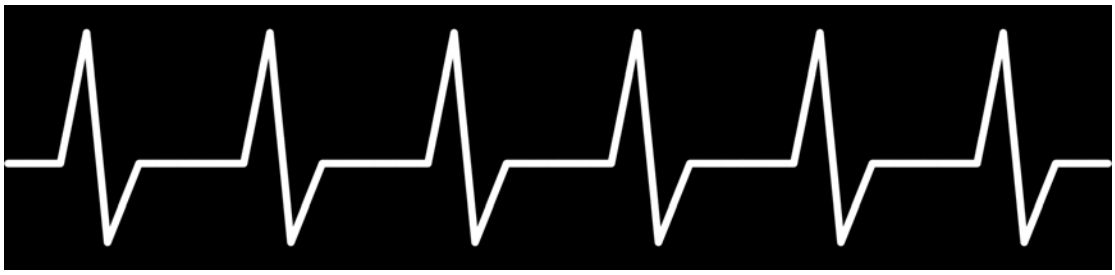
Helsinki University of Technology Software Business and Engineering Institute
Technical Reports 3

Teknillisen korkeakoulun ohjelmistoliiketoiminnan ja –tuotannon laboratorion
tekninen raportti 3

Espoo 2006 HUT-SoberIT-C3

PACING SOFTWARE PRODUCT DEVELOPMENT:
A Framework and Practical Implementation Guidelines

Kristian Rautiainen Casper Lassenius (eds.)



TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Software Business and Engineering Institute
Technical reports 3

Teknillisen korkeakoulun ohjelmistoliiketoiminnan ja –tuotannon laboratorion
Tekninen raportti 3

Espoo 2006 HUT-SoberIT-C3
2nd Printing, Electronic Edition
http://www.soberit.hut.fi/sems/PacingSoftwareProductDevelopment.pdf
ISBN 951-22-8382-4

PACING SOFTWARE PRODUCT DEVELOPMENT: A Framework and Practical Implementation Guidelines

Contributing authors (in alphabetical order):

Juha Itkonen
Mika Mäntylä
Kristian Rautiainen
Mikko Rusama
Jari Vanhanen
Jarno Vähäniitty

Edited by Kristian Rautiainen and Casper Lassenius

Helsinki University of Technology
Department of Computer Science and Engineering
Software Business and Engineering Institute

Teknillinen korkeakoulu
Tietotekniikan osasto
Ohjelmistoliiketoiminnan ja –tuotannon laboratorio

Distribution:
Helsinki University of Technology
Software Business and Engineering Institute
P.O. Box 9210, FIN-02015 HUT, Finland
Tel. +358-9-4511
Fax. +358-9-451 4958
E-mail: reports@soberit.hut.fi

Contents

List of Figures

List of Tables

Preface

Software product business is big business — and it is rapidly growing even bigger. The companies producing and selling packaged software, i.e., *software products* that are developed once and then sold "as is" to a large number of customers constitute the most aggressively growing segment in the whole software industry. In 2002, the world markets for packaged software was estimated at 184 billion USD. In Europe, the market value of the software product industry in 2002 was estimated at 54 billion €. It was the fastest growing segment in the Western European information and communication technology markets. The annual growth for the software product industry has been over 10% during the last decade, and the rapid growth is expected to continue.

In Finland, the software product industry is still young, immature, and economically quite insignificant. Most companies are young and small, and are struggling with the challenge of developing highly productised pieces of software for international markets. The overall value of the Finnish software product industry has been evaluated at 1 billion € in 2002, with a potential to reach up to 8 billion € by 2010. Of the total turnover, about 40% comes from exports.

Despite the economic importance and promising outlook of the software product business, the software engineering community has been slow to react to the specific needs of companies doing software *product development* as opposed to customer projects. Most existing models, such as the CMM and most standards, share two common traits that are challenging from the point of view of adopting them in Finnish software product businesses. Firstly, they have been developed from a large company perspective, and often require considerable resources to implement. Secondly, they are customer-project focussed, not market focussed. In our experience, these two points make the direct adoption of existing models hard, and simply "downscaling" them does not work. In addition, the link between business considerations and the software engineering aspects is extremely weak.

This book documents the result of a three-year research project at the Software Business and Engineering Institute at Helsinki University of Technology. The project, SEMS, aimed at understanding the particular challenges of product development in the software industry, and to develop an approach for companies needing to tackle those challenges. The end-result has been structured into an overarching framework that emphasises the importance of *development rhythm* as an overall approach for organising software product development. The framework identifies six focus areas that we have found important when tackling the challenge of organising product development in a small software company.

Despite the lack of applicable models in the classic software engineering literature, we have been able to draw upon several other streams of research. The work by Cusumano and Selby, who have extensively discussed how Microsoft, the world's most successful software product company does business and develops products has

been invaluable in our efforts. The development and popularisation of software development models for small teams working in turbulent environments, *agile development approaches* have provided a good understanding of how small teams can organise their development efforts. In particular, we have found the Scrum approach good for structuring development on various time horizons. Finally, the existing work on product development processes and product strategy in the new product development management literature has provided us with basic concepts and models that we have, to the best of our ability, tried to adopt to the needs of small software product companies.

The book consists of an introductory chapter that presents the overall framework, the *Cycles of Control*, as well as introduces the six focus areas: *commercial product management, pipeline management, software design and implementation, quality assurance, technical product management,* and *development organisation*. Of these, all but the last one are described in more detail in the subsequent chapters. Instead of being confined to an own chapter, the organisational aspects are discussed throughout the book.

The chapters have been written by the various members of our research group, according to their areas of speciality. We hope that the resulting differences in expression do not feel too distracting to our readers. Finally, it is worth pointing out that the book has been written with practitioners in mind. Academic readers, in particular, might find that referencing is scarce, and many, even important references, have been left out for readability reasons. Also, due to the focus on practitioners, the argumentation might not stand strict academic scrutiny. The book is probably not suitable for use as the sole textbook on academic courses, since it assumes familiarity with many software engineering concepts, and does present a somewhat narrow view of the field of software product development. However, it might be useful as additional reading on a software engineering course dealing with the particularities of software product development.

We do hope that you enjoy reading the book and that you find some value in the proposed models and practices. We welcome any feedback you might have — please send it to `sprg@soberit.hut.fi`.

Espoo, 14.4.2004

Casper Lassenius
Kristian Rautiainen

Acknowledgements

This book summarises the lessons learned in the SEMS (*Software Engineering Management System*) research project of the Software Business and Engineering Institute (SoberIT) at Helsinki University of Technology. During 2000-2004, SEMS studied how software engineering should be managed in small product-oriented software companies.

The SEMS project was funded by Tekes and a number of small and medium-sized software companies. Our industrial partners (in alphabetical order) were Avain Technologies, Arrak Software, Bluegiga Technologies, Intelligent Precision Solutions and Services, Mermit Business Applications, Napa, Oplayo, Popsystems, QPR Software, Smartner Information Systems, Smilehouse, and Softatest. In addition to funding, our industrial partners have allowed us to test our ideas in practice, exchanging insights and experiences. Because of your contribution, this book is written *for* the practitioners *by* the practitioners with us as the scribes – our warmest thanks!

An important thank you goes to our colleagues at SoberIT for sparring us during the years and providing fresh viewpoints. Also, we would like to dedicate extra thanks to the members of the SoberIT support team for running things smoothly, thus making our lives easier.

With the very best regards from the SEMS team,

Juha Itkonen
Casper Lassenius
Mika Mäntylä
Kristian Rautiainen
Mikko Rusama
Reijo Sulonen
Jari Vanhanen
Jarno Vähäniitty

Chapter 1

An Introduction to the SEMS Approach

Kristian Rautiainen and Jarno Vähäniitty

1.1 Motivation

Managing software product development is challenging but doing it well can be extremely rewarding. Profits from duplicating a product to thousands or millions of customers can be both luring and elusive. Statistics suggest that up to 50 % of companies founded in any one year are not in business five years later due to inadequate management. Success in the product business demands more than just succeeding in individual development projects. Shipping products at the right time, hitting windows of opportunity with the right set of features over and over again is at least as important. However, in the software product industry, time-to-market is constantly shrinking and technologies evolve at a furious pace. If a company tries to keep up with this pace and react to every change in its environment, it would not have time to do anything but react. The developers quickly go crazy with the indecision of the managers and the constantly changing product requirements. The key lies in striking the right balance between flexibility and control that serves both business and development needs.

Achieving this balance, however, is no easy task. For small companies — with less than 50 employees — which constitute the majority of software product businesses, it is particularly challenging. Many of these try to succeed in the product business, while at the same time doing customer projects to maintain cash flow. This leads to internal chaos, with people trying to do too many things at once. Projects exceed their budgets and schedules and only heroic efforts from individuals keep the projects going. Understanding the software process and using good practices might help, but everyone is too busy to stop and figure out what and how things could be done better. It is like the running man carrying his bicycle: he is too busy to stop, mount the bike and then pedal away.

The man carrying the bike has it easy compared to most small software product companies. At least, he only has two simple choices to choose from. For the software companies a myriad of process models, methods and practices exist that could help improve development performance. However, as Frederick Brooks Jr. puts it, there is no silver bullet, no magic methodology that can solve all your problems. Choosing and tailoring processes and practices is difficult, especially since most pro-

cesses and practices have been developed for and tested in large companies. For small software product companies operating in turbulent environments, so called *agile processes* might be a good starting point. They have been designed for small teams and projects facing a lot of uncertainty. They provide a set of values, principles and practices that enhance flexibility and help you embrace change. However, they do not provide any solutions to the particular aspects of the software product business. Business considerations should be taken into account and provide controlling aspects to the development process. A familiar example of this is Microsoft, a company that has been able to dominate certain markets for a long time, despite many people raising the issue of the bad quality of its products. Microsoft's success is based on deliberate strategic business decisions and a development process to back them up. Even though Microsoft is a large company, its principles can also be applied in small companies.

To sum it up, in order to successfully manage software product development in small software product businesses in turbulent environments, a holistic approach combining business and development aspects and providing a combination of control and flexibility is needed. Such an approach has been developed in the SEMS research project at the Software Business and Engineering Institute (SoberIT) at Helsinki University of Technology (HUT). The rest of the book introduces the SEMS approach as well as presents examples of its application.

1.2 The SEMS Approach

Software product development consists of many viewpoints and activities that have to be coordinated and managed for good results. In Figure 1.1 we summarise 3 viewpoints and 6 key areas of activities we have found important for managing software product development. Put together, they form the Software Engineering Management System (SEMS) of a company.

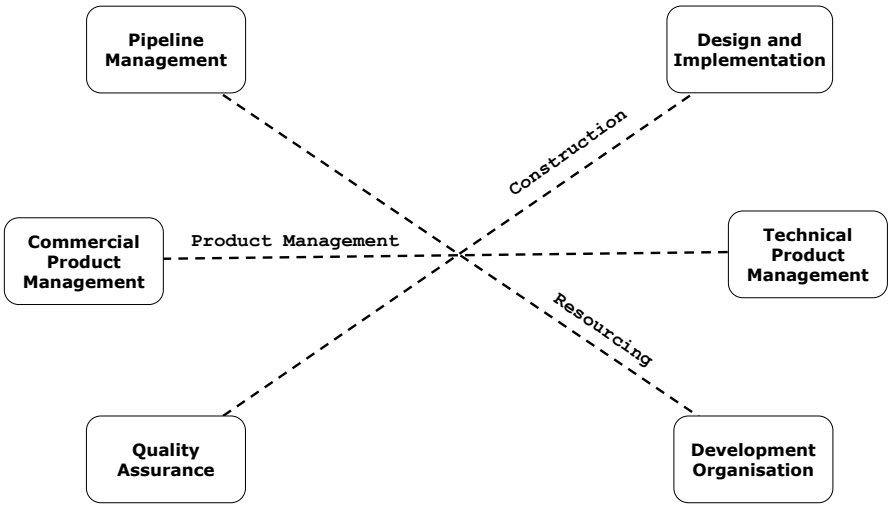


Figure 1.1: Components of a Software Engineering Management System (SEMS)

The three axes in Figure 1.1 represent *viewpoints* of software product develop-

ment management. We need to manage what products are developed and the business rationale behind them (Product Management), how these products are developed (Construction), and by whom and when (Resourcing). The six boxes at the ends of the axes represent *key areas* of activities. Below is a short overview of each of them. For a more detailed discussion, see the later chapters in the book.

1. *Commercial Product Management* is the link to the business. The company's strategic ambitions and the needs of the markets are translated into product release plans. The release plans address details, such as the goals of each release, the main features or requirements to be included, the target audience or market of the product, the role of the release (e.g., major, minor, maintenance) and the release schedule.
2. *Pipeline Management* deals with resource allocation in the product development pipeline. The pipeline consists of all the work done by the development organisation, and could range from developing a product to fixing bugs or installing the product to a customer. This work needs to be prioritised and reprioritised on a regular basis, so that the most important things get done.
3. *Design and Implementation* addresses managing product design or architecture, coding the product, and managing product evolution. You need to figure out what the product design should be like from different perspectives and choose the most appropriate one. You also need to manage product evolution during its life cycle, e.g., deciding when refactoring is necessary.
4. *Quality Assurance* manages building quality into the product and verifying the achieved quality level. Quality Assurance covers both testing activities that address how you verify that the product is of good-enough quality and static activities concerned with all practices that can help you develop quality products. To implement well-working quality assurance, you must establish what good-enough quality means for you and your customers. This includes, e.g., setting quality goals and release criteria and establishing a test strategy.
5. *Technical Product Management* deals with the product releases from a technical perspective. During development you need to manage intermediate builds and different versions of the software, as well as control change. When the product is finished and released, you need to manage the release package that contains more than just the software, e.g., an installation guide and a user's manual. Version control is as important for the release package as it is during development. You need to know what each release package contains, especially if you make some customer-specific installations. Otherwise maintaining the releases may become very challenging.
6. *Development Organisation* is about managing how development is organised. You need to establish the roles and responsibilities of the different stakeholders that contribute to product development. Another aspect is managing the necessary competences of the organisation.

The six key areas above have helped us structure and understand the situation in companies we have worked with, but they are by no means exhaustive, and a different set up and names would surely be possible. We have found looking at software product development through these areas useful in our practical work with companies. The areas form an interrelated whole, and they set constraints for each other. Changes in one area influence other areas and if the other areas are left unchanged, they may prevent the change. To give you an example we use requirements engineering, something you

might have felt missing in the picture. There is a very strong aspect of requirements engineering in commercial product management, because a big part of it is making plans on the content and scope of future product releases. However, our product architecture (design and implementation) may set constraints on what kind of requirements are doable, especially non-functional requirements such as performance criteria or vice versa; non-functional requirements influence the product design choices (design and implementation) and the resourcing of the development projects (pipeline management). The development organisation's competences (development organisation) also constrain what is doable within a certain schedule. If the requirements cannot be turned into a product without using new, unknown technology, this affects the development schedule because of the learning curve involved. Because of this people may be stuck in projects that exceed their schedules and cannot be used elsewhere (pipeline management). Below is a fictional example of how easily things can get out of hand.

> Jack, the senior developer, who two weeks ago handled the installation for customer company Snoot Ltd., is working on a must-have requirement for an upcoming product release at the end of a development increment. As he is taking a short break to stretch his muscles after an intensive programming session, Jane's phone rings. Among other things, Jane's tasks include customer support, but unfortunately, she is at the grocery store downstairs to buy doughnuts for the company-wide Wednesday afternoon coffee break. Taking a brief look at Jane's ringing phone, Jack notices that the caller is Tom from Snoot Ltd., who was responsible for last week's installation on the customer's behalf. Naturally, Jack is curious about how the company has got started with using the delivered product, and answers the call on Jane's behalf. As Tom thinks he has reached the helpdesk, he tells Jack of some improvement suggestions to some of the features he has had in mind and reports two suspicious phenomena he considers bugs. Jack listens and scribbles down Tom's observations on a post-it note found on Jane's table. After the phone call ends, Jack returns to his computer and spends the rest of the day and a good half of Thursday enthusiastically programming those two of Tom's improvement suggestions that he considered relevant. He also tries to reproduce and fix the bugs Tom had told about. On Thursday afternoon, Jack succeeds in fixing the second bug mentioned by Tom, sends him an update, and resumes programming the 'must-have' feature for the upcoming release. On Friday afternoon in the weekly development team meeting, product manager Jeremy reviews what everyone has done during the week, and finds out about the call to the helpdesk Jack intercepted on Wednesday. Partly glad that Snoot Ltd. had an experience of an instant reaction to their needs but mostly frustrated that the "must-have" feature got delayed by modifications of questionable significance to the majority of the customers, Jeremy asks Jack to provide Jane the details about those improvement suggestions he had not yet realised to be put into the feature and idea database. Unfortunately, Jack does not remember them anymore, and while the post-it note with the specs is still somewhere, it is likely that nobody is able to decrypt the handwriting.
>
> A few weeks later Jeff, the CEO of the company gets a brilliant idea to improve a certain feature in the product while making a sales pitch at prospect Boot Ltd. Returning to the office in the afternoon, he immediately tells his idea to junior developer Joe at the coffee table, and asks whether he thinks the idea would be possible to realise. After the conversation, Joe stops testing the feature he was instructed to test in the morning by the product manager, and starts working on a prototype to find out whether the CEO's idea would work. Two days later, Joe succeeds in demonstrating the validity of the idea, and runs to show it to the CEO, who is in the middle of a meeting with Jeremy about the status of the upcoming

release. After refreshing the CEO's memory and receiving his commendation, the poor junior developer also gets a scolding from the product manager for his actions. Although the idea has now been turned into a feature, one of the important features remains untested. Furthermore, a more experienced developer could have demonstrated the feasibility of the idea in a couple of hours.

While the company in the anecdote displayed great flexibility, control was missing. The people might not have been aware of their roles and responsibilities (development organisation) and thought they were doing the company a favour with very fast reactions to customer needs. They also did not realise that they jeopardised the resource allocation of the development project (pipeline management) thus risking future product releases (commercial product management). The CEO and the management team had probably not created an explicit product strategy or roadmap for all to follow (commercial product management) and thus there was no baseline or vision to consider trade-offs against. What we are saying is that while flexibility is good, too much flexibility can lead to chaos and therefore a certain degree of control is needed. Control should not stifle the flexibility and creativity needed in a small software product company operating in a turbulent environment. Instead, it should set the necessary constraints to prevent total chaos.

As the key mechanism to combine flexibility and control we have identified rhythm. It works as the backbone of product development and helps coordinate the components of the SEMS to a functioning whole (Figure 1.2). In the next section we present our framework for rhythm, the Cycles of Control, which can be used when creating a SEMS for a company.

