or ineffective. Hence, only about 50-60% of the entries in the matrix were filled.

Only a few conclusions could be made from aggregating the data in the matrix. (See *Appendix I: past project matrix description* for conclusions from aggregate data.) Because there were so many process variables that teams "tweaked", a variety of team and project types, and incompleteness of data, there was not enough data to make quantitative conclusions. (See Section 7.1 for quantitative study ideas.) However, the project matrix was effective for reference during dialogue about past projects and it is a good tool to collect future data.

3.2 Creation of consulting models

Before conducting case studies, much time was spent creating consulting "models". A model, in this sense, is a short (usually one page) drawing or slide that represents an abstraction of the process. The purpose of these models is to convey an idea and to promote dialogue about the process. An example of such a model is the 2 x 2 matrix in chapter 2.

The models were used for discussions with consultants and process experts. Often these discussions would lead to the creation of new models. The models that were most liked were kept and evolved into more detailed slides. The interest in these models helped me decide which aspects of the process people would most like me to study.

3.3 Case studies

Four case studies were performed, from which conclusions were derived. Each case was based on a previous Evolutionary Development project at HP. Interviews were conducted with team members, and questions addressed many topics related to the implementation of the process.

Questions asked during the interviews fell into 5 categories:

1) Project and process concepts and goals
2) Team background
3) Software architecture
4) Handling of Evo cycles
5) Handling of user feedback

Category 1 was intended to address the initial organization phase of the project. Most questions concerned the selection of the Evo process, how the process related to the project goals, and how the goals were communicated with the team. Category 2 addressed mainly team experience in relation to the product and process. Both of these sections consisted mainly of short, quick questions to get background information that can be compared with other projects.

Categories 3-5 contained the bulk of the interview. Most questions were open ended, to inspire anecdotes about how well or poorly something worked in the process. Category 3 attempted to characterize the amount of focus the group spent on their architecture, and in particular, how much of it was created up front, versus in the cycles. Category 4 addressed the managerial issues that occurred in managing short, repetitive development cycles (e.g. order of development, handling schedule slippage, etc.) Finally, category 5 addressed issues related to gathering user feedback (e.g. how often gathered, who were the users, etc.). If the project did not gather user feedback, this time was spent focussing on hand-offs with other teams.

Selecting candidates for case studies proved more difficult than was originally thought. First, projects who were considered to have "process failures" were ruled out. It was thought that only one or two major factors caused such failures, and most of these factors had already been determined by PGC consultants or the development teams. Also, groups generally have cultural aversions to talking about past failures. Therefore, it was concluded that more would be gained by talking with teams that had mostly success with the process.

Next, many of the 30 past projects were ruled out because the teams were no longer available. Since the project manager typically takes the lead on installing the new process, teams whose project managers had left were immediately excluded. Lastly, several more teams were too busy or not interested in helping to create a case study. A couple teams were in the middle of "crunch mode" at the time or had recently completed analysis of their process and did not wish to spend a few hours with interviews.

The field of 30 was whittled down to a handful and 4 studies were conducted. The first two were conducted on-site, by interviewing 5-6 team members for an hour each. In each case, the interviewees consisted of the project manager, technical lead, user liaison (usually an engineer), and some developers. Interviews for the third case were conducted via telephone with the project manager and technical lead.

This proved effective, since many of the focus areas were narrowed down after completing the first two studies. The fourth and last case was conducted via a series of emails with a project manager overseas.

The four teams had some similarities, by the nature of their relationship with PGC. Because of the variety of products at HP, software groups come in all shapes and sizes. Yet, most of teams that contact PGC for assistance have between 8-30 developers. (Teams with fewer than 8 members have less of a budget for consultants, while very large teams generally hire someone full-time to handle their processes.) Also, all four products were attached to hardware, and thus had to work closely with their corresponding hardware development teams.

The case studies were written up each as 3-4 page documents, with common sections, so that consultants and experts could easily compare them. Their full texts are presented as appendices (cleansed for external use) and short synopses appear in chapter 4.

3.4 Weekly meetings

Because an "evolutionary" approach was used to generate conclusions for this thesis (we were "eating our own dogfood"), the weekly meetings with PGC consultants and other experts were critical. In an effort to "practice what we preach", in-progress deliverables for the research project were presented incrementally, every two weeks, along with a progress report. User feedback was attained mostly from PGC consultants, some of whom will use the consulting models that were created. Evo team meetings were held weekly and 1-on-1 meetings were held with each of them bi-weekly on average.

Thus, weekly meetings were held with an "Evo team", consisting of PGC consultants whom had expertise related to the process. Team members were:

Bill Crandall – Intern mentor, some consulting experience with Evo
Nancy Near – consulted on many Evo projects
Guy Cox – social anthropology background, some consulting experience with Evo
Ruth Malan – architecture expert
Derek Coleman – architecture and requirements expert
Todd Cotton – consulted on many Evo projects

Other experts were contacted less frequently, but provided valuable feedback:

Tom Gilb – creator of Evo (contacted every 2 months)
Michael Cusumano – renowned researcher of software development, thesis advisor (contacted monthly)
Elaine May – originated Evo usage at HP, currently a Lab Manager at HP (contacted monthly)

3.5 Conclusion of best practices

Most of the best practices presented in chapter 5 were concluded from discussions related to the case studies or projects in the past project matrix. When Evo team members agreed that a practice that a group performed or neglected to perform impacted its success, the practice was generally included in the list of best practices.