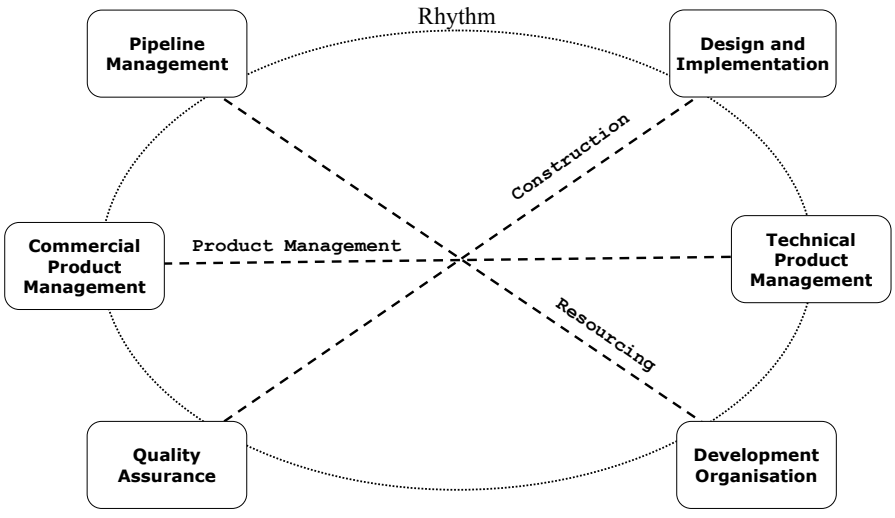


Figure 1.2: Rhythm as the coordinator of the SEMS components

1.3 The Cycles of Control: A Framework for Pacing Software Product Development

Overview and Structure of the Cycles of Control

The Cycles of Control framework, depicted in Figure 1.3, is based on the concept of *time pacing*. Time pacing refers to the idea of dividing a fixed time period allotted to the achievement of a goal into fixed-length segments. At the end of each segment, there is a milestone, at which progress is evaluated and possible adjustments to the plans are made. Changes can only be made at such a milestone. This accomplishes persistence and at the same time establishes the flexibility to change plans and adapt to changes in the environment at the specific time intervals. These time intervals, or *time horizons*, set the rhythm for product development. When the content of a time horizon is specified, a *time box* is created. In accordance with the time pacing idea, the schedule (end date) of a time box is fixed, whereas the scope (developed functionality) is not.

Figure 1.3 shows an overview of the basic building blocks of the Cycles of Control framework (CoC). Each cycle represents a specific time horizon and starts and ends with a control point in which decisions are made. The cycles and time horizons are hierarchical, meaning that the longer time horizons set the direction and constraints for the shorter ones.

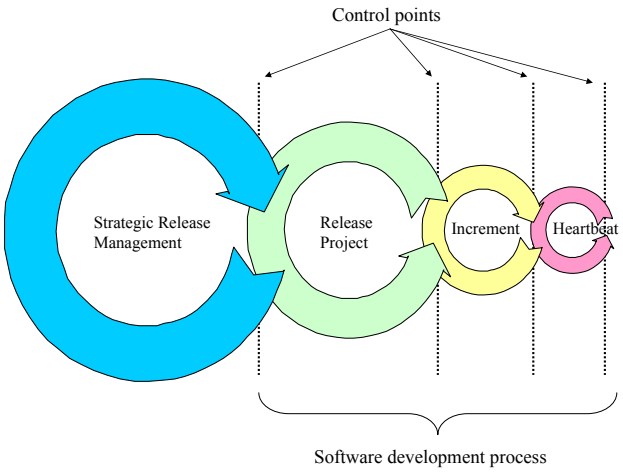


Figure 1.3: Cycles of Control building blocks

The leftmost cycle in Figure 1.3, strategic release management, deals with the long-term plans for the product and project portfolios of the company and provides an interface between business management and product development. Strategic release management decides what release projects are launched. Each individual product release is managed as a time-boxed project and dealt with in the release project cycle. Each project is split into time-boxed increments where partial functionality of the final product release is developed. Daily work is managed and synchronised in heartbeats. The three rightmost cycles constitute the software development process, which in our case is time-boxed, iterative and incremental. In order to better understand how the cycles relate to each other and show the hierarchy of the time horizons, the cycles can

be drawn on a timeline as depicted in Figure 1.4. In the example in Figure 1.4 we can see how the strategic release management time horizon spans two release projects, the releases are built in three increments, and the work is coordinated and synchronised with daily heartbeats.

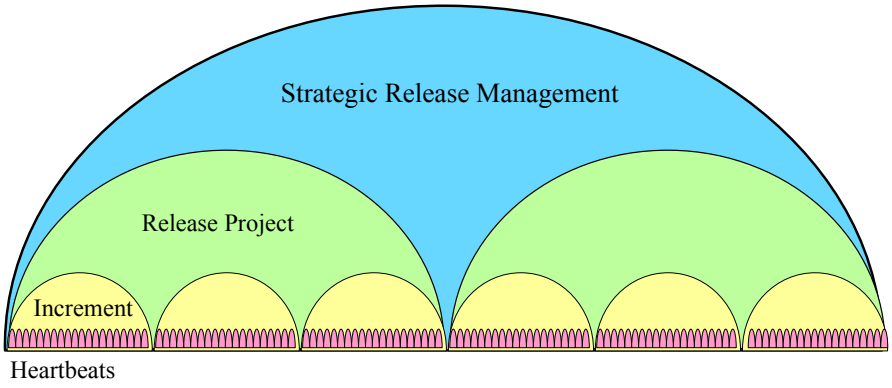


Figure 1.4: Cycles of Control on a timeline

Strategic Release Management

Strategic release management sets the direction for product development by aligning the product development efforts with the business and technology strategy of the company. This means considering the overall strategic ambitions of the company together with the competences and availability of people that do the actual work in conjunction with planning future releases of products. The starting control point is the planning of future releases and the closing control point is checking whether the goals have been met and whether the set direction is still valid or needs adjustment in the release plans. The roles that participate in strategic release management should include at least the most important stakeholders or stakeholder viewpoints of the products. These could be, e.g., the CEO, sales & marketing, customer services, product development, product management, and key customers.

The time horizon for strategic release management in a turbulent environment is 6-12 months, or 2-3 product releases ahead. The time horizon should be synchronised both with the marketplace and the internal capabilities of the company. Within the time horizon the upcoming product releases and the needed release projects are planned at a high level of abstraction (e.g., product vision, major new features and technology, quality goals, release schedule, coarse resource allocation) and documented, e.g., in the form of an aggregated release plan or a product and technology roadmap. This way there is a baseline against which to make trade-off decisions, e.g., when customers request something that has not been planned for the near future. Also, if a customer makes a request of something that is already in the roadmap, you can ask if the customer can wait until the planned release in 4 months, instead of immediately altering existing plans and disrupting work in progress.

Christensen and Raynor (2003) pointed out the crucial role of the resource allocation process in putting a company's strategic intention into action: "...a company's strategy is what comes out of the resource allocation process, not what goes into it."

This means that besides being time paced with, e.g., major roadmap revisions every 6-12 months, strategic release management should be represented in resource allocation decisions at least on the time horizon of an increment, since the outcome of increments is what the company actually does. In small companies the resource allocation decisions are especially important, since there is a very limited possibility to dedicate resources for longer periods of time. This is where rhythm can add stability and control. Of the key areas in Figure 1.1, strategic release management is involved with commercial product management, pipeline management, and development organisation.

Release Project

The release project cycle is concerned with the development of a release of a product. A steady release rhythm helps keep development focussed and provides opportunities to control development. The starting control point of a release cycle is release planning and the closing control point is the release decision, i.e., deciding whether the product version developed can be released or not. Depending on whether the release is internal or external, the quality goals and release criteria can differ. The roles that participate in a release project are the project team and strategic release management representation for the control points, as discussed above.

The time horizon for a release project is 3-4 months, during which a release candidate of the product is developed. In the beginning of the release cycle the release is planned and specified based on the goals and priorities set in the roadmap or long-term release plan. This includes deciding on the schedule and content of the increments in the project and appointing the project team. Also, you need to plan when and how you are going to test the product, so that you can make sure that the quality of the released version is sufficient to make releasing the product possible. This includes allocating necessary rework time in the end of the project cycle to fix the defects that need to be fixed. Since the project is time boxed it means that the schedule cannot slip. The requirements need to be prioritised so you can make decisions on altering the scope of the project, if everything cannot be finished within schedule. Based on the release goal(s) set by strategic release management, decisions about decreasing the scope can be made, as long as the release goal is not compromised.

Increment

The purpose of increments is to develop the product as a series of reasonably stable, working intermediate versions having part of the functionality of the final release. This is done to get feedback of the product during development. The starting control point of an increment cycle is the planning of the work to be done. The closing control point of an increment cycle is the demonstration of the developed functionality. The roles that participate in an increment cycle are the development team and strategic release management representation in the control points, as discussed above.

In a turbulent environment the time horizon of an increment cycle should not exceed 1 month, during which a stable increment is developed and integrated into the product. During an increment cycle the requirements and resources should be frozen. Therefore, if it is possible to split work into shorter increments without excessive overhead, it is advisable to do so to guarantee the availability of the allocated resources. The shorter the increment cycle the fewer the possible interruptions are. The developers should be allowed to concentrate on the work planned for the increment. The

key lies in including all known commitments that need attention from the developers into the resource allocation plan in the beginning of the increment. For example, if Jack needs to help in integrating the system at a customer's site, the time needed for this should be subtracted from Jack's product development time, and so on. Even for a month-long increment cycle, there should not be any big surprises, except for bugs found by the customers, that need immediate attention from the developers, and for this you might consider dedicating one person. By the end of the increment cycle, the new features developed are integrated into the product and the product is stabilised, meaning that testing and bug fixing needs to be planned and executed. This should be considered when planning the increment. Requirements need to be prioritised so that scope adjustments can be made. An important part of planning is converting the requirements into tasks for effort estimation. Subsequently, the selection of requirements to be done during the increment cycle can be based on the available product development time and the estimated effort for the tasks. The increment cycle ends with a demonstration of the product for all interested stakeholders, where comments and feedback are gathered and can be used when planning the following increment(s).

Heartbeat

At the lowest level, development is controlled by using *heartbeats*. A heartbeat is the shortest time horizon within which the progress of development is synchronised and reflected against plans. The starting and closing control points of a heartbeat can be combined into a status check that creates links in time from past to present to future in the form of three questions: What have you done? What problems are you facing? What will you be doing next? The participants in heartbeats are the development team members.

The time horizon of a heartbeat can range from a day to a week. During this time development proceeds and at the end of the heartbeat the status is checked. This way there is up-to-date information on project progress at regular, short intervals. This helps in identifying early warning signs of things that might compromise development goals. For example, if Joe has not been able to use his time as planned to developing the product, this is revealed in time for corrective actions to be taken, instead of being revealed at the end of the increment cycle when it is too late to react. Another example is that Joe cannot finish his task within the estimated effort, which could affect the work of others. Therefore, a part of the heartbeat cycle is updating the estimated effort left for tasks. A part of synchronising the work might be making daily builds and running automated smoke tests against them. This gives an indication of system status from a technical perspective.

The Benefits of Rhythm

On the strategic release management time horizon, rhythm helps to protect a company from the disrupting effects of external forces, while maintaining responsiveness. It supports trial-and-error type learning, which is important for start-up companies that cannot base plans on prior experience. A trial period to pursue a strategy must be long enough to protect against abandoning it prematurely as unsuccessful, but short enough to protect from chasing a lost cause for too long. Time pacing allows the company to pursue each strategy fully until its viability is evaluated. Simultaneous awareness of the shorter time horizons helps in harnessing the projects and increments to serve the long-term goals. This helps to maintain focus, instead of dribbling with too many

alternatives. Feedback on short-term progress gives an early indication of how doable the set goals are.

Time boxing the increments and projects sets a rhythm for packaging the product that helps keep surprises regarding, e.g., product quality to a minimum. Jim Highsmith (2000) reports what he sees as the benefits of time boxing:

1. Time boxes force hard business and technology trade-off decisions from all parties of a project (managers, customers and developers). Without the pressure to make these decisions, human nature delays them towards the end of the project, when there is very little manoeuvrability.

2. Short time-boxed deliveries force convergence and learning. People tend to want to do quality work, so when the deadline of the time box is absolute it helps overcome the tendency to gold plate a deliverable, before it is shown to others. Instead, when you are forced to deliver your work at the end of the time box, you can learn from any problems encountered during the time box and hopefully use this to your advantage in the following time boxes.

3. Time boxing helps keep the team focussed. A specific due date provides the focal point. When you have to deliver and present the product to outsiders of the development team (customers or other stakeholders), it forces the team to concentrate on the most important activities. At the end of the time box, when planning the next time box, any new or existing uncertainties can be addressed again.

All the points above can easily be misinterpreted and badly implemented. The first point can be turned into doing anything for the sake of keeping the project moving, not because it is the most important thing to do next. Also, if decisions are made ”mechanically” at each control point, the need to change the direction and scope of the project or even kill the project due to changed market conditions may go unnoticed. The second and third point can lead to stress and disappointment in your employees. As Highsmith says, people like to take pride in their work and thus it may feel awful to reduce the scope of the time box or being forced to turn in deliveries that are not ”perfect”. Both could be seen as reflecting poor performance, even when it is not true. Therefore an important issue in creating a development rhythm is to do it in the right mindset of learning and improving.

One benefit of a tight rhythm is that you have a mechanism for frequent resource allocation. In small companies people have multiple roles and responsibilities and managing what each person does next is often neglected or done ad hoc, not because it does not seem important, but because it seems hard or even impossible. However, with short increments you can freeze at least most of the people’s tasks for the duration of the increment.

Another interesting benefit of development rhythm is suggested by Larman (2004) and has to do with human nature: “*People remember slipped dates, not slipped features.*” It means that if you deliver the most important features (e.g. 60 % of all features) on time, your project is seen as a success. If on the other hand you deliver all the features a few months late, your project is seen as a failure. Creating a rhythm for product releases and holding on to it can make you more successful in the eyes of your customers.

Applying the Cycles of Control

In this section, we explain the general idea of how the CoC framework can be used when you create your own SEMS. Figure 1.4 above shows the different cycles and their time horizons on a timeline and gives you an overview of how the product development process works. The time horizons set the rhythm of product development, and can be used as a starting point for planning and mapping different practices and activities to that rhythm. For a practice or activity belonging to a certain time horizon means that it is planned and tracked with that pace. If necessary, a practice or activity can be split into parts to be tracked on a faster pace, and so on. To give you an example, let us consider how requirements can be managed (part of commercial product management in Figure 1.1) using *backlogs*, an idea presented in the Scrum process model (Schwaber and Beedle 2002).

> A bunch of product stakeholders and stakeholder representatives participate in the April roadmapping session to plan future releases of the products. Jeff (the CEO) represents the company strategy and wants to secure that the products reflect this strategy. John (the Visionary) provides some "out there" visions about the future development of the markets and technology, supported on the technology front by Jenny (the Chief Architect), who also is responsible for more "down to earth" assessments on the viability of using new technologies. Jermaine and Joanna (the Sales Directors) represent the customer viewpoint and bring the latest information from the markets. Jeremy and Jay (the Product Managers) are responsible for one product each and also represent the viewpoint of customer support (Help Desk and Product Delivery). Jericho (the Marketing Director) wants to secure sexy features to future product releases, so he can market them successfully.
>
> The meeting starts with Jeremy presenting the up-to-date product backlog of Widget, the older of the two products of the company. A product backlog is a prioritised list of product requirements and features of varying scope. All the ideas for the product have been gathered into the product backlog and Jeremy is responsible for keeping it up-to-date. Jeremy presents his preliminary suggestion for the release backlogs of the two following releases of Widget. A release backlog contains the requirements and features to be included in a product release and is a prioritised list, like the product backlog, only a bit more detailed. At the end of August a minor release of widget is scheduled, containing some bug fixes and a few new features. The next major release for Widget is scheduled just before Christmas and contains support for new databases that are needed to penetrate new markets and a bunch of other new features.
>
> Jeff is pleased with the release backlogs, especially since the strategic intent of the company is to move to new markets to generate new cash flow. Jenny expresses concern for the tight schedule, because Jack (the Senior Developer) has been very busy with rewriting Gadget (the second product of the company) for .NET, and the progress has not been as good as expected, as everybody could see in the last increment demo of Gadget. Since Jack is the database expert of the company, a decision must be made on which is more important, getting .NET Gadget out in time or extending database support for Widget. Jermaine (Sales Director of Widget) and Joanna (Sales Director of Gadget) argue that both are very important for their customers and Jericho shows that market research results support an aggressive strategy to move into the new markets now. Before the meeting breaks into on open fight, Jay (the Product Manager of Gadget) proposes that he shows the release plans for Gadget, so that the possible trade offs are clear to everybody. When Jay has shown his suggestions for release backlogs for Gadget, the lively debate continues until lunch. Everything seems important, and no trade offs can be made.

After lunch, when things have cooled down a little bit, Jenny suggests that Jack continues working on dot-NET-ting Gadget. But, instead of doing it alone, he pair programs with Jo (a junior developer). Pair programming has been used earlier with some positive results in different tasks, so Jack and Jo already have some experience in doing it. This way Jo would learn from Jack and in a couple of increments Jo would be able to continue on her own, if necessary, leaving Jack free to start working on the new database support for Widget. The drawback is that some of the features in the release backlog need to be reprioritised to lower priority, since Jo cannot be working on them, meaning that they probably cannot be finished in time for the release. But at least the most important goals for the releases of both products have a greater chance of being met. The meeting participants discuss Jenny's suggestion for a while and agree that this is the best course of action. The meeting then continues with more discussion and reprioritisation of the release backlogs.

A week later Joanna, Jay and Jenny meet with Jill (the Development Team Leader), Jack, Jo, Joe (another Junior Developer), and Jake (the Quality Engineer) to plan the next increment of Gadget. At the coffee table they have already discussed some of the ideas from the roadmapping session, so there are no big surprises for anyone. Joanna, Jay, Jenny and Jill have prepared for the planning session by discussing the most important release backlog items and what they mean in more detail, both from a business perspective and from a technical perspective. The results from these discussions are presented to the development team and questions about unclear things are asked and answered. Then the team is left alone to plan how these release backlog items can be broken down into tasks for the increment and effort is estimated for the tasks. When the planning is ready the development team presents the results to the others and also discuss the budget of available development effort for the increment. It is apparent that not all tasks can be done within the available budget, so Joanna, Jay, Jenny and Jill discuss and prioritise the scope of the increment. They also create the increment backlog, which contains the increment goal(s) and the features to be done including the planned breakdown to tasks for developing those features. When the increment backlog is ready the development team joins the others and the backlog is shown and discussed. After that the team accepts responsibility for realising the increment backlog, at least to the extent that the increment goal is met.

A month later the same increment planning process is repeated using what was left in the release backlog as a starting point. At this time new, emerged requirements can be traded off with those in the release backlog to reflect changed prioritisations. These should not, however, be in contradiction with the release goals. Changing the release goals every month would probably result in over reacting to changes in the market. Of course, if the initial analysis of the markets was totally wrong, even the release goals should be changed.

The example above includes discussion on many issues from Figure 1.1 and shows the nature of software product development management. To recap the main issue in the example, requirements can be managed using backlogs of different scopes as depicted in Figure 1.5. As we move downward in Figure 1.5 the backlogs get more detailed. Managing the requirements on a monthly rhythm gives us flexibility to change plans if we have missed something earlier. Each month we also see how much has been accomplished, giving us a measure of progress we can compare to the plans and goals. This gives us control to make corrective actions based on real progress, if our plans have been too optimistic or pessimistic.