For each non-obvious recommendation in the best practice section, sources are listed in brackets, following the recommendation (except when literature is footnoted). An explanation of how the source relates is included if relevant. Sources are from one of the following:

1) Past project data
2) Case studies
3) May and Zimmer, a paper based on a previous HP study on Evolutionary Development (footnoted)
4) Relevant literature (footnoted)
5) Recommendations from interviews with experts

As a caveat to the reader, please note that the best practices are not based on enough data to be statistically conclusive. However, each of them has passed a panel of experts in Evo, including most of the people listed above, in section 3.5.

The best practices were stored and presented to individuals in the form of a consulting model. The model started as a single page graphic, which separated the process into 3 phases, chronologically. Best practices were added into their corresponding phase, and the model soon expanded into a 7-page slide set. To house additional notes and ideas that could not be classified as best practices, a 20 page slide set was created which contained a page or more on each of the subcategories in the best practice model. These notes were called heuristics. In general, the best practice model describes the "what" of the process (i.e. what to do), and the heuristics answers the "how" (i.e. how to do it). Both of these slide sets were then combined to create chapter 5.

3.6 Pilot test of best practices

Finally, to validate the compilation of best practices, a pilot test was performed. The slide set was presented to a couple members of a team that was considering using the process for the first time. The team claimed that the material would be very valuable, yet they could not afford consulting services for their project.

However, as PGC consultants continue to assist HP groups with the process, they will use the list of best practices and attempt to validate its usefulness and correctness.

4 Overview of case studies

The approach to analyzing best practices for Incremental and Evolutionary Development involved creating four separate case studies of evolutionary projects at Hewlett Packard. The studies were created via interviews with between one and six team members, retrospectively. Two of the projects had completed within a few months of the study, while the other two completed two to three years prior. All four groups were software development groups for an end product that included both hardware and software.

(Even though all of the teams refer to their processes as "evolutionary", some teams did not collect significant user feedback, and thus were using Incremental Development, instead.)

4.1 Deep Freeze

The Deep Freeze project is a clean-cut example of a successful transition from Waterfall to an incremental process. The product was the software component for a semiconductor test system. Management clearly defined the risks of the product up-front. There was significant risk in meeting the target performance and also in meeting the development schedule. A cyclical development process was used mainly to make progress visible, in order to manage project risks. The team decided not to incorporate user feedback into the process.

The seven software developers were well experienced, yet had never used an incremental process before. Since they were required to write progress reports at the end of each cycle, many developers were not initially fond of the new process and its associated overhead. However, when they realized the importance of the reports to management, they soon regarded them as a trivial routine task.

Since there was a high schedule risk, low priority functionality was planned for the later cycles. In this way, these later cycles acted as a schedule buffer, in case the schedule slipped. Some of these features were indeed pushed off to the next product iteration, so that the schedule wouldn't slip.

The final product had phenomenal market success. It released within 2 months of the original target (total project length was 18 months), while the competitor's product was a whole year late. The group has since then, used the process in much the same way.

4.2 Redwood Firmware

The Redwood Firmware group, which developed firmware for a printer, was also successful with their adoption of the Evo process, even though they were part of a much larger development team. The firmware team had to interface with both the hardware and software teams for their product.

The software team was also using an Evo process for the first time. However, while the firmware team used 2-week cycles, the software team chose 3-week cycles. In the end, this didn't pose much of a problem, since the two teams were not very reliant on each other's releases.

Managing interactions with the hardware team proved more of a challenge. The hardware team was not using an Evo process, but the team was very dependent on new firmware releases. Since the firmware from the previous product was "broken and rebuilt from scratch", the hardware team could not use some of its basic functionality during the early phases of the project. Although, the decision to build the

firmware from the ground up had little to do with the firmware team's Evo process, the hardware team associated Evo with these problems, and subsequently formed a disliking of the process.

Team members also noticed that Evo requires developers to use their planning skills more than before. New engineers are notoriously optimistic when planning schedules, until they become more familiar with development efforts. On Redwood, there was often slippage due to overoptimistic estimations. However, by the end of the project, the accuracy of estimations had improved.

The firmware team met most of its goals and was very satisfied with using Evo. (Market success is too premature to determine.) One of the largest impacts of the process was that by forcing the team to do much of their detailed planning up front, many critical dependencies were determined. (The first 5 cycles were planned in detail before cycle 1 began.) However, the project manager said that the Evo process required much more of her time than previous processes.

4.3 Casper

The Casper team was developing the latest product in a line of Network Protocol Analyzers, the first of which to have a GUI. Evo was originally considered as an effort to improve quality up-front in the project, however, the team soon realized other benefits.

Since the team was releasing a product every 2 weeks, their progress was very visible, and they received much attention from management and marketing. Also, the incremental process allowed them to avoid the "big crunch" which often occurs at the end of projects.

Marketing personnel were utilized to give user feedback on the product. This worked well for getting feedback on the GUI, since there are always many usability issues of new interfaces. However, the team did not get feedback from any customers until late in the development process. At that point, demand for a key feature was realized, but it was already too late to implement it. Although the marketing team members were cheaper to use, they did not have the same knowledge of customer needs that actual customers had.

The project ended up meeting most of its goals and concluded that the new process was beneficial. Market success of the product, however was less than expected, due in part to the exclusion of the key feature. Team members enjoyed the visibility that the new process created. Yet, just as with Redwood, the project manager found that the process increased her workload.

4.4 Bistro

The Bistro team, which developed software for a printed circuit board tester, had mixed success with its incremental process. The project employed 23 software engineers, split into 2 main teams, with a project manager for each team. One of the teams focussed on throughput speed, while the other implemented a radical new feature. The new feature had a high degree of technology risk, and required several mathematicians to help with the development.

The group is one of the earliest adopters of Evo at HP, and has been using some form of the process for 8 years. Although previous projects have involved significant user feedback, Bistro used very little of it. The reason is that their customers are highly risk-averse and are usually reluctant to use a product that has not been thoroughly tested. (Some groups find that this is an encouragement to have Evo users, since the product can then be marketed as having been used longer.)

A remarkable disappointment with the project was the slippage in schedule of the new feature. Its schedule slipped so much that it was pushed off into the follow-on product. Several redesign efforts were necessary in the feature's development, each of which set it back by more than a month. Team members suggest that more up-front design effort should have been used for the feature.

On the upside, the performance goals of the project were met, and the team thought that their process was successful.

5 Concluded best practices

5.1 Phase I: Attend Ground School

Much like military pilots spending time in the classroom, development teams must spend some time in meeting rooms learning about the process, deciding how to tweak the process to meet their needs, and satisfying some entry criteria for the process. It is important that all affected team members, including project managers and oftentimes marketing personnel attend, to assure that the process meets everyone's needs.

Understand Evo philosophy

- ✓ Recognize the benefits and costs of frequent iteration and early & often user feedback
- ✓ Learn philosophy by comparing approaches and results of Evo and non-Evo projects

Since Evo requires expending some additional effort, it is extremely important to educate the entire team about the process, so that everyone will realize what value is achieved from the extra effort. Also, since there are many ways to implement the process in order to achieve different benefits, teams need to be aware of the "knobs" in the process, and what results are achieved by turning them.[8] (See section 2.3 for major process knobs and their associated costs and benefits.)

Many teams confuse Evo with the "milestone approach". Milestones, like evolutionary approaches, help avoid the "80% syndrome", where developers often incorrectly estimate that a task is 80% complete.[9] Yet, Evo has the additional requirement that a working product be delivered each cycle to measure progress and usefulness. If a working product (or partial product) is not delivered at the cycle boundaries, then user feedback cannot be given and progress cannot be easily measured.

An important heuristic to follow is to implement software "depth first", or one feature at a time, instead of "breadth first", or developing them all simultaneously. By developing depth first as much as possible, it will be easier to release a working product early with minimal functionality and it will be easier to smoothly distribute the release of new features.

One should also note that Evo is an "organic" approach, and thus its philosophy differs from "mechanistic" approaches. For instance, Waterfall Development, a mechanistic approach, assumes that requirements are well known and unchanging from the start. However, Evolutionary Development assumes that requirements are only partially known and will change during the project.

Develop product concept

- ✓ Translate customer needs into product concept that captures key features, quality attributes, and components
- ✓ Have a value proposition that answers the question, "why will customers buy your product vs. the competition's?"

---

[8] Past project data shows high correlation between the team clearly understanding the Evo value proposition and success with the process.

[9] Michael Cusumano and Richard Selby, Microsoft Secrets, Free Press, 1995, p. 277.

Creating the original concept for the product is not a trivial task. If basic characteristics of the product are not outlined at the start, defining proper project goals and creating a software architecture will be difficult. Even though these characteristics may change slightly during the development process, it is always best to set an initial target. A clear value proposition will also help guide prioritizing and decision making, and will make it easier for users and developers to understand why some changes are approved and others are not.[10]

Much research has been performed in the area of product concept development. *Concept Engineering*[11], a process developed by the Center for Quality Management, deals with clarifying the "fuzzy front end" of the product development process that precedes detailed design and implementation. Also, Crossing the Chasm[12], and related work by Geoffrey Moore, deals with value propositions.

Set project goals

- ✓ Set quantified targets for results (schedule, budget, market share, profit, revenue)
- ✓ Prioritize them
- ✓ Communicate them to the team and stakeholders

Once the product concept has been created, project goals can be set. Goals should be quantified[13], prioritized, and communicated to the team and stakeholders. Since the Evo process requires multiple people to make decisions often, alignment on project goals will aid the decision making in the project.

The most common method of setting team goals and making them public is by creating a compelling project vision, and then putting it on a banner on the wall. However, quantified targets are seldom posted on the wall for proprietary reasons. Yet it recommended conveying such information to key stakeholders by whatever means is appropriate.[14] (Some groups, for example, use password-protected web pages to post such data.) The most common stakeholders for projects are:

- developers
- customers
- marketing
- sales representatives
- test team
- dependent teams (e.g. hardware team)

---

[10] May and Zimmer, p. 6.

[11] Center for Quality Management, Concept Engineering, 1996.

[12] Geoffrey A. Moore, Crossing the Chasm, HarperBusiness, 1991.

[13] Personal interview with Tom Gilb, 8/1/98.

[14] Casper did not have an explicitly stated vision and each had conflicting motivations about whether to release the project early. Redwood Firmware and Deep Freeze both had explicit vision statements and did not have such problems.

Most project managers use an informal process to create goals before the requirements creation phase. However, Tom Gilb, creator of Evolutionary Development recommends a more formal process, such as his planning language, *Planguage*[15].

Identify risks and dependencies of project

- ✓ List the primary risks expected from product, process, people, and technology
- ✓ Consider how Evolutionary Development can help overcome these risks
- ✓ List key dependencies

Listing the project's major risks and dependencies will make explicit to the whole team the things that the experienced members are worrying about. In that way, everyone can help navigate through these murky waters. [16] Some common risks are schedule risk, uncertain product requirements, feature creep, new team member risk, and new technology risk. Common dependencies often involve deliveries to hardware groups, sub-teams, or trade shows.