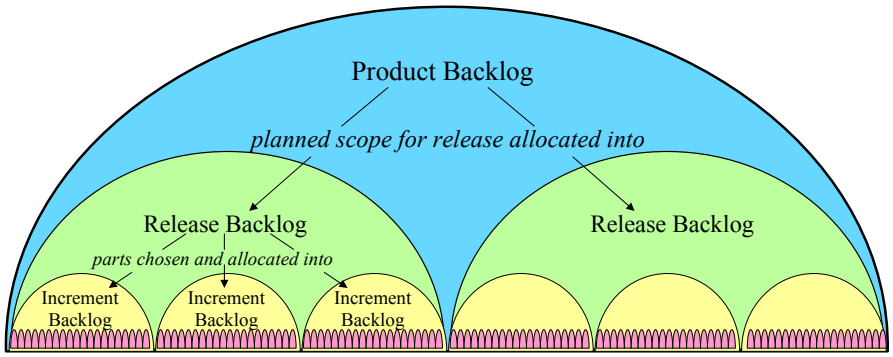


Figure 1.5: Managing requirements with backlogs

Finding Your Own Rhythm

Finding a suitable rhythm entails understanding the rhythm of the markets and the internal capabilities of the company. Releasing products to the markets should be done at an appropriate rhythm. For example, if a magazine publishes a product review at a certain time of the year, you need to release your product in time for that review. Or if a trade show is organised at a certain time, you need to have a product release ready by that time. Another example could be seasonality, e.g., you need to release a product for the Christmas market. Your product's maintenance agreement may also contain promises of maintenance releases with a defined rhythm. All releases of the product do not need to be external or commercial releases. You can also make internal releases that can be used in demos for potential customers or just used as intermediate versions for, e.g., thorough testing. This way you can get a better understanding of the product and improve it before you make it widely available.

While the rhythm of the markets tells you when you would want to release your product(s), the internal capabilities of the company constrain what is possible. With the internal capabilities we mean, e.g., how effective your processes are, how skilled your employees are, how easy it is to develop and test your product(s) incrementally, and how much development effort different people can contribute considering all the other tasks at hand. One way to find out the internal capabilities is to define and try some rhythm and see how it goes. For example, if you decide to make a commercial release of a product once a year, you could make two additional internal releases per year, which defines the release rhythm as 4 months. Then you could define 1-month increments for developing the product and use a daily heartbeat rhythm to monitor progress. If the tasks planned on a heartbeat time horizon start taking almost an increment to complete (instead of 1 day to 1 week) you have not been able to plan and split the tasks into small enough items. This could mean that you have selected too difficult and large features to be done in the increment or that the increment as well as the heartbeat is too short. If you think the increment and heartbeat rhythm is appropriate, you need to improve your skills in planning the increment and the tasks to be done in it. One of the problems may also be that you do not yet understand what you are supposed to get done or how it can be done using some new technology. In that case you might need to reconsider the release goals and increment contents to reflect that you are learning a new technology. The rhythm helps you show progress or lack thereof, which in turn helps you make informed decisions about continuing or

discontinuing pursuing the goals or turning to an alternative course of action in order to be able to make the commercial release.

A more structured way of planning and defining the development rhythm based on the internal capabilities is considering what needs to be accomplished by the end of each time horizon and how long that will take. For example, when a product is released to the market, there is much more involved than just coding and testing the product. You may need product documentation, such as installation instructions and a user's manual. You may need marketing material containing, e.g., screen shots of the product, well in advance before the product release. You may need sales material for the sales people, such as brochures and demonstrations of the product. The preparation of all these have lead times that need to be considered when planning the increments of the release. A good idea is to dedicate at least the last increment before a commercial product release to stabilising the product and preparing all the necessary accessories. For the screen shots for the marketing material you may need to make a visual freeze even earlier than in the last increment.

Stabilising the product means that we do not develop new features but rather make sure that the existing ones work properly. For this we need to do some testing and bug fixing, the amount of which depends on, e.g., how much and what kind of testing we have been able to do in the previous increments. If we need to do extensive testing that could take 2-4 weeks to complete, a 1-month increment is not long enough.

Adopting a Practice

To give you an example on using rhythm and the different time horizons in planning how to adopt a software engineering practice, we use refactoring. Refactoring means changing the internal structure of the code without changing the external behaviour of the software. Refactoring has been used successfully as a practice in eXtreme Programming (XP) and you might be interested in trying it out. You could directly try the XP way as described in (Beck 2000) or you might want to consider other approaches. Figure 1.6 shows three different approaches to refactoring: 1) refactoring heartbeat, 2) refactoring increment, and 3) refactoring release.

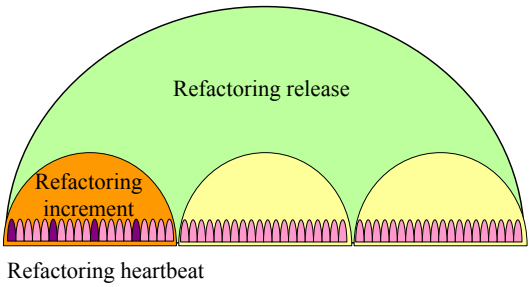


Figure 1.6: Different approaches to refactoring

1. Refactoring on a heartbeat time horizon could mean dedicating one day of the week to refactoring the code, as shown in Figure 1.6. It could also mean that refactoring is a part of every work day, as explained in XP. Each developer is responsible for refactoring code when it seems appropriate.

2. Refactoring on an increment time horizon could mean that the first increment of a release project is dedicated to refactoring the code, which is shown in Figure 1.6. Another option would be dedicating the beginning of each increment or some increments to refactoring the code.
3. Refactoring on a release project time horizon means dedicating a whole release to refactoring the code. This option is probably the least likely to be used. One should try to avoid getting the code in such a bad shape that this is needed.

The approaches above are by no means mutually exclusive. Rather you could combine them, e.g., by doing major refactoring in the first increment of a release project and then in the rest of the increments you do refactoring on a case-to-case basis, i.e., decide in heartbeat control points if and when refactoring is needed.

Putting it All Together

The previous sections discussed the basics of rhythm-based thinking when you start defining your own way of managing software product development. The first thing you need to consider is whether pacing is the right thing for you or not, but since you are reading this, you are probably ready to try it. The next step is figuring out the right rhythm for you on different time horizons, as explained above. Perhaps the most important part of this is defining the length of increments of different activities so that you can synchronise the increment rhythm(s), since this is used as the basis for resource allocation (pipeline management). For example, if the increment time horizon of a major product release project is 1 month (or 4 weeks), other increment time horizons (e.g., for maintenance releases) should be 4 or 2 weeks, or 1 week. This way they are all synchronised every 4 weeks, which would be the rhythm for making major resource allocation decisions.

A practical approach to continue with is to consider how you do things now and make small adjustments to accommodate the chosen rhythm, e.g., thinking like in the refactoring example above. Figure 1.2 shows the key areas of software product development activities, which are further elaborated in the rest of this book. The key area descriptions can be used as a reference in figuring out how you do things now and as guidelines for improvement. Then you need to define the roles and responsibilities and decisions to be made at the control points in Figure 1.3, after which you have defined your first version of a paced SEMS, as shown in Figure 1.7.

The resulting paced SEMS in the bottom half of Figure 1.7 is simplified for readability. In real life one picture is not enough to communicate all aspects and details of software product development, not even for simple overview purposes. The strategic release management time horizon and definition could be the same for all products of a company and their different release types, but for the other time horizons, additional definitions and pictures are needed — possibly for each product. The following list provides an example of a minimum set of views:

1. An overview of the release project for a major release, which could include themes for increments (examples can be found in Section 4.5) and how testing is organised (examples can be found in Section 5.5).
2. An overview of the release project for a maintenance release, because it is much simpler than a major release and this should be reflected in the picture and its definitions. Otherwise the view should include the same things as above.
3. An overview of the portfolio of products and their release project cycles including increment cycles. This view could additionally include other activities

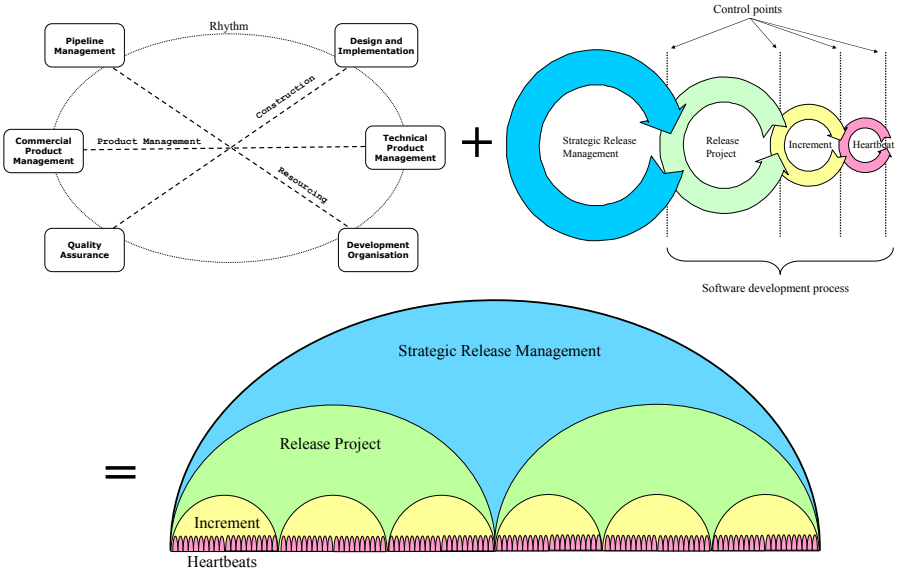


Figure 1.7: Putting it all together

demanding attention from product development, such as customer installations. This view is especially important for pipeline management (examples can be found in Chapter 3), as it shows where resources are needed.

The sub sections in Section 1.3 provided an overview of the different cycles, their control points, and participating roles. Some of these are further elaborated in the other chapters of the book. For descriptions on early implementations of the SEMS approach, please refer to (Rautiainen, Vuornos, and Lassenius 2003) and (Vanhanen, Itkonen, and Sulonen 2003). Next, we look at the organisation of the rest of the book.

1.4 Organisation of the Book

In this section we provide an overview of how this book is organised and guidelines on how you should read it.

Book Structure and Contents

Chapters 2-6 each correspond to one of the key areas of the Software Engineering Management System (Figure 1.1). There is not a separate chapter for the key area of Development Organisation. Rather, the organisational issues are discussed in appropriate places within the other chapters. Because covering the key areas completely would take up a bookshelf of books instead of one, we have tried to focus on issues that have so far received little attention in software engineering literature and research.

The following is a summary of the contents of each chapter.

Commercial Product Management (Chapter 2) consists of *product and release planning* and *requirements engineering*, and this chapter focuses on the former. Chapter 2 presents a terminology for software product and release planning, outlines the value of long-term planning, presents a framework through which prod-

uct and release planning and its relationship to requirements engineering can be understood, and finally, gives an example of a process used for product and release planning.

Pipeline Management (Chapter 3) means looking at the portfolio of development activities as a whole, and making appropriate decisions on resource usage in a timely manner. Software engineering has traditionally been primarily technical and tends to adhere to the viewpoint of individual development projects as far as management is concerned. However, an effective process for defining, evaluating, and prioritising the set of current and planned product development activities is crucial to product-oriented software companies' long-term success. Chapter 3 presents the principles of successful pipeline management, provides guidelines on managing the different types of activities that your developers have to do, and shows how the pipeline management process can be used to synchronise these. Chapter 3 closes with discussing how the strategic release management cycle links Commercial Product Management and Pipeline Management in practice.

Software Design and Implementation (Chapter 4) discusses how to organise design and implementation of software products in time-paced software development. Design refers to the design or architecture of the software and how to plan the implementation of new functionality. Implementation refers to actual programming. Chapter 4 discusses the importance of design and presents design principles and practices, and how to organise and manage implementation. Pair programming and refactoring are discussed more in-depth as examples of potentially beneficial modern design and implementation practices. Next, software evolution and so-called technical debt are discussed. Chapter 4 closes with showing how time pacing influences design and implementation and describes how the Cycles of Control framework can be utilised in organising design and implementation.

Quality Assurance (Chapter 5) refers to all dynamic (e.g., testing) and static (e.g., reviews) activities and practices that are applied to improving the quality of the software product as part of the development process. Chapter 5 focuses on how to organise quality assurance in an iterative and incremental software development process. First, the fundamental concepts of software quality in the context of iterative development are discussed. Second, key concepts that are useful when planning an overall quality assurance approach are defined, such as good-enough quality, test mission, and test strategy. Third, Chapter 5 describes how the Cycles of Control framework can help you see quality assurance as an integral part of the iterative and incremental development process instead of just being a "separate phase at the end".

Technical Product Management (Chapter 6) refers to a systematic approach to dealing with items such as requirements, design models, source code, test case specifications, and development tools. Chapter 6 discusses selected key concepts of technical product management (version control, change control, build management, and release management) in the context of iterative and incremental software development.

In addition to the five chapters dealing with different key areas of the Software Engineering Management System, the book contains two appendixes that cover two areas that we believe are in demand in most small software product companies. The first appendix (Tool Support) discusses tool support in the context of small companies employing iterative and incremental development processes. The second appendix (Software Process Improvement Basics) summarises a number of software process improvement success factors from literature and our own experience.

How to Use this Book

Below is a possible approach on how you should go about reading this book:

1. Skim through the book, look at the titles and read the points highlighted with boxes.
2. Read Chapter 1 to understand the core idea and concepts, and to get an overview of the rest of the book.
3. Read any of the chapters 2-6 to learn more on the key areas in managing software product development and their connection to development rhythm.

Depending on your role, you may find some chapters relatively more interesting than others. However, a key point on our agenda is to provide a common point of reference (that is, rhythm) to help Business and Development People understand each other's work.

Thus, if you find some of the chapters interesting and useful, we strongly recommend that you also take a look at those chapters that seem the least relevant for you, and try to figure whether that is really the case.

References

Beck, K. 2000. *eXtreme Programming eXplained*. Boston: Addison-Wesley.

Christensen, C. M. and M. E. Raynor. 2003. *The Innovator's Solution: Creating and Sustaining Successful Growth*. Boston: Harvard Business School Press.

Highsmith, J. A. 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House Publishing.

Larman, C. 2004. *Agile & Iterative Development: A Manager's Guide*. Boston: Addison-Wesley.

Rautiainen, K., L. Vuornos, and C. Lassenius. 2003. An Experience in Combining Flexibility and Control in a Small Company's Software Product Development Process. In *Proceedings of ISESE 2003*: 28-37.

Schwaber, K. and M. Beedle. 2002. *Agile Software Development with Scrum*. Upper Saddle River: Prentice Hall.

Vanhanen, J., J. Itkonen, and P. Sulonen. 2003. Improving the Interface Between Business and Product Development Using Agile Practices and the Cycles of Control Framework. In *Proceedings of Agile Development Conference 2003*: 71-80. Also available at: `http://agiledevelopmentconference.com/2003/files/R3Paper.pdf`

Chapter 2

Commercial Product Management

Jarno Vähäniitty

2.1 Introduction

This chapter discusses commercial product management in the context of small software product businesses. Commercial product management consists of *product and release planning* and *requirements engineering*. While requirements and backlogs are touched at many points in this chapter, the focus here is on the business aspects of commercial product management, that is, product and release planning. Requirements and backlogs are discussed throughout the book.

In this chapter we first explain what product strategy means and how it relates to issues like business and corporate strategy. Then we discuss why it is good to plan for the long-term even in situations where the benefits may not seem obvious. We provide a framework for product and release planning and describe a practical approach to formulating and maintaining product strategies and translating them into tangible release plans.

2.2 Commercial Product Management Terminology

Small companies find integrating a strategic perspective into product development decision-making more challenging than larger firms with more established strategy processes. Important product development decisions are often made based on the opinions of key personnel rather than through deliberate or explicit planning, and consequently, small companies are vulnerable to extensive rework and even market failure (Brouthers, Andriessen, and Nicolaes 1998; Smith 1998). While small companies may not need or even afford having dedicated strategy personnel, maintaining the 'big picture' is challenging for the key persons in the everyday bustle of multiple and sometimes even contradictory roles and responsibilities.

People often assume that managers know their strategies and business models cold. However, existing research (Linder and Cantrell 2002) as well as our experiences suggest that this is not the case. Product strategies and business models are easily left fuzzy in practice, because of their abstract nature.

Systematic and consistent discussion of long-term product and release planning is difficult without the solid foundation of a clear conceptual framework. Thus, we start

off by defining a set of terms that we have found useful for understanding the business end of managing software product development in small software product businesses.

> A clear terminology for commercial product management helps maintain the 'big picture' in the everyday bustle of multiple and sometimes even contradictory roles and responsibilities.

While reading, consider how your reality maps to these concepts. The mapping likely reflects how well the issues are understood at your company. You might also want to ask a couple of colleagues to explain how they view the respective issues without briefing them first on how you see things (. . . but do not blame us if they give quite different answers).

> Consider what the terms below mean for your company, ask your colleagues to do the same, and reflect on the results.

We examine some of the concepts below more closely, while others are just mentioned and provided with a reference for further reading. The reason for this is that only some of the concepts (such as *product strategy*) are central from the perspective of this book. Because many of the definitions below build on each other, terms that are mentioned before their definition are marked in *italic*.

Corporate strategy refers to a company's overall direction in terms of its various businesses, resource usage, general attitude towards growth, and management of business units (Hitt, Ireland, and Hoskisson 1997; Rajala, Rossi, and Tuunainen 2003; Wheelen and Hunger 2002). Common terms that deal with the same general area as corporate strategy are, e.g., business policy, management policy, and organisational strategy (Johnson and Scholes 1993).

Business strategy focuses on how to compete in a particular industry or product/market segment. The business strategy is commonly summarised in a vision and mission statement. (Rajala, Rossi, and Tuunainen 2003; Wheelen and Hunger 2002) In small companies, corporate and business strategies are often not distinguished from each other, which may or may not be a good thing, depending on whether the company is actually involved in multiple businesses or not.

Business environment refers to the environment a business (as opposed to a corporation with multiple businesses) is operating in, in terms of competition, financing and resourcing, the market, and customers (Rajala, Rossi, and Tuunainen 2003).

Business model refers to the manifestation of a business strategy designed for a particular product and a particular business environment.Its four elements are *revenue logic*, *marketing & sales*, *delivery*, and *product strategy* (Rajala, Rossi, and Tuunainen 2003; Vähäniitty 2003). The relationship between a business strategy and a business model is similar to that of a product vision and the software architecture that is used in realising it.

Marketing is one of the four elements of a business model, and it refers to market segmentation, selection of target market(s) and presenting the products to the customers in a way that enhances their perceived value. This also encompasses decisions on how new features can be marketed while they are still under development.

Sales is one of the four elements of a business model, and it refers to decisions on how the products are intended to reach their markets, in other words, how the products

are *sold* and *distributed* to the customers. Besides the initial reference customers and direct sales, small companies must often employ indirect sales channels to reach their markets.

Revenue logic is one of the four elements of a business model, and it refers to how the company extracts value from the product — its mechanisms for creating sales revenue, the basic idea behind pricing, and utilising other possible sources of financing (Afuah and Tucci 2002; Rajala et al. 2001). Some examples of decisions regarding revenue logic are: does the company sell product licenses and provide upgrades and maintenance for free, do the customers get the software product for a negligible price and pay the company for associated services, and what share of profits do possible indirect sales channels get.

Product strategy is one of the four elements of a business model, and it defines a *product vision* and a *release strategy* for a specific product/service proposition. Product strategy is formulated in *product and release planning* and enacted through *pipeline management*. From the perspective of this book, product strategy is the most important element of the business model.

(Long-term) product and release planning is the process for creating and updating product strategies. In short, this means deciding what enhancements future releases of the products should contain, when they should be made generally available, and what complementary services should be offered to ensure future competitiveness (Penny 2002; Vähäniitty, Lassenius, and Rautiainen 2002). Product and release planning is further discussed in Section 2.4. In practice product and release planning is often referred to simply as *roadmapping*, which is a popular metaphor for planning and portraying artifacts, resource usage, and their relationships over a period of time.

Pipeline management is the process of looking at the portfolio of development activities as a whole and making appropriate decisions concerning resources in a timely and organised manner. Effective pipeline management is crucial from the perspective of enacting strategy, and we have dedicated a separate chapter to how it should be conducted in time-paced development. Together, product and release planning and pipeline management constitute the strategic release management process.

Product vision is an artifact that describes a high-level view of the product. It describes what the product is and where it should be going with respect to what it does, its business case, market, competitors, business model(s), and extending and further developing its technological basis. Please look up (Cusumano and Selby 1995), (Dver 2003) and (Kroll and Kruchten 2003) for details on product vision documents.

Release strategy consists of the decisions for the *release cycles*, *goals*, and *contents* of future product *releases*, considering future *resource needs* and possible changes to the underlying technologies. Release strategies should be documented in product roadmaps to summarise and communicate the key decisions regarding release cycles, goals, contents and perceived resource needs.

Release refers to passing on a software build and associated documentation to one or more parties outside of development (Penny 2002).

Release goals refer to the releases' intended implications for the company from a business perspective (Rautiainen, Lassenius, and Sulonen 2002).

Release contents refers to deciding which features should be included in which future release and justifying these decisions from a business perspective (for example by linking features to business needs and market opportunities) (Wiegers 1999).

Resource needs refer to considering whether proceeding as planned will require more (or less) resources or completely new kinds of competences.

Release cycles refer to an understanding of what kinds of releases of the product are made, and when. Release cycle planning means making a classification of different kinds of releases according to some schema that characterises the *targeted audience*, *timing*, and *extent* of the release. Timing means setting the release dates of subsequent product releases of different types to correspond to the *market rhythm*. Target audience means that releases can be internal, for example from the developers to the testers, or public, for example to a specific end user, a trade show, or to all customers. Extent basically denotes how much changes were made compared to the previous release based on the amount and type of product development effort spent. Figure 2.1 illustrates the results of release cycle planning.

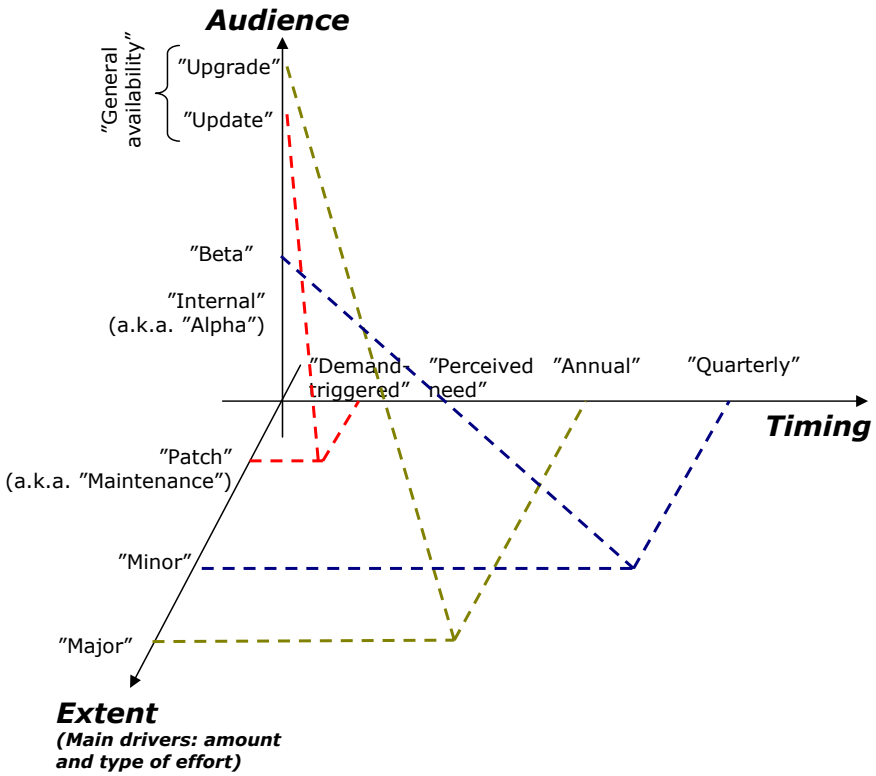


Figure 2.1: An example output of release cycle planning

In Figure 2.1, major releases are made once a year, and maintenance releases are made when a customer specifically requests for one. Also, beta versions of the product are released each quarter for certain customers to get early feedback on new features. Major releases require a separate upgrade fee, while minor and maintenance releases are free for existing customers.

Market rhythm refers to the perceived dynamics of a market in terms of, e.g., major events. Market rhythm sets demands for the product development process of the company through release cycle planning. In other words, the rhythm of the product development process should reflect the rhythm of the business, and this means that you should try to pace your product development according to the rhythm of your market.

Pace your product development according to the rhythm of your market.

Thus, the goal of process improvement should be to ensure a fit between the demands set forth by the rhythm of the market and the capability of the development process. For example, the need to release a product having a near-zero tolerance for defects sets demands on quality assurance. This in turn places constraints on the release cycle length.

In this light, issues such as process maturity and capability that have traditionally been paid lots of attention to in software engineering literature, are important only to the degree they make it possible for your product development to operate according to the rhythm your business demands. The constraints and requirements set by the market rhythm and development capability on each other should be reconciliated in the release strategy.

The goal of process improvement is to ensure that the development process can cope with market rhythm.

In other words, process maturity and capability are important only to the degree they support your release cycles.

Table 2.1 lists drivers that can affect release cycle length, which you should consider. In addition to market rhythm, the business model, in other words, the revenue logic, marketing & sales, and delivery models are also likely to impose constraints and requirements for the release cycles.

A final word of assurance for those of you who at this point feel anxious about understanding the rhythm of your market; even an attempt to identify possible factors, judging them, and then setting the development rhythm according to the resulting "gut feeling" is definitely better than doing nothing. In the companies that we have worked with, release cycle length ranged between three months to one year, and increments ranged between two to four weeks. We suspect that it is likely that similar numbers will fit your company as well.

If you feel that it is unclear how the business you are in affects what kind of product development processes you have, you can find consolation in the fact that the key issue is to build rhythm into the product development process, improve the process so that the rhythm can be maintained, and then re-tune the rhythm. Regardless of your business, it is better to have some rhythm than no rhythm at all.

Identifying the market rhythm is a subjective exercise, but that does not make it any less important. Think about what factors are the most essential in your business and discuss their effects.

A quarterly release and four week increment rhythm is a good rule of thumb to start with.

Regardless of your business, it is better to have some rhythm than no rhythm at all.

Table 2.1: Example factors affecting release cycle length

Factor	Explanation
Market events	Match your release cycles with those of your competitors, and check that your releases' timing is in proper synch with the most important events for your industry (trade shows, etc.)
Rate at which customers can absorb new software	Users may lack a convincing need or the technical support to upgrade their systems regularly (Dver 2003).
Intensity of competition	Intensity of competition can emphasise the effects of competitor-related factors, such as competitor's release cycles and product life cycle.
Product life cycle	You should have short development cycles in the start-up phase to get early feedback, establish a market position and keep the technology moving forward. Don't wait to create the "perfect" product. As the product matures, you should shift the emphasis gradually to quality instead of features and time-to-market. (Cusumano and Yoffie 1998) Also, innovators and early adopters are more tolerant to problems in the product, such as bugs, difficult installation, and so on because they accept that these will be fixed in upcoming product releases (Hohmann 2003).
Number of expected installations	A mass-market product will have a higher demand for regular releases that include a handful of new features. Customers expect that bug fixes and other maintenance patches will be released, as they are available.
Product complexity	For complex products, upgrading may be more complicated as well; for example, delivering an update may require servicing effort from your company (Dver 2003).
Product criticality	Product criticality in most cases means that the periods for system testing have certain 'minimal lengths' that must be adhered to. This means that the release cycle can never go below the time to develop a reasonable set of new functionality + the "quarantine time" from system testing.

2.3 The Value of Long-Term Planning

Effective planning is crucial for all companies. For a start-up it often makes the difference between survival and ruin. This is especially true for companies in the software product business because of shortened cycle times. Ironically, these may also be just the companies that are most tempted to bypass the planning stages in an attempt to shortcut the development process and bring products to market more quickly (Mello 2002).

Long-term planning is traditionally perceived to consist of steps such as analysing the industry, selecting a strategy, and building tactics around it (Mintzberg, Ahlstrand, and Lampel 1998). Often, managers question whether their company should "indulge in elaborate strategic management techniques", and omit long-term planning on the grounds that it is based on theoretical ideals having little to do with management realities.

Indeed, a major problem with long-term planning is that its outcome, i.e., plans, are always to some extent wrong (Brown and Eisenhardt 2002). This is because the analysis does not reflect all relevant factors, and anyhow, plans become obsolete quickly in a turbulent environment. Thus, formal strategic management procedures are of questionable value for small companies operating in environments where conditions change fast (Artto, Martinsuo, and Aalto 2001). Despite this, long-term planning has been found vital to small companies' growth and development even in turbulent environments (Berry 1998).

Effective planning is crucial for all companies. Don't fool yourself by thinking that you are different.

There are three key values in long-term planning, all of which must be built up independently of each other, but failure in one can endanger all of the potential benefits. This means that the benefits from long-term planning are truly manifested only when all of the key values are present. Why long-term planning is valuable is summarised in Table 2.2 and explained below.

Table 2.2: The key values in doing long-term planning

Key value	**Explanation**	**Challenges addressed by the value**
#1: Intent	The primary driver for doing something. Helps in coordinating a complex set of activities.	What's next? — If we don't know where we want to be, how can we tell which way to go? Unfocussed thinking yields unfocussed action.
#2: Clarity	Explicates the direction of intent for 1) taking action 2) feasibility evaluation and further refinement	How can we know what we really think until we hear what we say? And, if so, how could our organisation fare any better here?
#3: Awareness	Helps in making short-term decisions and trade-offs. Plans are just plans — commit to "a future" but retain the flexibility to notice and adjust to the "real future" as it arrives.	Seeing short-term moves and their potential results as pieces of a larger puzzle. Despite your planning, you are better off if you do not have illusions that you can control the circumstances (or analyse them thoroughly).

Intent means that long-term planning has value in coordinating a complex set of efforts among people — without planning, the overall direction is missing, and chaos prevails over organised efforts. Also, planning serves a symbolic role, being useful as an emotional rallying point and can provide the necessary momentum to reach new objectives. This power of solid intent — summarised in the timeless adage *"As a man thinks, so he is"*[1] — is definitely not to be underestimated. Intent may also serve as a self-fulfilling prophecy.

[1] Proverbs, 23:7

Intent provides the direction and is the source of momentum.

The importance of intent is illustrated by the following discussion at HardSoft (Adapted from (Carroll 2000); in the original version, the CEO is a little girl and the consultant is a cat with an extraordinarily wide smile.)

> Jeff the CEO looked at the product backlog. 'Which of these customer requests should we include in the next release?' he asked. 'Where do you want to be in three years?' responded Jeeves the Consultant wearing a pinstriped suit. 'I'm not sure,' Jeff answered. 'Then,' replied Jeeves, 'why does it matter which ones you choose?'

Clarity is about concretising and communicating the intent as plans so that they can be acted on — or refuted — when necessary. While the senior management may have a good gut feeling of what should be done next, it is important to get it across to the personnel as well. While real 'gut feelings' are by definition something you should follow (Hayashi 2001), there are also other kinds of sentiments that at first may *seem* like authentic intuition, but are twisted by personal bias (due to, e.g., fear or prejudice). In these cases, clarity forces to explicate these feelings, thus allowing the intent as well as the respective course of action to be refined. When clarity is missing, managers and developers are less likely to see the rationale behind the decisions made, or even the point behind the action that is to be taken. Thus, conducting the tasks may be carried out half-heartily or hesitantly. Combined with clarity, having a long-term plan makes it possible for the collective organisational intelligence to get behind the intent.

Clarity makes it possible to act and is also quality assurance for the intent.

Awareness is also about mental clearness, but from a different angle than clarity. It means that having a shared, long-term vision of where the product should be going makes it possible to see short-term moves and their potential results as pieces of a larger puzzle and thus helps in making rational short-term decisions and trade-offs. Especially in dynamic markets, long-term planning is less about gaining insight about the future, but rather guiding present resource usage (Brown and Eisenhardt 2002). Additionally, awareness means you should realise that plans are just plans, and although they are the basis for action, they should not be a straitjacket that limits from adapting to the future as it unfolds (Brown and Eisenhardt 2002). Similarly to a person having a dream, recognising it as such and enjoying the show, awareness solves the dilemma of committing to "a future" while retaining the flexibility to notice and adjust to the "real future" as it arrives.

With awareness you can embrace contradiction.

In summary, when successful, the value of long-term planning is threefold. Intent helps in taking action, thus avoiding the feeling of powerlessness and being overly reactive. Clarity ensures that the intent is communicated and refined when necessary. Awareness disperses the illusion of being able to control circumstances, which is a major potential drawback in doing long-term planning.

Intent provides the momentum, clarity evaluates, refines and communicates it, and awareness helps adapt to changing circumstances.

2.4 Product and Release Planning in Practice

This section discusses the time horizons of product and release planning. Based on this understanding and the terminology from Section 2.2, we present a product roadmapping process you can use as the starting point for product and release planning in your company.

Time Horizons and Related Artifacts in Product and Release Planning

Figure 2.2 presents a summary of how product and release planning connects the perspectives of Business (in other words, the market rhythm, expressed as release cycles) and Development (in other words, product management artifacts, e.g., roadmaps and backlogs). We have found this model helpful in practice for understanding the proper level of detail in planning.

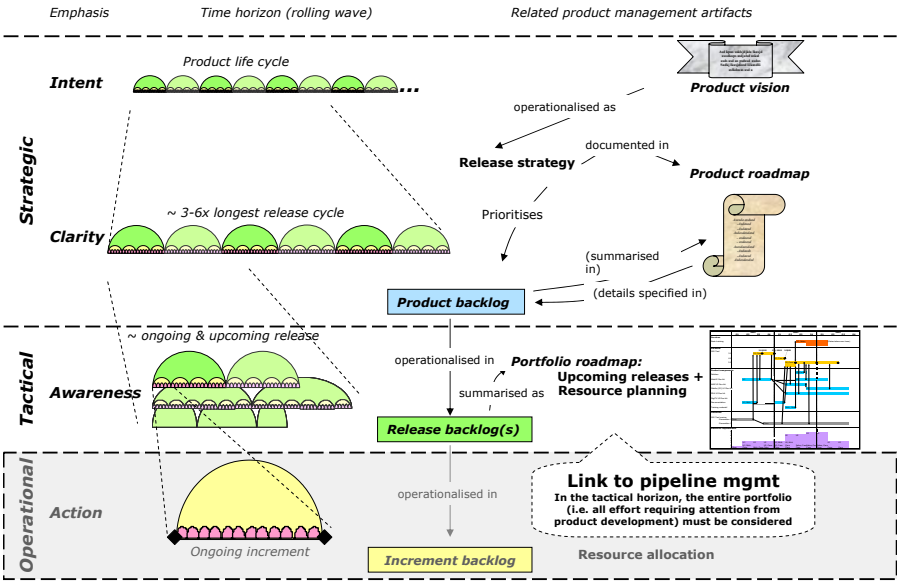


Figure 2.2: Long-term planning, time horizons and related product management artifacts

There are three horizontal layers in Figure 2.2. These represent the major time horizons involved in product and release planning: *strategic*, *tactical* and *operational*. For each time horizon, the emphasis of planning is different and involves different product management artifacts. This section focuses on the strategic and tactical time horizons. The operational layer, including resource allocation and putting out fires (that is, dealing with unexpected situations that demand reaction before the next increment) is discussed more in Chapter 3.

Product and release planning connects the perspectives of Business and Development on strategic, tactical and operational levels.

The level of detail on each planning horizon should comply with the *rolling wave* principle — that is, the nearest release cycles should be more detailed, while less detail is needed for the later ones.

The Strategic Layer: Forming the Vision and Setting the Goals

The strategic layer encompasses two time horizons. The uppermost time horizon in Figure 2.2 emphasises *intent* for the entire life cycle of the product through creating and updating the product vision. The other time horizon on the strategic layer emphasises *clarity* through strategic goal setting. This means operationalising the product vision as a release strategy and documenting it in a product roadmap. The product roadmap can be further specified in the product backlog. The product roadmap may also be updated bottom-up by prioritising the items in the product backlog and allocating them to releases. The time horizon for a product roadmap should be from four to six times the longest release cycle.

A good product roadmap document has three characteristics: first, it should, together with the product vision, clearly communicate product strategy and the rationale behind it in such terms that everybody can understand. Second, it should summarise the product backlog in terms everybody can understand. Third, it should be light enough to be kept easily up-to-date. Despite of how difficult this may sound, a couple of slides may well suffice.

We recommend that the product vision and the product roadmap should be updated simultaneously. A smaller update should be done at the beginning of each release project. A more comprehensive revision taking an in-depth look at the product vision and the release strategy should follow the rhythm of the strategic release management cycle, which in Figure 2.2 is done at the beginning of every other release project.

The strategic layer emphasises intent and clarity through the product vision, the product roadmap, and the product backlog.

The Tactical Layer: Resource Planning and Link to Pipeline Management

In the tactical layer the emphasis of planning shifts to promoting *awareness* through looking at the entire product portfolio and what is feasible to do with available resources and current technologies. The portfolio roadmap links individual product strategies and connects long-term release planning with pipeline management. It summarises release and development schedules, services requiring attention from product development, and planned resource usage in a single document.

The contents for the current and upcoming releases, selected by the Business (the stakeholders of the product from a commercial perspective), receive effort estimates from the Development (the development team or the chief architect), and are further specified in release backlogs. The entire product portfolio is considered and resource planning is done, first based on the resource needs indicated in the product roadmaps, and then refined based on the effort estimates from the release backlogs. A suffi-

cient time horizon is usually up to two times the longest release cycle in the product portfolio.

> The tactical layer emphasises awareness through the portfolio roadmap.
>
> The portfolio roadmap operationalises the long-term plans.