In most cases, Evolutionary Development will help to reduce a project's overall risk. The types of risk that Evo helps mitigate are:

- schedule risk
- customer acceptance risk (e.g. from unknown requirements)
- component integration risk
- team motivation risk

However, one must keep in mind that introducing Evo for the first time creates new risks. Some commonly encountered new risks are:

- acceptance of the process by team
- managing delivery to dependent teams
- feature creep
- difficulty in obtaining early users
- testing neglect during the cycles
- temptation to skip up-front architecture and design work

Create plan for "Flight Plans" stage

- ✓ Define evolutionary plan for gathering initial requirements, defining high-level architecture, creating development environment and preparing for development

The next phase of the process, "Make Flight Plans", can be performed in an evolutionary fashion by breaking down each of the tasks into sub-tasks and creating frequent checkpoints. For instance, architecture can be broken down into first setting the principles and style and then secondly breaking down the system into components. Creating sub-tasks and checkpoints allow for easier progress monitoring and allow dependencies to become more recognizable. [17]

---

[15] Tom Gilb, (see http://result-planning.com)

[16] Redwood failed to identify the dependency between the hardware and firmware teams. Bistro's schedule slipped dramatically from high technology risk.

[17] Source: PGC experts.

While the process gurus usually recommend such cycling for the middle stage, some project managers believe otherwise. Hence, it is important to recognize the pros and cons and formulate a decision for your own team. "Investigation phases" are generally somewhat amorphous with developers not yet in the mode of thinking about deadlines and dependencies with other teammates. Evolutionary planning for this phase will add more structure, and give managers more progress visibility in order to manage a schedule-driven project. However, some argue that this structure is too constrictive for developers and that such detailed planning will stifle new ideas.[18]

5.2 Phase II: Make Flight Plans

Now that the team has completed Ground School, they are ready to start planning for the mission. Several activities need to be performed up-front before implementation begins. The four main activities are (1) requirements, (2) architecture, (3) creating the development environment, and (4) project planning.

Develop initial requirements

- ✓ Identify functional requirements and quality attributes for customers and stakeholders

According to Tom Gilb, requirements definition is the one area where software projects have the most need for improvement. For evolutionary projects in particular, it is important to make requirements explicit especially if the team plans to utilize user feedback to better match the product with customer needs. If so, these requirements will be changed and refined by analyzing the user feedback data.[19]

In my experience, it is rare for groups to keep a formal requirements document that is updated after analyzing user feedback data. The reason for this discrepancy is unknown. However, this is what process consultants recommend.

Requirements definition is an entire discipline of its own, with likely more literature available than all of the lifecycle processes put together. HP customers can contact PGS experts in this field and request related reports. (Contact Derek Coleman.) Another good source is Gilb's Requirements Driven Management[20].

Quality attributes are often referred to as the "ilities" of a project. Examples are usability, reliability, maintainability, etc. These "ilities" can be broken into two categories: real-time and development. Quality attributes can often be best described with use cases. (Contact PGC for further info.)

Develop high-level architecture

- ✓ Develop meta architecture - architectural vision, guiding principles, philosophies, style
- ✓ Develop conceptual architecture - breakdown of system into components with responsibilities and interconnects

Since evolutionary projects' designs have the tendency to change during the project, creating an architecture that is "flexible" enough to anticipate changes is ideal. One way of making the architecture more flexible is to design the *inter*-component architecture up front, and save the *intra*-component

---

[18] The Casper project manager presented this argument.

[19] Source: interviews with Gilb, PGC experts.

[20] Tom Gilb, Requirements Driven Management, (currently in publication, see http://result-planning.com)

architecture until right before implementation for the component. Inter-component architecture can be divided into meta architecture and conceptual architecture.[21]

Specifically, architectures should be designed to allow "depth first" development, where features are implemented serially and plugged in one after another. Although much literature is available in the realm of software architecture, little research is focussed on flexible architectures that allow such serial development of features. Instead, groups are advised to employ experienced and talented architects when appropriate.[22]

HP developers are encouraged to contact PGS experts in this area. (Contact Mike Ogush.)

Create development environment

- ✓ Install source tree, version control
- ✓ Lay groundwork for build process
- ✓ Lay groundwork for unit, system, integration, and regression tests

Creating a development environment that supports Evo is important because with Evo there is "more frequent everything." Builds should be more frequent, so creating automated scripts to handle the build (both working and not working scenarios) is advisable. Also, since testing can be done during the cycles, it is best to create the testing infrastructure early so that it is not neglected.

Many groups in HP do not fully automate their build process. The choice of whether to automate or not is generally based on which method will incur the least amount of development time. Automating the process at the start is sometimes not worth the effort for very short projects with few builds. However, since Evo involves continual additions of small features, rather than a "once a month" plug-in of a subsystem, it is best to build frequently in order to keep up with rapid code changes. Automating the build process will make this easier. [23]

An example of an automated build is one that compiles the entire code set, performs a dead-or-alive test (e.g. printing "hello world"), and sends an email to the build owner telling whether the build worked or not. If the main code base has been broken, a message is sent to the entire team, informing them of the breakage, when the most recent check-ins occurred, and who checked them in. The group that implemented this builds script set it up to run twice daily. By alerting them to problems immediately, they avoided wasted effort in searching for which changes caused the problem.[24]

Cultural incentives can also be created to keep developers from breaking the build. The fear of disrupting someone else's work causes developers to be very careful in testing their code before checking it in. However, some groups create informal policies of small punishments for build breaking offenders. For

---

[21] Deep Freeze split their "up front" and "during the cycles" architecture in this way and found it effective.

[22] Casper created a strong architecture that allowed them to create a prototype easily and interchange components with little effort.

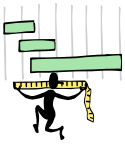
[23] Redwood Firmware did not automate their build process because of hardware difficulties, and later claimed that automation would've helped them.