Portfolio roadmapping is more complicated than product roadmapping because of the need for more detailed information. Figure 2.3 presents an example of the portfolio roadmap used at HardSoft. The roadmap communicates the tactical level plans visually, as well as enforces a degree of accuracy through the use of semi-formal notation. The roadmap expresses the release and development schedules for the product(s), the composition of individual releases, changes to the underlying technology, services requiring attention from product development, and planned resource usage. The main point here is to illustrate what kinds of things can be relevant in a portfolio roadmap, not dictate how they should be visualised.

The visualisation consists of five layers, with the top-most four layers depicting the development of various parts of the product offering as *activities*, and the bottom layer showing the planned resource usage at a given moment.

Activities and their planned schedules and effort estimates are presented as horizontal bars. Possible activity types are performing services, preparing releases, and developing modules and platforms. A *product platform* is here a core software asset on top of which the product is built and expanded on, and may be generic enough to be used in other products as well.

Modules are high-level business requirements translated as software, meaning relatively independent (groups of) features. The related business requirements, as well as more detailed information on the functionality should be kept in the requirements management system. Documentation, whether internal or intended for the end-user, is depicted as modules when necessary. As there are no exhaustive rules for discriminating between 'plain' technologies, product platforms, and modules for roadmapping purposes, a reasonable conceptual structure must be resolved case-by-case.

Preparing a *product release* consists of putting together the related modules and platform(s), doing system testing and error correction, as well as performing other product release activities. In the notation there are three kinds of possible releases: major releases, minor releases, and patches. The first diamond in a new release activity denotes a major release and subsequent diamonds and circles mark minor and patch releases, respectively. Only releases on the release layer are visible to customers.

The *services* a company offers are classified and dealt with based on the kind of attention they need from product development. The classification consists of *product accessories*, *customer-specific development services*, and *other services*. Product accessories fulfil a need common to many customers. Typically, they are initially developed for a specific customer, but are to be included as part of the standard offering. Product accessories are expressed in the product roadmap as regular modules integrated into a future release of the product. Customer-specific development services require resources from product development, but their outcome is limited to the customer receiving the service. They are depicted on the service layer. Other services refer to services that at the moment do not appear to require attention from product development. These kinds of services do not have to be included in the product roadmap.

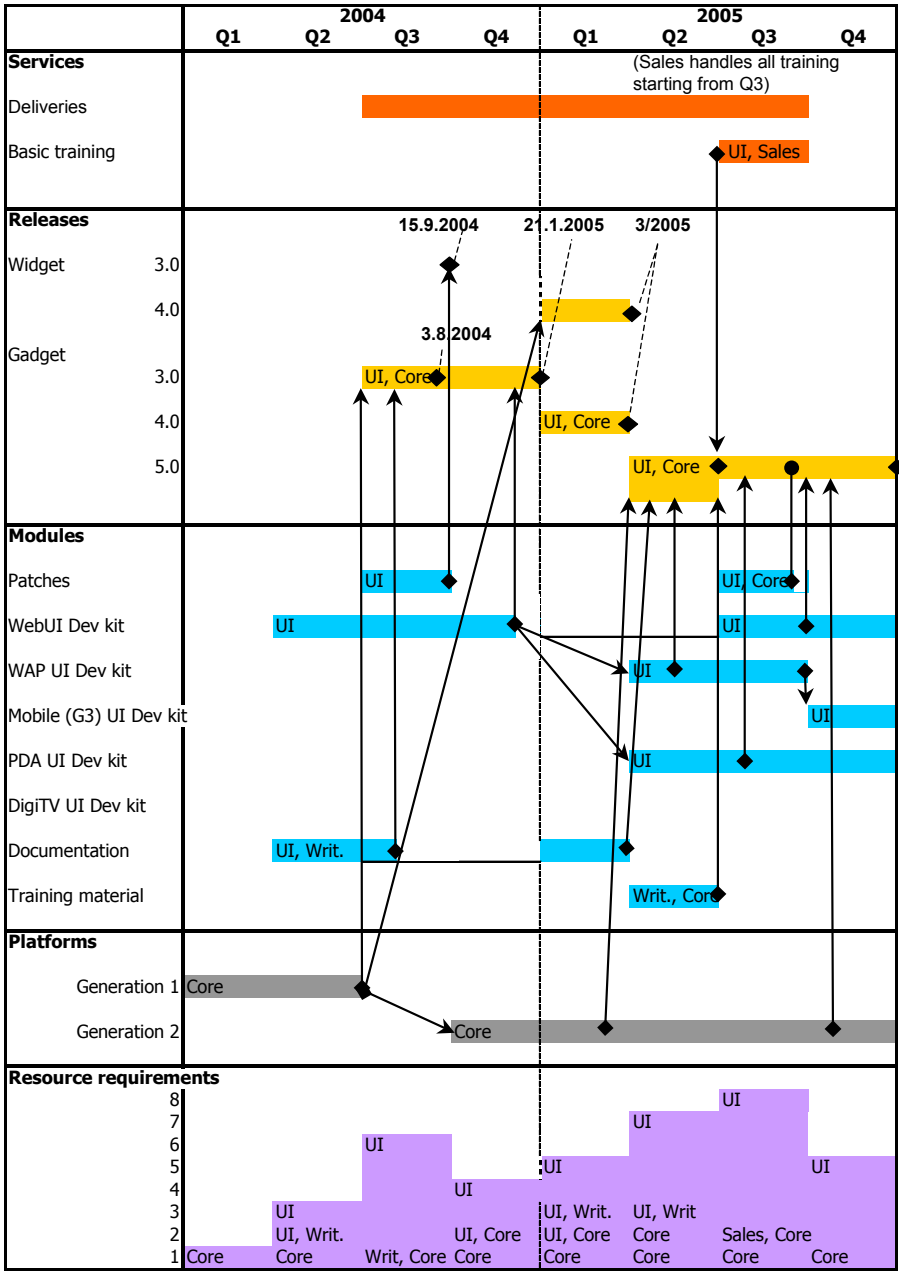


Figure 2.3: An example portfolio roadmap

The thicker an activity is in the visualisation, the more resources are needed for the period indicated. At HardSoft, different people or teams work on the platform and modules. To help in resource planning, textual notation is used inside the activities to denote the allocated *resource type(s)*. Resource information is also summed in the bottom layer of the visualisation.

Arrows going from one activity to another denote both composition and timing regarding integrating the activities' outputs and may, depending on the context, imply reuse as well. Thus, the visualisation communicates the relationships between product releases, components and platforms, and how and when they are built on top of each other.

The example roadmap in Figure 2.3 shows the tactical plans regarding Widget and Gadget. The only complementary service identified to require product development attention is the one-day basic training per license sold, which is to be offered starting from major release 5.0 in 8/2005. Preparing materials for the training is depicted as a module. The other modules are add-in modules for various terminal devices, and end-user documentation that are shipped with the product starting from a minor release in 8/2004. On the platform layer, the roadmap shows two generations of the 'engine' used by both Widget and Gadget, with the second generation to be used starting from Gadget 5.0.

> Experiment on what level of detail is enough in your company for the portfolio roadmap and what issues can be left in the backlogs.

An Example Process for Product and Release Planning

Product and release planning identifies, evaluates, and selects between identified alternatives to achieving desired objectives. Product roadmapping is an approach to product and release planning. The resulting roadmaps' implementability is as important as their strategic value.

> "Plans are nothing; planning is everything" (Dwight D. Eisenhower)

The basic steps of the product roadmapping process used at HardSoft for conducting product and release planning are summarised in Table 2.3 and described below.

The first step is to define (or revise) the product vision in the light of the mission and vision of the company. All companies, no matter how small, should have an idea of their purpose and desired future, which is clear enough to be written down, before they plan their operations in more detail. Often some kind of product vision exists even if the company's mission and vision are not explicitly defined. A company's mission and vision should act as guidelines for shaping the product vision and making trade-offs. Looking back at Figure 2.2, this step addresses the upper half of the strategic layer.

The second step is to identify major trends in the environment. This encompasses looking at potential customers, competitors, the industry, and trends in relevant technology. Many well-known models and techniques, such as Porter's five forces, strategic group analysis, and competitor profiling (Johnson and Scholes 1993) can be used to steer the management's attention. This should result in an understanding of the de-

Table 2.3: Steps in product and release planning

Step	**Objective**
#1 Create (update) the product vision	Clarify and communicate what business the company is in and specify how this product fits in. Define and document the product vision.
#2 Scan the environment	Scan the environment. Choose position and focus, assess the realism of the product vision and examine what technologies should be used
#3 Create (update) the product roadmap	Revise and distil the product vision as a product roadmap. Establish the release cycle, set objectives for releases and plan for needed resources. Record decision rationale as business requirements or business cases.
#4 Sanity check	Assess whether the planned development is in accordance with the product vision. Estimate the product life cycle and respective cash flows.
#5 Create (update) the portfolio roadmap	Consider the product roadmap together with the entire mix of development efforts planned, and adjust the upcoming release as necessary.

sired focus and position for the product as well as guide in technology selection. The product vision should be revised based on the analysis conducted.

In the third step, the product vision is concretised as the product roadmap. This means establishing the release cycle, setting goals for future releases in terms of business objectives (such as product life cycle and respective cash flows), and looking at the amount and types of resources needed. Decision rationale should be recorded as, e.g., business requirements or business cases. By including business requirements and their objectives explicitly into the product backlog and keeping track of their history, the rationale behind the roadmap's evolution becomes transparent. Scenario thinking (van der Heijden 1996) may also be of help, if choosing between alternatives is difficult.

The fourth step is to state expectations regarding the life cycles and financial implications of product releases, components and platforms, and consider the mix of planned development activities from the business objectives' perspective. This acts as a sanity check and evaluates whether the planned development is in accordance with the product and company vision.

The fifth step is to update the portfolio roadmap. Updating the portfolio roadmap should start from the major and minor release cycles and continue with further clarifying the business requirements and expectations for the upcoming releases, taking internal factors such as other products, human and financial resources, competences, and infrastructure into account.

References

Afuah, A. and C. L. Tucci. 2002. *Internet business models and strategies — text and cases*. Singapore: McGraw-Hill Higher Education.

Artto, K., M. Martinsuo, and T. Aalto. 2001. *Project portfolio management — strate-*

gic management through projects. Helsinki: Project Management Association Finland.

Berry, M. 1998. Strategic planning in small high-tech companies. *Long Range Planning* 31 (3): 455-466.

Brouthers, K. D., F. Andriessen, and I. Nicolaes. 1998. Driving blind: strategic decision making in small companies. *Long Range Planning* 31 (1): 130-138.

Brown, N. and K. M. Eisenhardt. 2002. *Competing on the Edge: Strategy as Structured Chaos*. United States of America: Harvard Business School Press.

A classic on the importance of balancing flexibility and control through time pacing.

Carroll, L. 2000. *Alice's Adventures in Wonderland and Through the Looking Glass*. United States of America: Signet Classic.

Cusumano, M. A. and R. W. Selby. 1995. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People*. New York: The Free Press.

A classic on the relationship between business strategies and software development.

Cusumano, M. A. and D. Yoffie. 1998. *Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft*. New York: The Free Press.

Dver, A. S. 2003. *Software product management essentials — a practical guide for small and mid-sized companies*. Tampa: Anclote Press.

One of the first books on commercial software product management, and at the time of writing this, one of the best ones as well; recommended.

Hayashi, A. M. 2001. When to trust your gut. *Harvard Business Review* 79 (2): 59-65.

van der Heijden, K. 1996. *Scenarios: the art of strategic conversation*. London: John Wiley & Sons.

Hitt, M., D. Ireland, and R. Hoskisson. 1997. *Strategic Management: Competitiveness and Globalization: Concepts.* 2nd ed. St. Paul, MN: West Publishing Company.

Hohmann, L. 2003. *Beyond software architecture: creating and sustaining winning solutions*. United States of America: Pearson Education.

Like the title says, goes beyond software architecture and into commercial product management and the Business-Development interaction issues; recommended.

Johnson, G. and K. Scholes. 1993. *Exploring Corporate Strategy.* 3rd ed. Hempstead: Prentice Hall International.

Kroll, P. and P. Kruchten. 2003. *The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process*. 1st ed. United States of America: Addison-Wesley.

Linder, J. C. and S. Cantrell. 2002. Five business-model myths that hold companies back. In *IEEE Engineering Management Review* 30 (3): 26-31.

Martin, C. 2002. *Managing for the short term — the new rules for running a business in a day-to-day world*. 1st ed. New York: Doubleday.

Bridges the illusionary gap between planning for the long term and acting soundly for the short term. However, do not let Martin's use of the terms calendar-driven and event-driven (p. 66) confuse you — the basic idea is the same as ours.

McCarthy, J. and M. McCarthy. 2002. *Software for your head — core protocols for creating and maintaining shared vision*. United States of America: Pearson Education.

An innovative set of procedures and patterns for facilitating teamwork; highly recommended!

McDermott, P. 2003. *Zen and the art of systems analysis*. 2nd ed. Lincoln: Writers Club Press.

For those who like to think "out of the box" every now and then.

Mello, S. 2002. *Customer-centric product definition — the key to great product development*. United States of America: American Management Association.

Mintzberg, H., B. Ahlstrand, and J. Lampel. 1998. *Strategy Safari : A Guided Tour Through the Wilds of Strategic Management*. New York: Simon & Schuster.

Katajala, J., T. Koskinen, K. Männistö, P. Noponen, R. Räisänen, T. Saiki, J. Välimäki. Roadmapping Tool [A graphical tool developed as a project assignment on the course T-76.115 Software Project at the Helsinki University of Technology during 2003-2004]. [Referenced April 13^{th}, .2004]. Available under an open source license from `http://www.katajala.net/t76115/docs/DD/delivery.html`.

Näsi, J. and M. Aunola. 2002. *Strategisen johtamisen teoria ja käytäntö*. Helsinki: Metalliteollisuuden keskusliitto MET.

Recommended for a concise, sober and in-depth discussion on strategic management (for Finnish-speaking readers).

Naumanen, M. 2002. *Nuorten teknologiayritysten menestystekijät*. Helsinki: SITRA.

A bilingual guide for product and technology roadmapping aimed at practitioners.

Penny, D. A. 2002. An estimation-based management framework for enhancive maintenance in commercial software products. *In proceedings of IEEE International Conference on Software Maintenance 2002 (ICSM'02)*: 122-130.

Phaal, R., C. J. P. Farrukh, and D. R. Probert. 2004a. Customizing roadmapping. *Research Technology Management* 47 (2): 26-37.

Advice for creating your own roadmapping process.

Phaal, R., C. J. P. Farrukh, and D. R. Probert. 2004b. Technology roadmapping — a planning framework for evolution and revolution. *Technological Forecasting & Social Change* 71 (1-2): 5-26.

Sums up research on roadmapping; read this for an overview on the topic.

Rajala, R., M. Rossi, V. K. Tuunainen, and S. Korri. 2001. *Software Business Models: A Framework for Analysing Software Industry*. Tekes, Technical Review 108/2001. Helsinki: Tekes.

Rajala, R., M. Rossi, and V. K. Tuunainen. 2003. A framework for analysing software business models. In *Proceedings of 11th European Conference on Information (ECIS2003)*

Business model encompasses more than just how you make money. This paper explains what else should be considered and why.

Rautiainen, K., C. Lassenius, and R. Sulonen. 2002. 4CC: A Framework for Managing Software Product Development. *Engineering Management Journal* 14 (2): 27-32.

Smith, J. A. 1998. Strategies for start-ups. *Long Range Planning* 31 (6): 857-72.

Vähäniitty, J. 2003. Key Decisions in Strategic New Product Development for Small Software Product Businesses. *In proceedings of 29th IEEE EUROMICRO Conference on Software Process and Product Improvement*: 275-383.

Vähäniitty, J., C. Lassenius, and K. Rautiainen. 2002. An Approach to Product Roadmapping in Small Software Product Businesses. Center for Excellence Finland, *Conference Notes.*, Helsinki: 12-13

Wheelen, T. L. and D. J. Hunger. 2002. *Strategic management and business policy*. 8th ed. United Kingdom: Pearson Education.

Wiegers, K. 1999. *Software Requirements*. United States of America: Microsoft Press.

Chapter 3

Pipeline Management

Jarno Vähäniitty

3.1 Introduction

Pipeline management means looking at the portfolio of development activities as a whole and making appropriate decisions on resource usage in a timely manner. Pipeline management answers the questions who, when, and with what priorities. The term 'pipeline' comes from manufacturing, and refers to the production line, both in terms of work-in-progress and resourcing — the things that are resourced currently (or in the future).

This chapter consists of three sections. We start by explaining the importance of understanding the big picture of resource spending and present four principles of successful pipeline management. Then we examine the different types of effort developers in small software companies are responsible for. We note how different types of effort should be managed differently and provide guidelines for distinguishing the effort types at your company. Finally, we relate pipeline management to development rhythm and present how the pipeline management process can help synchronise the different types of product development efforts.

3.2 Pipeline Management Principles — Understanding the Big Picture

Pipeline management is crucial because product strategy is realised only through actual spending of resources in various development activities. Product strategies set the direction for individual product development activities, but it is up to pipeline management to understand and uphold the big picture, and ensure that at each moment, the most important activities get done.

> Strategy is realised through resource spending; resource spending is governed by the pipeline management process.

In small software companies, the developers' attention is divided between developing new product releases, doing customer-specific development, customer deliveries, and performing other services. We use the term *product development portfolio*

or *portfolio of (product) development efforts* to refer to all of the tasks requiring the developers' attention, whether these actually involve software development as such or not. Pipeline management consists of making go/kill/hold decisions for individual product development activities on an ongoing basis (e.g. between each increment), as well as conducting periodic reviews of the entire portfolio.

Below is an example illustrating some of the problems that process consultant Jeeves observed at HardSoft at the start of their cooperation. If your company exhibits symptoms such as these, your pipeline management process could need improving.

> Important product development decisions were often made based on the opinions of key personnel and without explicit discussion or justification. Because of unclear and apparently shifting priorities, overbooking of resources was common. Some important activities such as long- term product and release planning did not receive enough attention. Although important portfolio decisions were made, they were not taken deliberately, but rather inadvertently, or even through inaction. Likewise, the key personnel were not always aware of the gamut of important decisions they in effect were making.

While pipeline management is essentially about resource allocation, the inherent complexity of the issues involved keeps it far from being a mechanistic exercise. Successful pipeline management achieves a balance between the four potentially conflicting principles of

1. *maximising the financial value* of the portfolio of product development efforts,
2. *balancing* the portfolio on relevant dimensions,
3. ensuring *strategic direction*, and
4. ensuring that the *total number of ongoing activities is feasible*. (Cooper, Edgett, and Kleinschmidt 2002b)

Also, like in Section 2.3, all of the principles must be followed for successful pipeline management.

Value maximisation means allocating resources in order to maximise the value of the activities in the product development portfolio. That is, you select projects to maximise the sum of the values of commercial worth of the development efforts in terms of some business objective, such as profitability, economic value-added, or return on investment. (Cooper, Edgett, and Kleinschmidt 2002b) Maximising portfolio value can be supported by investment calculations and other financially based methods and scoring models that build the desired objectives into weighted criteria lists (Artto, Martinsuo, and Aalto 2001).