[24] The author draws on his own software development experience at a group within another company (unnamed).

instance, some groups charge offenders $5 and others require offenders to own the build process for the next few builds.[25]

As far as setting up a testing environment, teams are encouraged to treat "testware" in the same fashion that they treat software. Thus, it should be properly architected, designed, developed in an evolutionary fashion, and tested. Also proper planning and good testing infrastructure should have the goal of keeping developers from neglecting testing during the cycles.

Create Evo plan for "Missions" stage



- ✓ Choose cycle length
- ✓ Decide upon a sequencing strategy for deciding the order of tasks
- ✓ Develop a chunking strategy for decomposing tasks
- ✓ Define a rapid decision-making process
- ✓ Assign Evo-specific roles of Technical Manager and User Liaison
- ✓ Create cycle template
- ✓ Create detailed plan for first few cycles, highlights for rest of cycles

Now, a lot of Evo-specific details need to be handled. The most time-consuming of these is creating the cycle plans. Usually, teams decide to call a large meeting or hold an Evo workshop to hammer out detailed plans for the first few cycles. Then during each "Fly Missions" cycle, teams should hold a planning meeting to determine what should be implemented for the next cycle.

Before creating the detailed plan, however, several other tasks should be completed. First, a cycle length must be determined. Most HP groups use either 2 or 3 weeks and a constant length is always recommended to create rhythm. The cycle length should be based on how often the team desires to view its progress, versus how much it wants to avoid the overhead incurred at cycle boundaries (i.e. builds, cycle reports, getting customer feedback, replanning). One should also note the difference between *backroom cycles* and *frontroom cycles.* Frontroom cycles involve delivery of the product to customers or users, while backroom cycles are all the internal tasks that involve only the team itself.[26] Hence, it is often beneficial to have varying lengths for backroom and frontroom cycles. For instance, when attaining customer feedback is costly a group might set its backroom cycle length to 2 weeks and its frontroom cycle length to 4 weeks.

Next, the group should determine their sequencing strategy. "Sequencing" refers to the order in which you decide to develop the features of the product. Some of the most common strategies for sequencing include:

- Pick highest impact features first
- Let customer select
- Fulfill most important requirements first
- Show some visible progress first
- Show insight into areas of greatest risk first

---

[25] Michael Cusumano and Richard Selby, Microsoft Secrets, Free Press, 1995, p. 271.

[26] Tom Gilb, Evo: The Evolutionary Project Managers Handbook, (currently in publication, see http://result-planning.com), p.52.

- Do step necessary for coordination with other teams first

It is recommended for groups to pick a strategy to determine a default ordering, yet still maintain the freedom to revise the strategy during the cycles.[27]

A chunking strategy is the next task to complete. "Chunking" is the discipline of breaking down development into small, discrete mini-projects. It is a difficult skill and requires decent architectural ability, as well as good project planning skills.[28] Also, object oriented projects using Fusion or a similar process should find that there are synergies with evolutionary projects because of the method of decomposition that it provides.

Defining a rapid decision making process is another key task. Decisions occur more frequently in evolutionary projects. They most commonly involve what to do in the upcoming cycle based on what progress has been made so far. Hence, it is important to explicitly decide up front how decisions will be made during the cycles. If a team typically takes longer than its cycle length to make decisions that stick, some problems may occur and hence the team should work hard to improve its decision making process.[29]

Two Evo-specific roles are usually assigned for evolutionary projects. The Technical Manager is involved heavily and planning and keeping track of progress. At HP, the Project Manager usually takes this role. The User Liaison is responsible for collection of user feedback and keeping users happy so that they will continue to give feedback throughout the project. Marketing personnel are excellent for this role. It is also often beneficial to have a separate liaison for internal and external users.[30]

Next, a cycle template is a good idea to create. Some events should occur regularly in the cycle, such as builds, code freezes, and shipments to users. Thus, for the efficient planning of each cycle's tasks, one should start with a cycle template.[31]

---

[27] Source: PGC experts.

[28] Elaine May, "Dividing a Project", Hewlett Packard (unpublished)

[29] Source: PGC experts.

[30] May and Zimmer, p. 7; One project in past project matrix also suggested that such roles were needed.

[31] May and Zimmer, p. 6.

M	T	W	Th	F
• final test of last week's build • source code frozen	• "ship" last week's build • design and start to implement new features • source code open	• incremental build overnight	•	• weekend build from scratch
M	T	W	Th	F
	• all user feedback collected	• functionality freeze • incremental build overnight	• test new functionality • determine changes for next release	• test new functionality • weekend build from scratch

Figure 8: Example cycle template

(Note that this template is for illustrative purposes. Most groups use more detailed tables that also list task owners and estimated time to complete task.)

Lastly, the group must create a detailed plan for the initial cycles. Approaches vary, but many groups have found it best to create detailed plans for the first 5 cycles or so.[32] The disadvantage to planning the rest of the cycles up front is that the plans are very dynamic, and are likely to change if items take longer or shorter than expected. However, it is usually best to plan ahead for more than one cycle in the future, so that dependencies can be seen between current and upcoming tasks. As the project progresses, the team should get a better feel for how many cycles in advance they would like to plan.

5.3 Phase III: Fly Missions

The team is finally ready to embark on the missions that will ultimately create the finished product. Since every cycle is a mission, teams can concentrate on the current mission and the immediate deliverables for that mission.

Takeoff and land frequently--design, develop, integrate, and test incrementally

✓ Include in each cycle, if appropriate:

1) update requirements and architecture
2) get user feedback on previous cycle's delivery
3) design
4) implement
5) unit test
6) integrate (build)
7) test integrated system
8) check-in
9) measure progress
10) re-plan

Each mission should result in an executable product that can be analyzed by a user (or expert) and tested for quality. This shouldn't imply that each team member needs to accomplish something monumental each cycle, but nonetheless something. Usually, developers will be responsible for steps 3-8. Project managers, technical leads, or appointed individuals should handle steps 1, 2, 9 and 10.

[32] Redwood Firmware found this effective.

Teams often decide to operate some of these tasks every other cycle or every few cycles. Gathering user feedback, for example, is sometimes costly and some groups decide to gather less often. These are critical decisions that should be decided in part by looking back at the explicit goals for the project.

Teams are highly encouraged to both integrate and test in each cycle. In general, new code will be easier to integrate while it is written, as opposed to after it becomes a major subsystem. This implies that it is best to integrate early. Also, testing is much better done sooner rather than later, since new bugs will be easier to attribute with the new code.[33]

Evolve requirements and architecture

- ✓ Modify requirements as appropriate, upon analysis of user feedback
- ✓ Develop logical architecture - detailed component structure, interaction diagrams, interface definitions
- ✓ Develop code architecture - partition code into files

If user feedback is analyzed as a part of the process, requirements should be updated upon analysis of the feedback.[34] The software architecture within components should also be hammered out whenever the new components are created. Intra-component architecture consists of both logical architecture and code architecture.

Logical architecture includes both static and dynamic views. The static view defines the interface, while the dynamic view shows the protocol for using the interface over time.[35]

Keep your eye out the window--get user feedback early and often

- ✓ Get feedback not just on bugs, but also on usefulness of existing features, desired features, look and feel, etc.
- ✓ Identify users that would benefit from early use, really use product, and/or give useful feedback
- ✓ Use "surrogate customers" if no access to real ones, or experts if getting feedback on quantitative measures
- ✓ Prioritize feedback and make changes when appropriate

User feedback is very important for certain products and less important for others. If the team decides to incorporate user feedback into the process, much care must be taken in identification of Evo users and managing the feedback process. Users that do not have an incentive to use the product are difficult to count on. Also, if the team does not incorporate a suggestion without explanation, users may become frustrated.[36]

---

[33] Past project data says that several projects adopted Evo to avoid problems with late integration, implying that they had been burned by it in the past.

[34] Interview with Tom Gilb, 8/1/98.

[35] Deep Freeze saved this portion of the architecture for creation during the cycles, which worked well for them.

[36] May and Zimmer, p. 8.

Using the marketing and sales teams for feedback is usually a good practice. They will often be more engaged than external customers will. Also, since they are usually more directly responsible for customer satisfaction than the development team, involvement in the process can help them achieve that goal. Ideally, marketing personnel can both act as an internal user and be a liaison to external users.[37]

There are several things to keep in mind in order to keep the users engaged. By giving them timely response to feedback, they will feel that their comments are being addressed. Also, when the team decides not to implement a suggestion, they should let the users know the rationale. If not, the users may get easily frustrated. Lastly, the team should share the development plan, as appropriate, so that the users will know what changes are coming, and how the team is responding to the feedback.

Build frequently

- ✓ Build code base as often as your resources allow
- ✓ Prevent breakage

Since the executable must stay "live" throughout the whole development phase, it is important to build the code base frequently. The more frequent the builds, the sooner new code can be tested and utilized, and the sooner that faulty check-ins will be spotted.[38] Teams may want to discourage breakage of the build via cultural incentives, such as requiring the guilty party to own the build process, or to pay a small fine.

Ideally, the team should completely automate the build process. Often, the cheapest resource a team has is overnight machine time. If not already utilized, this machine time should be used to build and test the code in order to prevent new errors from entering the code. If the resources are available, a nightly build is best. Regression testing should be automated, so that the code can also be easily tested at every build.

---

[37] Redwood Firmware and Deep Freeze could've benefited from feedback from their marketing and sales teams.

[38] Redwood Firmware built only every two weeks and believed that they could've benefited from more frequent builds.

Debrief after each mission

- ✓ Identify what worked and what needs to be improved in terms of product, process, plan, and people
- ✓ Reassess progress towards target
- ✓ Reassess risks
- ✓ Implement course corrections

Since results are visible after every mission, schedule changes and design changes are more frequent with the Evo process than with others. Teams should prepare to replan at the end of every mission. Also, in the spirit of continuous improvement, analysis of how things went in the current mission can lead to greater success in the next mission. After the last mission of the project, the team should similarly debrief about how well the process worked and how they can improve it.[39]

It's best to think of problems not in terms of what is wrong or broken, but how great the opportunity for improvement is. Improvements should also be rewarded. Creating a culture of continuous improvement in the group is important.

Lastly, it's important not to procrastinate replanning. If schedule adjustments are neglected, the schedule soon becomes meaningless. Better to expend the effort sooner, than to spend double the effort later.[40]

---

[39] Bistro had not debriefed about Evo in recent years, which led to their engineers not understanding the philosophy of the process.

[40] Past project data lists two projects that experienced this problem.

6 General conclusions

6.1 Process preparation

Initial preparation is required before the cycling can begin. One of the primary concerns that the PGC consultants usually have of their customers is that they are not doing "enough process" to be successful. Specifically, with Evo, since there are dozens of new things to learn, teams often concentrate on a few of them, and may leave some important things out. Many of the tasks that are neglected are the up-front preparatory activities, such as educating the team on the process philosophy and proper cycle planning. Hence, it was decided to present the best practices chronologically, to indicate where to concentrate at a given time in the project. These up-front tasks can actually be separated into two preparatory phases. In the "attend ground school" phase, teams must educate themselves and make clear their goals for the process and project. This phase is relatively quick, compared with the next two. Next, in the "make flight plans", the team needs to develop the infrastructure for the project. After these two phases, the team will be finally ready to "fly missions".