Balancing the portfolio means achieving a desired balance of development efforts in terms of relevant parameters (Cooper, Edgett, and Kleinschmidt 2002b). Typical parameters on which a company's portfolio should be balanced are short versus long term, current versus new platform(s), high versus low risk, research versus development, focus versus diversification, and resource allocation between development efforts (Artto, Martinsuo, and Aalto 2001; McGrath 2000).

Strategic direction means ensuring that regardless of value maximisation and balance, the final portfolio of development efforts reflects the company's strategy. The link to strategy can be tuned, for example, by applying portfolio reviews and building strategic criteria into scoring and prioritisation models, project selection tools or go/kill models, or by setting aside resources for different types of projects.

Making sure that the **number of ongoing activities is feasible** is emphasised because the opposite situation is quite common (Cooper, Edgett, and Kleinschmidt

2002b). Having too many ongoing activities results in pipeline gridlock, where projects end up in a queue, take longer and longer to finish, and key activities such as doing planning are omitted because of a lack of people and time. Thus, this principle means that the pipeline management process should actively observe the balance between required and available resources and have mechanisms for keeping the pipeline optimally loaded.

As mentioned earlier, the four principles of successful pipeline management are potentially conflicting. The portfolio that yields the greatest net present value may not be a very balanced one. For example, it may consist of short-term and low-risk projects that are overly focussed on certain customers. Likewise, a portfolio that emphasises strategic direction may sacrifice other goals such as short-term profitability. A great variety of methods, tools and approaches to managing new product development have been developed, and the most successful ones utilise the principles described above (Cooper, Edgett, and Kleinschmidt 2001). Look at the references at the end of this chapter for methods and techniques to support pipeline management. Be advised that our term pipeline management closely corresponds to the term portfolio management in literature. Also, be aware that models and techniques usually emphasise some of the four principles more than others.

3.3 Managing Different Types of Development Effort

In large companies, pipeline management means deciding on a multitude of development project opportunities concerning different product lines and their extensions, possibly across several business units (Artto, Martinsuo, and Aalto 2001). In contrast, pipeline management in small product-oriented software companies emphasises the contents of upcoming releases for relatively few products. However, because pipeline management in small companies also involves resourcing different kinds of development-like or even non-development effort requiring attention from the developers, it is not clear that pipeline management is actually any less complex in this context. Consider the following observation made by Jeeves the Process Consultant at HardSoft illustrating the challenges in resource planning and allocation due to the multiple roles and responsibilities of developers:

> The developers' attention was divided between developing new product releases, doing customer-specific development, making customer deliveries, and performing other services. These activities were not necessarily explicitly managed as projects or even recognised as part of the product development portfolio. This complicated resource planning, often to the degree that it was more or less omitted because of its perceived futility.

Besides the fact that different kinds of activities compete for the developers' attention, these activities should be managed differently. For example, a proper process for developing a new major release is likely to emphasise different things than a process for conducting customer-specific tailoring, not to mention the process for making customer deliveries or training the customers. While you might argue that the last two are not "development effort", it is best to define the product development portfolio so that it includes all of the activities that require attention from the product developers.

Different types of development effort need to be managed differently.

There should be a proper balance in how the available resources are spent on the different types of development effort. For example, if your product development is funded based on sales revenue, having sources of revenue besides licence sales might be a good idea. You could consciously develop new features for a specific customer first, and then include the results as part of a later product release. Or, some of your developers could do consultancy work or body shopping to fund your product development efforts.

> Explicating the types of development effort is the first step in establishing pipeline management.

Summarising from above, managing different types of development effort comprises of answering three questions.

1. How do I recognise the types of effort that comprise my product development portfolio?
2. How should I manage each of them in terms of rhythm?
3. How do I establish, communicate, and track the proper balance between the effort types?

We now address each of these questions in turn and show how the Cycles of Control framework can be used to communicate the answers.

Distinguishing Between Effort Types

There are two main drivers that affect whether you should distinguish one effort type from another from the management perspective. These are the *nature of the work* itself and the *degree of customer involvement*. These are illustrated by the following reasoning as told by HardSoft's development team leader Jill:

> "There are basically five things developers can work on: platform development, application development, deliveries, tailoring, and body shopping. However, the developers have been doing less and less body shopping during the last year, and it is likely that in the future we are so well off thanks to increased license sales that we do not have to make body shopping contracts. Thus, I ended up with four potential effort types to start with. Next, I considered the degree of customer involvement. Of these, platform development and application development do not directly involve customers, but deliveries and tailoring do. I figured that maybe two effort types would suffice, one for internal development and the other for those involving customers. However, when I reconsidered the nature of the work, customer deliveries were much more clear-cut and involved less effort than tailoring projects. Also, because deliveries often spawned tailoring projects, I figured that distinguishing between these two was a good idea. With respect to platform and application development, it seemed that there was a need to make internal releases of the platform on a more frequent basis than making external releases of the applications, and this difference in rhythm suggested a distinction between these types of effort."

Thus, Jill ended up with four development effort types, illustrated in Figure 3.1.

Figure 3.1: Types of development identified at HardSoft

Figuring out Proper Rhythms for the Effort Types

Next Jill considered how the four effort types should be managed:

> "I recalled that our process consultant Jeeves talked a lot about the importance of rhythm, so I started thinking about the proper project length for each of the four effort types. Considering our market rhythm together with Jeff (the CEO), we agreed that we should be able to make releases quarterly, but things do not change so fast that having one-month increments would be too long a time to react. Thus, I ended up with the 3-month release/1-month increment rhythm for application development. The rhythm for platform development was also derived from the demands of its 'markets', in other words the application development. As I discussed with Jeremy and Jay (product managers of Widget and Gadget, respectively), we agreed that because the development of both Widget and Gadget depend on what the platform is capable of doing, internal platform releases should be made on a more frequent basis. We ended up with a model where the platform is developed in two-week increments, where every increment produces a version for internal use, and for every three months, a new version of the platform is released, stable enough to be included in customer deliveries. Delivery projects are typically one month in length from the signing of the deal to the closeout. However, they commonly have periods of no-activity. Also, deliveries are nowadays simple enough that we'll try them without explicit internal control points. Tailoring, however, typically takes longer, but we haven't yet gotten around to deciding whether they should have internal control points or not."

The rhythms that Jill ended up with are illustrated in Figure 3.2.

Balancing Resource Spending Between Effort Types

Establishing target spending levels for effort types means deciding how much resources in an ideal case should be spent on the identified development effort types. Establishing target spending levels and observing actual resource usage is a practical way of bringing many of the pipeline management principles mentioned above into practice. At HardSoft, target spending levels were decided to be 30% for platform development, 40% for application development, 10% for deliveries and 20% for tailoring. This, as well as the result of the entire development effort type identification exercise at HardSoft is summarised in Figure 3.2.

3.4 Synchronising the Development Portfolio

Different types of development effort should have their Business-Development control points tailored to the nature of the activities. For example, control points for internal development of the product platform, making different types of releases, and

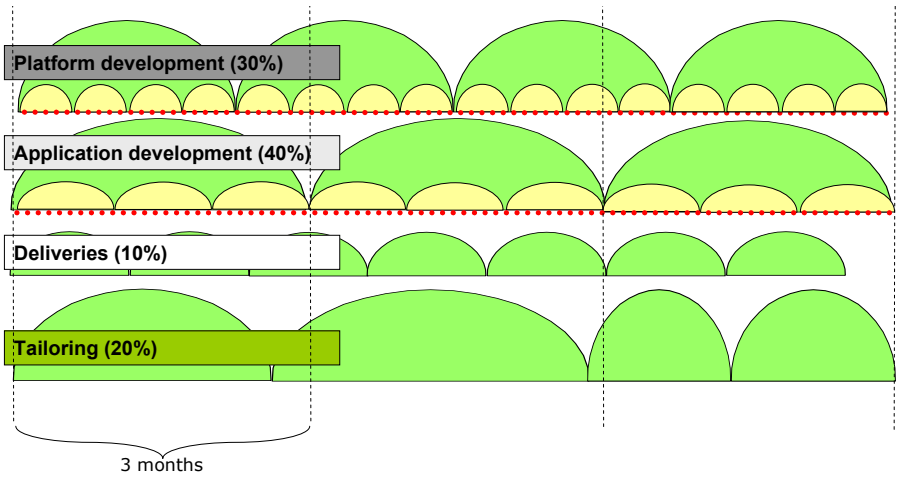


Figure 3.2: Development types, respective processes and targeted spending

doing customer deliveries (possibly including integration and tailoring) should in most cases emphasise different things. Also, the agenda of the control points should vary depending on when the control point occurs compared to the life cycle of the project.

This means that the strategic release management process can in practice get quite complicated due to the number of meetings requiring attention from Business. Consider the following example as told by the CEO of HardSoft, Jeff:

> "So, we now understood that our development efforts must have a rhythm, and this rhythm stems from having control points. We identified our types of development effort as *platform development*, *application development*, *deliveries*, and *tailoring*. This helped a lot in making things clearer regarding what the developers were actually spending their time on and for what purposes. However, the Business now had *a lot* of new meetings to attend to in the form of control points! And while each of these meetings was short, they clogged our calendars with a steady rhythm (laughs). Besides, the sales directors for Widget and Gadget, Jermaine and Joanna, not to mention our marketing director Jericho were really busy to start with! So, from the perspective of the Development, the new process was working fine, but for the Business, it was quite exhaustive. Not wanting to slip into the old ways, we, together with Jeeves (the process consultant), came up with *development portfolio synchronisation* when we realised that the increment cycles could be made similar for each of the development types."

Figure 3.3 illustrates the problem described by Jeff. In this example, deliveries and tailoring do not have a clear internal rhythm (besides the fact that they are organised as projects).

An out-of-sync problem also leads to problems in resource planning and allocation on the tactical and operational levels (see Figure 2.2). When push comes to shove, the development types with close customer involvement tend to override platform and application development, which in turn makes product and release planning more difficult. This is illustrated by the following quote from HardSoft's development team leader Jill:

> "Earlier, the pressures from making deliveries and doing tailoring made resource planning for platform and application development very difficult. Although we

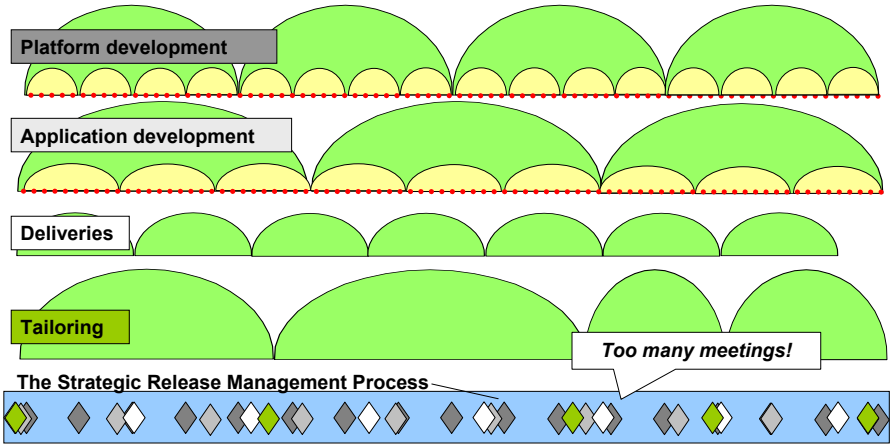


Figure 3.3: An out-of-sync portfolio of development efforts

> wanted to make new releases approximately two or three times a year, we could not know in advance how much of the developers' time deliveries and tailoring would take. Since long-term product and release planning required some knowledge of the availability of resources — the one thing that we didn't know — we more or less neglected long-term planning as well."

The product development portfolio must be synchronised for strategic release management.

Synchronising the development portfolio means organising the control points for different types of development effort so that the overall strategic release management process becomes as simple as possible for both the Business and the Development. In practice, this can be done through grouping the increment and project level control points for individual development projects into *portfolio reviews*.

The resulting portfolio reviews effectively set the rhythm for the entire development organisation. Thus, portfolio reviews are for the entire development organisation what the control points are for an individual development project. They make it possible to dedicate the resources and aim for a certain scope for the next increment. For this to succeed, the rhythm of the entire portfolio of development efforts must, at least on the increment level, be similar. In other words, in the case of HardSoft deliveries and tailoring would not by themselves require increment level control points. However, to enable portfolio synchronisation, deliveries and tailoring must adhere to an increment level synchronisation, where all development effort is considered. Figure 3.4 illustrates how HardSoft does this in practice.

Strategic release management extends the Cycles of Control to cover the entire development portfolio.

As shown in Figure 3.4, the *strategic release management process* (see next sec-

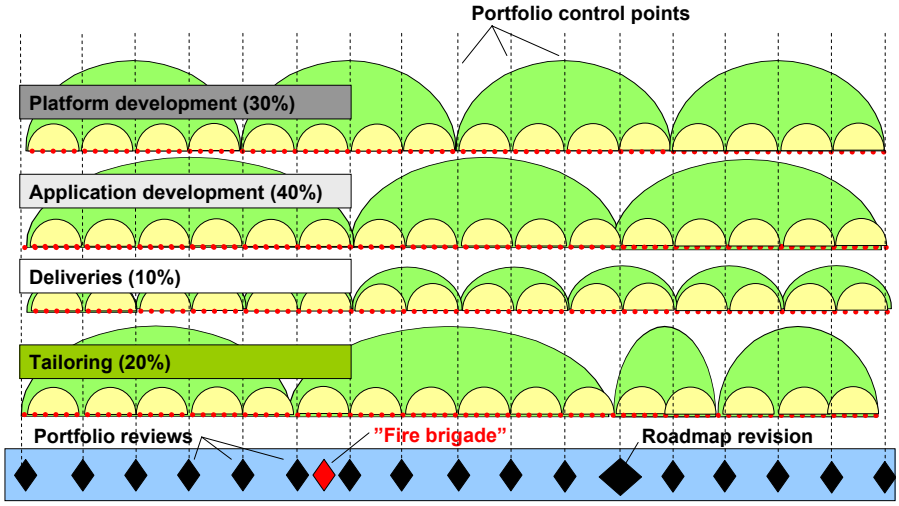


Figure 3.4: A synchronised development portfolio with target spending levels

tion) is paced according to the increment rhythm. Also, in addition to portfolio reviews, there are two other kinds of strategic release management control points, *roadmap revisions* and *fire brigades*.

> Product strategy is formulated and enacted through roadmap revisions, portfolio reviews, and fire brigades.

3.5 The Strategic Release Management Process

Together, product and release planning and pipeline management constitute the strategic release management process. This is illustrated below in Figure 3.5.

The strategic release management process links product and release planning and pipeline management through synchronising the product development portfolio in the portfolio control points. Next, we discuss these control points more closely.

> The strategic release management process links product and release planning and pipeline management.

We start by listing and describing the attributes of a *generic* control point in Table 3.1. Then we present the strategic release management control points at HardSoft. Besides the strategic release management process, the attributes in Table 3.1 are applicable to project, increment, and heartbeat level control points, and can be helpful in designing and documenting them.

At HardSoft, portfolio reviews occurring before the beginning of the company-wide increment cycle, are the primary mechanism for linking strategy with operations. In other words, portfolio reviews update the portfolio roadmap according to the input from the operational and the strategic layers (see Figure 2.2). Thus, portfolio

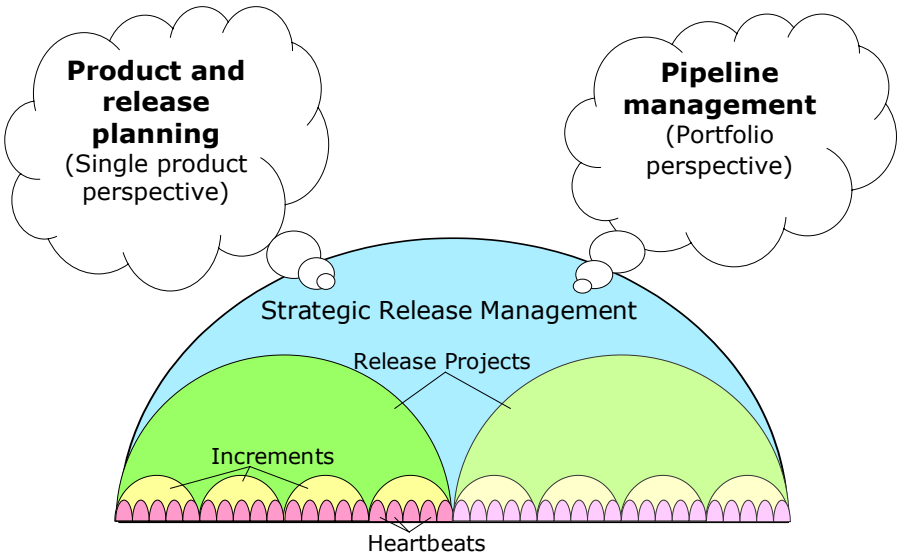


Figure 3.5: Components of the strategic release management process

reviews are like increment planning meetings, which instead of looking at a single development activity, take a look at the portfolio as a whole and decide on the contents and resource usage for the upcoming set of development increments. The agenda of portfolio reviews consists of reviewing the activities in the development portfolio as dictated by the control points of the respective development effort types involved. In addition to the roles whose presence is already required by the development control points (being the sales directors and product managers for both products, as well as the development team leader), Jeff the CEO is either present or represented by John the visionary or Jericho the marketing director. As a result of a portfolio review, product, project, and increment backlogs are updated to reflect the new situation as well as the resources allocated, and any changes that propagate to the product roadmap or product vision are updated there.

> Portfolio reviews are time-paced groupings of product development control points.

Fire brigades are event-triggered portfolio reviews. While the objective is to freeze the resources and set a target scope for the increments in portfolio reviews, business realities often make 100% adherence to this principle impossible in practice — sometimes you just have to react even though it causes mid-increment changes. Fire brigades are HardSoft's way of doing this systematically, and they are essentially the same as portfolio reviews, but with two exceptions. First, the decisions made can only affect the goals and the resources of the ongoing increments. Thus, updating the product backlog or the roadmaps is not necessary. Second, the reason (and the root causes that led to this reason) for calling the fire brigade is documented, as well as the resulting decisions. This makes it easier in the future to spot similar situations in advance (thus making it possible to account for them already in the "previous" portfolio

Table 3.1: Attributes of a generic control point

Attribute	**Explanation**
Name	A descriptive name of the control point. For example, a "portfolio review", "increment demo", "planning game", "roadmap revision", "scrum", etc.
Purpose	An explanation of why the control point is important and how it relates to other control points and the overall process. Control points can consist of other control points, for example, a portfolio review control point typically includes going through gate-type control points (for example increment planning meetings) of individual development activities.
Rhythm and timing	Rhythm refers to how often the control point is to occur, and *timing* complements this information by relating the occurrences to other control points or events. For example, a product management team meeting is occurring every two weeks on Wednesday afternoons, while the developers' weekly meeting is every Wednesday morning. This way, the latest information is available for the product management team.
Agenda, input needed, and related checklists	Agenda describes what issues are to be covered in the control point. Input needed refers to what information is needed (beforehand) so that these issues can be appropriately dealt with. It is often useful to document these as checklists.
Responsibilities involved (and associated roles)	Responsibilities refer to who should be involved in the control point as well as how they should be involved. Associated roles refers to this in terms of the (company-specific) grouping of responsibilities into roles.
Related documents and tools	The documents / tools that are used as the input and/or updated as the result of the control point.

review). It also provides a baseline for estimating how often a need to put out a fire is likely to occur, which in turn helps make more realistic portfolio roadmaps.