6.2 Process philosophy

Understanding the process philosophy is important. Often, teams follow processes like recipes, without understanding any of the theory behind the process. In the absence of complex conditions, this can still produce a reasonable result. For instance, I have no idea why a zucchini makes a chocolate cake so good, so when I run out of zucchinis, I have no idea which vegetable to substitute. The problem is more severe with Evo, since the process has many parameters. Without understanding the theory, teams will not be able to adapt the process to meet their needs. Therefore, the team should spend time in learning the philosophy behind the process. Since the process affects the entire team, all team members should be educated as such.

6.3 Team competence

Competent teams are more successful with complex processes. It is somewhat an obvious notion that teams who are more experienced with software processes will be better able to adopt complex processes. However, in practice, new teams or new team members attempt to adopt Evolutionary Development without the proper training. If a team has a large number of inexperienced programmers, the team may want to consider using Waterfall Development, for its simplicity. Waterfall Development is considered less complex than Evolutionary Development, which in turn is considered less complex than Spiral Development. If a team using an evolutionary process hires only a few inexperienced programmers, these programmers should be "hand-held" through the process, especially when it comes to planning.

6.4 Creating the "right product"

Ownership of "creating the right product" is often neglected in organizations. Traditionally, marketing has served the function of determining customer needs and product development has served the function of creating the product based on the specification. Usually, it is a joint effort to define the specs, based on the needs. Then the project progresses and only when the product fails in the marketplace do we reexamine the specs to see what went wrong. With today's technology, we've realized the ability to get customer feedback on a product while a product is still in development. The ability to do this is especially great in software, because prototyping costs are low. Yet, in many organizations, neither the marketing team or development team for a product is very interested in shaping the product to meet customer needs. This is most likely due to the fact that neither team is given the job of making the

product best for the user. Organizations need to determine whom to give this task's ownership to, and how to set proper incentives for teams to spend effort on it.

6.5 Motivational implications

Cyclical processes are both encouraging and discouraging. The impact a process has on a team members' motivation is a strong influence on whether the group will successfully adopt the process. On the upside, teams generally enjoy the ability to see the fruits of their labors immediately. On the downside, however, teams may get frustrated from the additional overhead the process requires. Also, when using the process for the first time, groups often find themselves continually adjusting their schedule, which tends to dampen spirits. Frustration over such issues with the process may cause teams to reject the process or blame unrelated problems on it. To counter the negative feelings about the new process, managers should educate the team well about the process and assure them that these effects are somewhat expected, but the overall impact of the process should be positive.

7 Recommendations for future research

7.1 Quantitative study

This study was by in large qualitative. Although many past projects that were referenced, mostly anecdotal advice was gathered from them. Qualitative studies are valuable in that a broad range of ideas and practices are presented. However, in order to sway developers to adopt practices that require major modifications in behavior, rigid proof is generally needed. A quantitative study, by gathering statistical evidence over a larger set of projects, could provide such proof. (However, the major drawbacks of quantitative studies is that they generally cannot say very much, and they often conclude what many people are already sure of, anyway.) Future studies should consider evaluating the successfulness of:

- cyclical lifecycle process
- short cycles (2-3 weeks)
- user feedback during development
- high build frequency (~daily)

7.2 Importance of flexible architectures

The case studies presented here attempt to assess the impact of architecture on Evolutionary Development. However, the flexibility of a given architecture is difficult to measure. When asked how flexible their architectures were, most architects responded that they tried to make their architectures as flexible as possible, and not much more information was forwarded. Thus, a more technical analysis of the architectures is needed. Such a future would require a researcher who is somewhat familiar with the technical aspect software architecture. Then, the proper questions can be asked which would characterize different architectures in aspects related to flexibility.

7.3 Tailoring of the model to specific needs (large teams, new markets, etc.)

Perhaps the most asked question in the realm of lifecycle models is, "which model is best for a given project?" Unfortunately, few companies use more than one model in addition to the Waterfall, so comparing say, Evolutionary Development and the Spiral model would probably require an industry-wide study. However, since there are many variations of Evo at HP, the question that comes up is, "what variation of Evo should I use for a given project?" Since some variations of Evo are essentially other lifecycle models, the two questions are quite similar.

The later question was the original one to be approached by this thesis. It is feasible to believe that one variation of the model works best for large teams, another works best for high technology risk, and yet another works best for products in new markets. We listed about 10 major project attributes and theorized which variations of the process might fit them best. However, our sample of projects proved too homogeneous to make any such conclusions. Most of the HP teams that call upon the PGC consultants for lifecycle assistance are of medium size (8-20 engineers) and are iterations of an existing product.

In retrospect, it would've been best to analyze several projects that did not claim to be using Evo. Many of these projects would likely be following a Waterfall process, or cyclical process similar to Evo. (Often, groups do not give a name to their lifecycle process.) If a case study was performed for each of the major project types, much can be learned about how well their current process works for them. Ultimately, it would be nice to conclude where on the 2x2 matrix a given project should lie.

7.4 Relation to Open Source Software (OSS) method

During the research gathering for this thesis, the famous "Halloween Document" was leaked from Microsoft and grabbed the attention of many of the PGC consultants. (The document assesses the threat that the OSS method presents to Microsoft.) Mark Interrante and I discussed how the process might relate to cyclical lifecycle models, such as Evo.

OSS, in a sense, uses very short cycles, as well as user feedback. Versions of the product change rapidly, with each user's check-ins. Also, the developers become the users, since many of them are motivated to develop in order to make the product easier to use. However, OSS development lacks the drumbeat that Evo has (or rather, in OSS the drumbeat is more silent).