> Within-increment resource allocation should only be changed by the Fire brigade.

Roadmap revisions focus on the strategic instead of the tactical layer (see Figure 2.2). Although at HardSoft, the rhythm of roadmap revisions for both Widget and Gadget is the same (every 8th increment), you should consider whether a different roadmap revision rhythm is appropriate for different products. Roadmap revisions at HardSoft include a portfolio review, meaning that first the previous increment is closed, then roadmap revision occurs, and based on the updated long-term plans, the second half of the portfolio review takes place. Thus, in addition to the regular portfolio review, the agenda for roadmap revisions is for each product going through the product roadmapping process described in Section 2.4. The involved roles are

the same as for portfolio reviews, except that Jeff the CEO, John the Visionary and Jericho the Marketing director are all present, and from development, chief architect Jenny joins the group. As a result of a roadmap revision, the product visions, product roadmaps, and product backlogs are updated to reflect the current understanding, concretised into the portfolio roadmap, and discussed with the entire company.

> Roadmap revisions are portfolio reviews extended to cover the strategic time horizon as well.

References

Artto, K., M. Martinsuo, and T. Aalto. 2001. *Project Portfolio Management — strategic management through projects*. Helsinki: Project Management Association Finland.

Cooper, R. G., S. J. Edgett, and E. J. Kleinschmidt. 2001. *Portfolio Management for New Products*. 2nd ed. Cambridge: Perseus Books.

State-of-the-art book on product portfolio and pipeline management, including theory, techniques and tools. Recommended, despite the fact that it is written for large companies whose products involve manufacturing.

Cooper, R. G., S. J. Edgett, and E. J. Kleinschmidt. 2002a. Optimizing the Stage-Gate Process: What Best-Practice Companies Do. *Research-Technology Management* 45 (6): 43-49.

State-of-the-art article describing the Stage-Gate process for implementing product and pipeline management; recommended.

Cooper, R. G., S. J. Edgett, and E. J. Kleinschmidt. 2002b. Portfolio management: fundamental to new product success. In *The PDMA Toolbook for New Product Development*, eds. P. Belliveau, A. Griffin, and S. Somermeyer, 331-64. New York: John Wiley & Sons.

Summarises the key points from (Cooper, Edgett, and Kleinschmidt 2001); read this as your first primer into portfolio and pipeline management.

McGrath, M. 2000. *Product Strategy for High Technology Companies*. New York: McGraw Hill Text.

An in-depth discussion on formulating product strategies. Addresses timing, technological change, globalisation, product differentiation, cost and price, contingency planning, marketing and financial considerations. Written for large companies whose products involve manufacturing.

McGrath, M. 1996. *Setting the PACE in product development*. Newton, MA: Butterworth-Heinemann.

A concise primer on managing new product development. Written for large companies whose products involve manufacturing.

Vähäniitty, J. Key Decisions in Strategic New Product Development for Small Software Product Businesses. In *Proceedings of the 29th IEEE EUROMICRO Conference*

on Software Process and Product Improvement: 275-83.

Vähäniitty, J. 2004. Product Portfolio Management in Small Software Product Businesses — a Tentative Research Agenda. Forthcoming in *Proceedings of the 6th International Workshop on Economic-Driven Software Engineering Research (EDSER-6), ICSE 2004*.

Outlines the challenges in implementing pipeline and portfolio management as explained in the literature new product development in the case of small software product businesses.

Chapter 4

Software Design and Implementation

Mika Mäntylä

4.1 Introduction

This chapter discusses how to organise design and implementation of software products in time-paced software development. The term design refers broadly to the design or architecture of the software and to the planning of how to implement functionality. The term implementation refers to the actual coding of functionality, not including testing, which is discussed in Chapter 5. An integral part of software product development is software evolution, which is also discussed in this chapter.

Because of the multitude of specific practices for design and implementation, only some of them that we consider important are described in this chapter. There are several books and articles covering these practices as well as the broad topic of this chapter and we point the reader to them where appropriate.

In this chapter, we first discuss the importance of and basis for software product design, followed by a presentation of design principles and practices. Then we discuss implementation, specifically how to organise and manage it. Following this, we present two potentially beneficial implementation practices: pair programming and refactoring. In the fourth section of this chapter we discuss software evolution, including the very practical concept of technical debt. Following this, we discuss how time pacing influences design and implementation. We describe how the CoC framework can be applied to design and implementation.

4.2 Software Product Design

Software product design or architecture can be one of the keys to success. Good software product design matters, especially in the long run when you year after year reap the benefits from your good solutions. Good product design and architecture should be viewed as an investment or a company asset. Below we have listed some reasons why good product design is needed (Hohmann 2003; Jacobson, Booch, and Rumbaugh 1998)

Organisation. Development is often organised based on the software architecture, where teams are formed around the architecture, e.g., a GUI team, a platform

team, and so on[1]. Software architecture can reduce the communication need between the developers, e.g., GUI developers do not need to know details about the platform implementation and vice versa, as long as the interfaces between the two are clear.

Evolution. Successful software architectures outlast their creators. Good architecture can last for up to 30 years. During the course of software evolution changes have to be implemented to the software. Making the evolutionary changes is much easier if the architecture is good.

Reuse. With good architecture there is no need to implement anything more than once. The existing architecture can offer developers a great deal of tested functionality that can be reused.

Understanding. Good software architecture makes the software easier to understand. This helps the communication for everyone involved. Communicating and understanding the product and the problem it solves can greatly increase the success of the product.

The Basis for Software Product Design and Architecture

There are four building blocks that form the basis for successful software product architecture. *Domain knowledge* provides you insight about the real world where the software product is used. A *predecessor system* gives you one solution that shows how such software can be implemented and operated. *Developer competence* also greatly affects the software product design and architecture. Finally, the *requirements* for the software tell you what needs to be implemented. We are not saying that these four building blocks are in any way orthogonal and we are aware that they are overlapping. For example, domain knowledge helps you in brainstorming the requirements and so on. Still, being aware of these building blocks helps you in making the right architecture decisions.

Domain Knowledge

Domain knowledge is essential in creating successful software products and it makes software design much easier. Domain knowledge helps you make the right decisions when choosing between design alternatives and it gives you a vision of what properties the system needs now and in the future. The example below clarifies this issue.

> A small software company, HardSoft, aims for the software product market. It has recently been forced to generate revenue through customer-specific projects. The company has been selected to create pulp mill control software for PAM-Seven's new pulping machines. Unfortunately Jezebel, the employee of HardSoft that negotiated the deal, has just left the company. Jezebel also possessed most of HardSoft's knowledge about the pulping industry. Therefore, HardSoft now has problems understanding the domain. This increases the risk of misunderstanding some of the requirements, which will make the project more expensive. It is also possible that HardSoft misses future business opportunities by making poor design decisions. It might be possible to create the pulp mill control software so that it could be sold to other pulp machine makers. To create such software HardSoft needs to understand what features and system capabilities are specific for PAM-Seven's pulping machine and what can be used in other pulping machines. For example, PAM-Seven could have a requirement that all pulping

[1] Sometimes the architecture is based on the organisation of the development and not the other way around. This can however lead to an architecture that is not suitable for the product.

machines that have feature A also need to have feature B. Some other manufacturer's machine might require feature C instead of features A+B. Without domain knowledge HardSoft's software will likely be specific to PAM-Seven's pulping machines. It is also very likely that without domain knowledge HardSoft would build unnecessary features and capabilities, because they would make wrong assumptions about the importance of the features and capabilities in the software.

Domain knowledge can greatly reduce the effort of understanding the system requirements.

Predecessor System

The cases where production-quality software is created without any type of predecessor software are few. History has shown that even the best planning does not get it right the first time. This idea has been first captured by Brooks (1975): *Plan to throw one away; you will, anyhow.*

Although throwing away software might sound like wasting money and resources, it is actually the opposite. Developers, customers and users are not sure what they want when software is created for the first time. The customers do not know or cannot articulate what the actual problem is that the software tries to solve. The users are not sure what features are needed and how the system should behave. Developers do not know how they need to design and implement the software. Thus, a predecessor system offers a possibility for the different stakeholders (developers, users, customers, etc) to reflect their ideas about the software and forms the basis for the discussion of how the new system should work.

If you are planning to create a new software product it is highly recommended that you create a throw-away prototype first. This lets the software developers study the technology and design alternatives and gives the other stakeholders a possibility to give feedback. If you are replacing a legacy software product with a new one, the amount of prototyping needed depends on how closely the predecessor and successor systems are related. It is possible that you have learned so much from the legacy software product that you do not need to build a prototype at all. In a way the legacy product has been the prototype for the new software. However, be aware of the issues that concern rewriting legacy software. These issues are studied in more detail in Section 4.4.

In some cases software product companies do not own the rights for their predecessor system. In fact, small software product companies are often forced to do customer projects to maintain a revenue stream. Sometimes the first version of the product has been created in a project where the customer has received full rights for the software. After or during the project the company has realised that there could be a wider demand for such a product. Therefore, the company re-creates the product from scratch to receive rights to the product. In doing so they also use the knowledge gained from the customer project to create a better version of the software. The rights of the predecessor system do not necessarily have to be owned by the company, because you can retain the lessons learned from the customer project and use them in future development activities.

Another type of predecessor system a company might possess is legacy software without the original developers, customer, or users. This might happen, e.g., as a

result of a company acquisition. In such a case the benefits of having a predecessor system are reduced, because some or all of the people who have learned essential details about the software are no longer available.

This section introduced predecessor systems as an important building block when creating the architecture for a software product. We identified four types of predecessor systems: legacy software product, prototype software, software from customer project, and legacy software product without the original stakeholders with knowledge about the product. Except for the last type these are likely to be beneficial when designing the new software architecture.

All involved stakeholders can benefit from the lessons learned from predecessor systems.

The benefits of predecessor systems can also be gained from customer-specific projects.

Developer Competence

Developer competence affects your software product architecture. For example, if no one in your company knows J2EE it does not make sense to use J2EE to create a software product. However, you might get into a situation where you absolutely must use a technology that you do not have prior knowledge of. It is possible that the company's strategy is to get more customers by using a new technology. In such a case there are basically two alternatives:

1. Get some of your people to learn the new technology. However, this will lead to a case where productivity is low and the resulting product will likely have low quality, which means it can only act as a prototype (see previous section about predecessor systems).

2. Hire a new employee or a consultant to work with your employees on the software. If you hired a consultant, make sure that someone in your own organisation participates and learns how software is developed with the new technology. Otherwise, when the consultant leaves you end up in a situation, where you still have no one who knows the required technology, but you have to maintain a system developed with it.

On the other hand, people's prior knowledge can lead to an anti pattern called Golden Hammer (Brown et al. 1998): *"I have a hammer and everything else is a nail."* For example, consider a task where previous versions of data files need to be stored. The solutions can vary from a version control system and database to scripts copying backups to a separate directory. More information would be required to actually determine which of the solutions is best for the current problem. However, it is certain that if the developer has a lot of experience with either version control systems or databases, it is likely that he will use the tool he is familiar with rather than going for the simplest option of creating scripts that store the data to a separate directory.

Whenever possible, use a solution that utilises your developers' competences.

Don't let developers' competences restrict technology selection and future solution ideas.

Requirements

Requirements naturally form the basis for planning your product architecture. There are two issues about requirements that are discussed in this section. First, you must prioritise your requirements. We have seen cases, where all requirements are prioritised as the most important (especially the CEO's seem to have the habit of doing this). Such prioritisation, however, offers no guidance to product design (or for anything else). There is only one response to people who say that all the requirements are the most important: **Grow up, you cannot have everything!** You do not need fancy mathematical functions based on various characteristics and metrics to prioritise requirements. A simple prioritised list will do just fine. An example of this is using backlogs, which was shown in Section 1.2.

Second, you should distinguish between functional and non-functional requirements. Non-functional requirements are important because they are difficult to add to the system later in its life cycle. For example, consider adding security to a system afterwards. Also, non-functional requirements demand continuous effort and awareness from developers. For example, there can be no single task such as "make the system flexible". The non-functional requirements should be made measurable. For example, the system architecture for a web application on the Internet with thousands of concurrent users and a response time requirement of less than 3 seconds (not to mention security issues, etc.) is different from a web application on a company intranet with 5 concurrent users.

Functional requirements traditionally form the basis for the software architecture. Non-functional requirements like maintainability, effectiveness, and security can be taken into account by using Bosch's (2000) methodology.

Prioritise your requirements.

Define and communicate also your non-functional requirements.

Design Principles

This section highlights some of the important design principles that should be applied when developing software. These principles can be applied to procedural programming as well as object-oriented (OO) programming[2]. However, using an OO language makes applying the principles a bit easier. These design principles are general enough to be applicable for any type of software. Violation of the design principles is probably okay, as long as you know there is a good reason for it.

[2] In fact, they have been developed long before the breakthrough of OO development

Low Coupling

The idea of low coupling comes from IBM researchers Stevens, Myers, and Constantine (1974). Low coupling means that software elements, e.g., routines, classes, modules, sub-systems, etc., should not be heavily connected to other software elements. The stronger the connection is from one software element to another, the more dependency there is between the elements. Some of the problems of software elements with high coupling are described below (Chidamber and Kemerer 1994; Larman 2002):

- Changing one software element will also force you to change the other elements that have couplings with the first element.
- A software element is difficult to understand when it relies heavily on other elements.
- Reusing the software element is more difficult.
- More rigorous testing is needed for a software element that is heavily coupled.

"Too much" coupling between software elements is harmful. You can measure the amount of coupling the objects in your system have and concentrate on the objects with the highest coupling. But how much coupling is actually too much? This is context dependent, so it is difficult to give exact numbers. To make things more difficult, there is no single coupling metric. For example, the following metrics can be used to measure coupling: methods share data, method call from one method to another, and class is passed as parameter. Look in (Briand, Daly, and Wüst 1999) if you want to study this issue in more detail.

You might want to study the qualitative elements of coupling instead of the number of couplings. The following example illustrates the importance of qualitative elements.

> A high number of method-to-object couplings (MOC) can suggest that a method has poor design. However, the MOC metric does not say anything about the quality of the couplings. If a method's MOC metric is high because it has many couplings to classes that are relatively stable and of a utility nature (Java API), it is not as bad as if the couplings are to other functional classes that are developed simultaneously with the method itself.

Qualitative elements of coupling that you may wish to study can be, e.g., are there couplings that should not be there, and is decreasing couplings possible and would it increase modularity.

For routines there is the Law of Demeter (Lieberherr and Holland 1989), which can be helpful when studying method-level couplings. Basically the law states that a routine should only

- call methods within its class (*you can only play with yourself*),
- call methods with directly held objects (*you can play with your own toys, but you can't take them apart*),
- call methods with any parameters that were passed into the method (*you can play with toys that were given to you*), or
- call methods of classes it created (*you can play with the toys you made yourself*).

High Cohesion

Stevens, Myers, and Constantine (1974) presented the idea of high cohesion simultaneously with low coupling. Cohesion means how closely related the functionalities or

responsibilities in the software elements are. A class that performs several unrelated operations or tries to do too much has low cohesion. Some problems of low cohesion are listed below (Chidamber and Kemerer 1994; Larman 2002):

- Low cohesion increases complexity therefore making the software hard to understand and maintain.
- Elements with low cohesion are difficult to reuse since they perform several tasks.
- Low cohesion means lack of encapsulation.
- Low cohesion elements are constantly affected by change.

Low cohesion for software elements is undesirable, but as with coupling no exact number can determine when the cohesion is too low. Also, several cohesion-related metrics are available (Briand, Daly, and Wüst 1997).

Indication of a class suffering from low cohesion can be found from the number of 'and' words in the answer to the question "What does this class do?" The same rule can be applied to method-level cohesion with the distinction that in the method level the number of 'and' words should always be very close to zero.

Information Hiding

Information hiding is another classic design principle (Parnas 1972) that is still very valid. Information hiding means that module details that are likely to change should be hidden from the outside world. This way the outside world will not be affected at all when the internal structure of the module is changed. For example, if the module's data structure is changed from stack to linked list, the other modules will not be affected by the changes if the information hiding principle has been properly applied. Of course, no one can perfectly predict the areas that are likely to change. The book Code Complete (McConnell 1993) offers the following list of areas:

- Hardware dependencies
- Input/output
- Non-standard language features
- Difficult design and implementation areas
- State variables
- Data-size constraints
- Business rules
- Complex data

Information hiding is closely related to two other important concepts; encapsulation and abstraction. Although some sources claim that information hiding is a synonym for encapsulation, this is not the case. For the meaning of abstraction, information hiding and encapsulation we quote (Berard 2000):

One could argue that abstraction is a technique that helps us identify which specific information should be visible, and which information should be hidden. Encapsulation is then the technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible.

A good source for further studies on object-oriented design principles is (Larman 2002).

Design Methods

This section shortly illustrates some of the important design methods. The list of design methods described here is by no means exhaustive. The methods listed here range from low-level module design to high-level architectural design, which means that several methods can be combined. The purpose of this section is to briefly describe the different methods in order to give you ideas of how software design can be done. To get a deeper insight to any of the listed methods you should study the references. Software design can be combined with software quality assurance. Two of the methods presented in this section do this, namely Test-driven development and Design by contract.

Test-Driven Development

Test-driven development (TDD) is strongly emphasised in the Extreme Programming software development methodology (Bianchi et al. 2003). The key idea of TDD is to write the tests before the code is written. In TDD writing tests does not mean describing test cases in natural language. Rather it means writing test code that will test the production code. Usually this is done with the help of unit test frameworks, such as JUnit, which operates in Java (similar frameworks are also available in other languages, like CUnit for C and DUnit for Delphi).

Writing a test becomes a design activity when it is done before the actual code is written. For example, in TDD a software developer must first write the test cases for the class he is about to implement. When the developer writes these tests he has to specify his intentions for the functionality of that class. In doing so he is forced to think through the class behaviour at a very detailed level. In this way, as the developer writes tests for the class, he also designs its behaviour. The written tests act as a design "specification", but you also get the benefit of having the tests in their traditional role as quality assurance tools. For more examples of TDD, please look up (Beck 2002).