Since the two processes are similar, perhaps Evo can borrow some techniques from OSS development. For instance, one of the conclusions of the Microsoft analysis was that OSS projects can be used to create very high quality software (which is contrary to most people's suspicions). If the attributes of OSS which lead to high quality were discerned, perhaps they could be applied to Evo.

On the other hand, OSS is criticized for not allowing rapid time-to-market, which Evo does allow. Therefore, there might be an ideal process that contains the best of both models. Future research can try to identify what, if anything from OSS can be applied to Evo to improve it.

Bibliography

Ron Albury and Hans Wald, "The Evolutionary Software Development Process", SDRC/Metaphase working paper (unpublished)

Mikio Aoyama, "Concurrent-Development Process Model", *IEEE Software*, July 1993, pp. 46-55

Barry Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988, pp. 61-72

Todd Cotton, "Evolutionary Fusion, A Customer-Oriented Incremental Life Cycle for Fusion", *HP Journal*, August 1996 Volume 47 Number 2

Michael A. Cusumano and Richard W. Selby, Microsoft Secrets, Free Press, 1995

Michael A. Cusumano and David B. Yoffe, Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft, Free Press, 1998

Michael A. Cusumano and Richard W. Selby, "How Microsoft Builds Software", *Communications of the ACM*, June 1997, Vol. 40 No. 6, pp. 53-61

Michael A. Cusumano, "How Microsoft Makes Large Teams Work Like Small Teams", *Sloan Management Review*, Fall 1997, pp. 9-19

Kathleen M. Eisenhardt and Shona L. Brown, "Time Pacing: Competing in Markets that Won't Stand Still", *Harvard Business Review*, March-April 1998, pp. 59-69

Robert G. Fichman and Scott A. Moses, "An Incremental Process for Software Implementation", *Sloan Management Review*, Winter 1999, pp. 39-52

Tom Gilb, Principles of Software Engineering Management, Addison Wesley, 1988

Tom Gilb, Evo: The Evolutionary Project Managers Handbook, (currently in publication, see http://result-planning.com)

V. Scott Gordon and James M. Bieman, "Reported Effects of Rapid Prototyping on Industrial Software Quality", *Software Quality Journal*, Vol. 2. 93-108, pp. 93-108

D. Greer and D.W. Bustard, "Towards an Evolutionary Software Delivery Strategy based on Soft Systems and Risk Analysis", *IEEE ?*, September 1996, pp. 126-133

Todd Lauinger, "Risk Driven Iterative Development", *Object Magazine*, April 199?, pp. 29-34

Alan MacCormack, "Towards a Contingent Model of Product Development: A Comparative Study of Development Practices", Source: EIASM 5th International Product Development Management Conference, Lake Cumo, Italy, May 25-26, 1998, pp. 653-670Marco Iansiti and Alan MacCormack, "Developing Products on Internet Time", *Harvard Business Review*, September-October 1997, pp. 108-117

Elaine L. May and Barbara A. Zimmer, "The Evolutionary Development Model for Software Development", *HP Journal*, August, 1996, page 39

Elaine May, "Dividing a Project", Hewlett Packard (unpublished)

Steve McConnell, Rapid Development, Microsoft Press, 1996

Johanna Rothman, "Iterative Software Project Planning and Tracking", *Software Quality*, July 1998, pp. 1-8

Nancy Staudenmayer and Michael Cusumano, "Alternative Designs for Product Component Integration", Sloan working paper #4016, April 1998 (unpublished)

Roberto Verganti, Alan MacCormack, and Marco Iansiti, "Rapid Learning and Adaptation in Product Development: An Empirical Study of the Internet Software Industry", Source: EIASM 5th International Product Development Management Conference, Lake Cumo, Italy, May 25-26, 1998, pp. 1062-1079

Appendix I: past project matrix description

Rows and columns of the matrix

- Division
- Project name
- Product
- Does organization still exist?
- Division Contact
- PGC Contact
- Date of Project
- Project Status (done, on going, or cancelled)
- Project Success?
- Evo Success?
- Project size
- Project Focus (SW, FW, HW or system)
- Cycle Size
- # of Cycles Completed
- Variation of Cycle Length
- Evo Value Prop clearly understood?
- Team experience with Evo (1st, 2nd, 3rd time using?)
- Team used Fusion or SWI Architecture processes?
- Objective of use of Evo
- Sponsorship and motivation
- How was cycle length determined?
- Use of Customer Feedback
- Factors that occurred during the project
- Chunking Strategy, Sequencing strategy
- Was Entry Criteria identified, satisfied?
- Special Roles
- Barriers to successful Evo adoption
- Key Learnings

Notes on the data

Gray boxes signify unknown data. Most data was collected during 6/98 - 8/98 from SWI account managers. When an account manager did not remember much data from an Evo project, the project manager (PM) was contacted in many cases. However, there are many PMs that are no longer with the same group and some that have left HP, and thus some data could not be determined.

Both project and process success are somewhat subjective, based on the opinion of either the account manager or project manager.

Conclusions from aggregate data

- 28 projects included from 1990-1998.

- 12 successful projects, 5 failed projects, 11 currently unknown.

- 9 projects successful at using Evo, 4 projects unsuccessful, 15 projects currently unknown.

- High correlation between Evo success and understanding the Evo value proposition. 11/15 (73%) successful projects understood the Evo value proposition clearly, while 4/6 (66%) unsuccessful projects didn't understand it clearly.

- Range of cycle lengths is 1-6 weeks, average of 2-3.

- Range of total number of cycles is 3-65, average of about 10.

Appendix II: full texts of case studies

(See attached sheets)