Design by Contract

Design by contract (DBC) has the same core idea as test-driven development. In DBC each method and class has a contract that defines its behaviour. These contracts are specified with assertions that tell what condition must hold in each situation. A method has two conditions, a pre condition and a post condition. Pre conditions tell the conditions that a caller must satisfy in order for the call to be correct. Post conditions express the conditions that must hold after the method call. The following code example (adopted from the eiffel.com web page) shows the method-level contract in practice. In the example an element is inserted into a data structure so that it is retrievable with a key. The pre conditions require that there is space in the data structure and that the given key does not contain an empty string. The post conditions demand that the data structure contains the inserted element, the element is retrievable with the key, and the count of the data structure is increased by one.

```
put (x: ELEMENT; key: STRING) is

     require
          count $<$= capacity
          not key.empty

     do
          ... Insertion algorithm ...

     ensure
          has (x)
          item (key) = x
          count = old count + 1
     end
```

At class level it is also possible to specify contracts that must hold for all class instances. Also, inheritance is supported through subcontracting. In DBC it is possible to specify during compile time, which contracts if any are evaluated when the program is executed. This way we can get rid of the performance hit caused by the assertions.

In DBC, as in TDD, the idea is to first specify the exact conditions for a class or a method. This specification is actually the detailed design for the code that is to be implemented. The major downside of DBC is that it has native support only in the Eiffel language. For other programming languages (e.g. C++ or Java) an extension plug-in is needed. For more information on design by contract, the reader is referred to (Meyer 1992).

Prototyping

The practice of prototyping has been successfully used in many engineering disciplines. Prototyping has also proved to be a critical success factor in software development (MacCormack et al. 2003). Prototyping can be classified as a software design method, since it helps in envisioning the final product in several ways:

1. Prototypes can be used to study new technology. A typical question would be: "Is it possible to create X by using Y in the environment Z?" The goal of a prototype might be: “To study if it is possible to create an interface to the company’s mainframe from the company’s employees’ digital television boxes.”
2. Another type of prototype focuses on finding the best architecture. You might want to explore which one of two architectures is best (according to some specified metrics) when using Java as programming language.
3. Some prototypes focus on discovering the requirements. In such cases the ideas behind the product are so vague that a prototype is needed so that different stakeholder can see the product in practice and give feedback for further elaboration of the ideas.
4. In some cases user interface issues need to be studied with a prototype. For example, you might have a clear idea of a completely new product concept and you know what features the product should contain, but you want to explore different options for a user interface to maximise usability.

Design Patterns

Design patterns in particular represent reusable designs. Gamma et al. (1994) have adopted the definition from (Alexander et al. 1977) that describes patterns in tradi-

tional architecture: "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*" In other words, software design patterns represent something that experienced and talented developers have seen as a good solution to some design problem.

Design patterns can make software design much easier, since they represent already tried and tested solutions. Therefore, no time is wasted on rediscovering the wheel. The downside of design patterns is that you must know when to apply them. Implementing design patterns into software, when a simpler solution would suffice, results in wasted effort. Unnecessarily implemented patterns also make the software more complex and more difficult to comprehend.

Since developers can abuse patterns, you should be aware of their possibilities, but not use them in places where they do more harm than good. More experienced developers will likely have a better touch on when patterns should be applied and when not. Therefore the decision to use design patterns in a particular part of the software should be made by senior developers or software architects.

For further reference we recommend (Gamma et al. 1994), which contains patterns that consist of few interacting classes. (Buschmann et al. 1996) also discusses design patterns, but in addition it presents architectural patterns (like pipes and filters and blackboard) that can be used in software design. (Larman 2002) also discusses some patterns, although its main focus is software modelling.

Modelling

There are probably hundreds of books of software modelling which all promise better software, timely delivery, fewer bugs, and so on. The Unified modelling language (UML) is currently the dominant software modelling language. With UML you can illustrate different views of the software, its requirements, and the domain. UML itself has several types of diagrams: use-case, component, deployment, activity, sequence, collaboration, class, and state. UML does not currently cover database modelling, which will increase the number diagrams even more, if you decide to model also your database.

Unfortunately none of these diagrams or software models compile into executable code that you can directly sell to make revenue. Therefore we feel that it is unlikely that a small software product company would need all models and diagrams *in full detail*. However, we do not claim that the models are useless either. For example, when creating a network protocol, a state diagram for the client and the server can greatly simplify the implementation and reduce the amount of bugs. Also, when the software developers do not possess domain understanding, it is important to create a domain model, which decomposes the domain into individual concepts and thereby makes software design easier. Thus, you should be aware of the possibilities provided by different models. This way you can apply the ones that are needed in your current situation.

If you like to learn more about software modelling, we recommend (Larman 2002). Good pointers for "light-weight" modelling are given in (Ambler 2002).

Architectural Documentation and Design

Architectural documentation of a software product is important. Some even acclaim that if you do not have documentation of your architecture then you really have no architecture at all. This is because architecture is about communication. With architectural documentation you communicate the architecture to different stakeholders. For example, developers might implement features in a way that decay or violate the architecture, since they are not aware of it, or they might not utilise the reuse possibilities, but do unnecessary work instead.

Several methods exist for defining high-level software architecture. We will not cover any of them here since these methods are often quite laborious and probably not suitable *as such* for small companies' product development. However, you can use the different methods to get ideas for architectural design. For a good overview of the different methods (Quality Attribute Workshop, Attribute-Driven Design, Attribute Trade-off Analysis Method, Active Reviews for Intermediate Designs, and Cost-Benefit Analysis Method) we recommend (Barbacci 2003).

4.3 Software Implementation

There is more to implementation than just writing code. Especially when more than one person is involved, the efforts need to be coordinated. There are also many implementation practices that can be used to make coding more efficient. This section contains two areas of discussion that cover the issues above; organising and managing development, and implementation practices.

Organising and Managing Implementation

Regardless of the processes used, there are common issues to consider for organising and managing implementation. To coordinate the efforts of many individuals, some common rules and guidelines should be applied. These can be gathered into a developer guide. The physical setting needs to be considered, especially the office layout. In order to get good results, you also need good and motivated people, so leading development is an important area. These issues are discussed further in the following sections.

Developer Guide

This section illustrates the type of information you need to spread to your developers. The information can be gathered into a developer guide consisting of one or several documents. Note that the document does not have to contain many pages of text. A few figures and checklists can often be the best way to communicate the relevant information.

Your developers should have the following:

- Description of product architecture
- Coding conventions
- Harmonised development tools
- Standard procedure for using version control
- Build system that is easy to use
- Test environment that is easily restored and extended
- Knowledge of how each work product should be tested and documented

The description of product architecture could be done by answering at least the following questions. What is the structure of your product? What is the product architecture like? What are the main modules and what are their functions and responsibilities? Who are the best people to answer the questions concerning module X? What external components are utilised? Is there a standard way of adding certain functionality to the system? For example, every time you add a view for the end user you must perform certain standard steps.

Coding conventions strive to assure that all code is written in a standard way. The quality attributes each company emphasises can vary from maintainability to security and efficiency. Currently the main focus on coding conventions has been on understandability of the source code. Coding conventions should say how the code is formatted (use of braces, indenting, etc.), what the naming convention is (often it is best to use self-documenting names rather than trying to use as short names as possible). Coding conventions can guide how to design the code, e.g., all methods should have less than 150 lines of code. Coding conventions can also contain more general instructions, like the ones found in (Ambler and Constantine 2000), e.g., a method should be understandable in less than 30 seconds.

Developers need information about the used development environment. It is a good idea to harmonise the development tools. For example, when all developers have the same IDE it is easier to adjust the IDE settings to format all developers' code in a similar fashion. Using the same tools improves the collaboration between developers, e.g., when all are using the same IDE a senior developer can easily come over to a junior developer's computer and help in debugging.

Besides similar tools, you should have a procedure on how to use the version control system. Developers should know how and when to commit code. They should also know how to use different tags in the version control system. For more information, see Chapter 6.

From the developers' perspective, making builds should be as easy as possible, since the developers need to build the product continuously when developing and debugging code. Preferably, the build system should work in a similar fashion for all the modules and there should be a way to build the entire system from scratch with a single command.

There should be a test environment also for the software developers. In some cases only a test database is needed if the developers can run and debug the rest of the application from their own machines. An important issue concerning the test environment is that it should be easily restorable to its original condition, e.g., to its initial state with predefined tables and data. The test environment should also be extendable, e.g., developers can write scripts to fill the database with problematic data.

The final two issues that should be made clear for the software developers is how the work products are documented and how each work product should be tested before the work is considered as complete.

Physical Setting

The office layout can affect development performance. A common dilemma is whether to have an open-plan office or individual rooms. A typically utilised solution is a combination of these extremes; bigger rooms with 2 to 6 persons in each. Dedicated rooms give developers more privacy and therefore a better chance to focus on the work at hand. On the other hand, rooms with more people facilitate communication, which can increase the productivity. There is no easy answer to this issue. For more information and ideas, please refer to (Cockburn 2002) and (DeMarco and Lister 1999).

Leading Development

One important issue in leading development is that you encourage good coding (McConnell 1993). One technique to achieving this is to have more than one developer work on each part of the software. This ensures a sanity check to the code by programming peers, which prevents sloppy code. It also lowers your dependency on individuals, since there are always at least two persons that know different parts of the software.

You might consider picking examples of good code and rewarding the developer that has done the exemplary work. However, beware that technical expertise is required to do this. If you lack technical knowledge, you can let the developers choose the exemplary pieces of code. You can also apply this idea to other areas, e.g., writing good unit tests, creating good documentation and so on.

Another good idea is to require developers to sign for their work. In other engineering disciplines this has resulted in increased quality of work. When people know that a certain work product will have their name on it, it is likely that they create something they want to be proud of.

The objectives for each task also have a great effect on the outcome. Studies have shown that the primary objectives in programming work are usually achieved with the cost of other objectives (Weinberg and Schulman 1974). You should consider what is important in each programming task, e.g., what are the goals when implementing module X: are security issues critical, does the module need to extended in the near future, is high performance required, does the module need to be reused in a future programming task, and is the module needed quickly to satisfy an angry customer. It is unlikely that you would be able to achieve all the goals. Therefore you need to specify what the 2 or 3 most important goals are for each task. If you only require fast programming, then you will likely end up with a poorly maintainable system with security-critical errors that is also inefficient.

Other important issues in leading development are motivation, team building, and getting the best people. For motivation and team building we strongly recommend reading chapter 11 in (McConnell 1996), which describes the motivational factors for programmers and team building. Also, Humphrey (2003) has some viewpoints to this issue.

Implementation Practices

A myriad of implementation practices exist, but in this section we concentrate on presenting two interesting and beneficial methods that might be new to you, pair programming and refactoring.

Pair Programming

Pair programming is a technique where two programmers design, code and test software collaboratively at one computer (Williams and Kessler 2002). One programmer writes code and the other thinks more strategically whether they are heading in the right direction or not, at the same time effectively reviewing the code. Both communicate continuously with each other to understand what is happening. The roles can be changed at any time during a pair programming session. Within the project team the pairs should be rotated in order to allow better transfer of knowledge between team members.

Lately pair programming has received a lot of publicity being one of the core practices of Extreme Programming (XP). However, pair programming has been known for a long time and is by no means limited to be used with XP only. Also, the requirement in XP, that pair programming should be used for all programming is quite extreme, and not necessarily the optimal way to do pair programming.

Several benefits on using pair programming have been reported based on tentative studies. It seems that pairs produce code with fewer defects, in shorter elapsed time and more enjoyably without using significantly more effort (Williams and Kessler 2000; Nosek 1998). Considering the additional proposed benefits for teamwork, knowledge transfer and learning (Cockburn and Williams 2000), pair programming may be a very advantageous technique for a software development organisation.

Some situations have been found especially well suited for pair programming. When doing very complex tasks, a pair may be able to solve harder problems than either individual alone. Because pair programming also means quite thorough, immediate code reviewing, it is a good practice when making changes that require special care, e.g., just before releases. The training aspect of pair programming makes it a very efficient practice for getting new developers productive, if the more experienced partner is willing to act in a mentoring role in the beginning.

The advantages of higher productivity and a lower number of defects are intuitive. More effective learning of development tools and domain also leads to higher productivity of the development team, as does the enjoyment of work. The improved transfer of knowledge of the developed system is a way to avoid the negative effects of personnel changes in the development team (losing the only person who knows a module, getting a new person to a level of being productive). If the increase in the development effort when pair programming is low, the realisation of just some of the proposed advantages would make the overall effect of pair programming positive from the organisation's viewpoint. However, sometimes the effort has been reported to double, which may be too costly for pair programming to be practical.

Refactoring

Refactoring is improving the structure of existing code without changing its observable behaviour (Fowler 2000). Thus, refactoring makes the code easier to understand and extend, bugs can be found more easily, and the software design is improved. Extreme programming has chosen refactoring as one of its practices to compensate for

the lack of up-front design that is missing from the methodology. According to the extreme programming philosophy it makes no sense to invest into up-front software design, since refactoring can be applied to change the structure of the software.

Currently there are tools that can do some simple automatic refactoring. For example, in Eclipse it is possible to do extract-method refactoring where you select a piece of code inside a method and extract it to create a new method. The tool automatically determines whether performing such refactoring is possible and what the parameters and return values should be. Finally, the tool replaces the part of code that was extracted with a method call to the newly created method.

Regardless of whether you believe in the extreme programming philosophy or not, you should be aware of the benefits of refactoring. Refactoring is especially important in the software product business, where software evolution can cause code decay. Software evolution is discussed in more detail in the following section.

4.4 Software Evolution

During a software product's life cycle the software is not created only once. For a successful product, several generations of the software are developed in succession, which may all address the same fundamental problem while the internals of the product change over time, e.g., through the use of new technology.

There are some fundamentals of software evolution that were discovered in the early dawn of software engineering in the 1960's and 1970's. The Mythical Man-Month (Brooks 1999) states that when it comes to software even the best planning does not get it right the first time: "*Plan to throw one away; you will, anyhow*." On the other hand, the laws of software evolution (Lehman 1980) state the following:

Software, which is used in a real-world environment, must change or become less and less useful in that environment.

As an evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon.

The laws of evolution state that the need to re-engineer the software structure is continuous. You will likely need to rewrite a great deal of the software during its life cycle. Emerging new technologies and changing requirements are also likely to increase the need for rewriting and reengineering.

The rest of this section introduces the technology roadmap as a way to plan your technological future. We also discuss the concept of technical debt, which can be used to manage effects of software evolution. Finally, we bring up some issues to consider when planning major software reengineering or rewriting.

Technology Roadmap

A technology roadmap is needed to manage the technological future, plans and options of your product(s). It should include big technology decisions, such as whether to go with J2EE or .NET. The roadmap should also include at least the following: outsourced software parts (e.g., code libraries, databases, and XML parser), employed technologies and your own software architecture. For more on roadmapping, see Section 2.4.

Your technology roadmap should discuss and document the following:

- Technology selection — new possibilities
- Outsourcing — make or buy
- Software architecture

Technical Debt

The idea of technical debt was first introduced by Cunningham (1992). Tight schedules can force the development team to take technical debt in order to reach the primary goals of the development increment or project. Technical debt increases when software developers choose quick and dirty solutions instead of proper implementation. Other things that might be considered as technical debt are the failure to upgrade the software to be built with the latest compiler, poor software documentation, and so on. As technical debt consists of things that have not been done properly, it slows down your future development activities (this is called technical interest). For example, a document describing the software design helps the developers to implement features more efficiently and according to the design. Quick hacks or patches that are often used under deadline pressure can break the design, making the software less maintainable, understandable, flexible, testable, and therefore more expensive to develop.

Increasing the technical debt makes perfect sense if you are doing it to reach a critical deadline. However, you must also remember to decrease the debt. Decreasing debt should preferably be done at the beginning of the next release project or increment (for increment themes see Section 4.5).

The technical roadmap contains plans, possibilities and goals for future development of the product. The technical debt contains issues that decrease software quality and should be fixed or improved in the product to make future improvement easier or even possible.

Technical debt is development tasks that already should have been done, e.g., refactoring to improve code design, redesign, upgrading to the next java compiler version, and documentation of the software.

Interest for technical debt is paid in future development activities, which are slowed down by the debt.

Rewriting and Reengineering Issues

Sooner or later you start wondering whether it would be easier to rewrite the whole legacy software from scratch instead of trying to continuously cope with its complexity. Another way of rewriting software is reengineering, which uses the existing code as a basis and reimplements it with new design and structure. Both of these approaches require considerable effort and therefore you should carefully consider whether to allocate resources to it or not. Unfortunately, there is no easy answer to when major rewriting or reengineering efforts should be started. Ideally, in incremental development the way to prevent major rewriting efforts is to do it a little bit at a time, e.g., by refactoring, but that is not always enough in the long run.

One of the key factors is that the organisation has learned how to make the product better from previous versions of the product. In the software product companies we have seen, most products are at least second-generation products, i.e., an earlier generation of the product has existed that has been completely replaced by the new generation product. Such predecessor systems can be though of as prototypes for the current system. Like prototypes, these systems have provided valuable information for the organisations and made development easier. This kind of generational knowledge has also been identified as an enabler of flexible development processes (MacCormack, Verganti, and Iansiti 2001).

Technology changes (language, platform, etc.) are a source affecting the need to rewrite software. Naturally, moving to a new technology must be motivated by business reasons, i.e., if you can attract more customers and make more money using another technology, you should carefully compare the cost of rewriting the software to the benefits gained.

Big system changes can entail major reengineering or rewriting effort. For example, consider moving to an online system from a system that previously has been accessed from a console only. An online system that operates on a single computer can become a bottleneck. Therefore, you might want to create a distributed system. Such decisions can require big changes to the software structure and therefore might require major reengineering and rewriting effort.

Quality attributes that hinder software development — such as poor extendibility, changeability, and maintainability — can also cause a need for system rewriting. In such cases the software has grown out of its original design. Typically prototype systems without proper design or systems that have been developed patch after patch suffer from this problem. These kinds of systems eventually result in unhappy and unmotivated developers and slow down implementation of new requirements.

Improving other system qualities like reliability, security, and performance can also require major rewriting. In some cases the only way to get the system reliable is to rewrite it from scratch. The same argument can also hold for security and to some degree for performance.

Software size is one of the most important factors you should consider when planning major rewriting or reengineering. You should avoid a complete rewrite if the application is large (Chan, Chung, and Ho 1996). This idea is based on the risks involved in rewriting large applications. Instead, you should consider a partial rewrite and preferably make it incrementally. Properly timed reengineering and rewriting helped Microsoft to win the browser war (Cusumano and Yoffie 1998). Microsoft redesigned Internet Explorer in version 3.0 while the code base was still small. Netscape tried to do the same with Communicator 4.0 and 5.0, but failed, as the code base was considerably larger. An idea of the bad shape the code was in is given by Netscape's release manager's comments on the client software: "*if part of our interview process were to show people what our code looks like, people wouldn't come to work for us*." This example illustrates the importance of considering size in software reengineering. Support for failures in large rewriting initiatives is also found on the *Joel on Software* web site (Spolsky 2000) that lists several failures while doing complete rewrites (Netscape, Ashton-Tate, Lotus 123, Apple's MacOS, Borland's Arago, Quattro Pro, and even MS Word).

The so called *black box effect* is another issue that should be considered. With the black box effect we mean that the development organisation has so poor knowledge of the software that it starts to view the software as a black box. The black box effect is often the result of unavailability of the original developers that understand

the design and technology. The existing developers see the software as a mysterious and scary legacy system that cannot be touched. The developers also start to perceive the technology as outdated. Therefore, the developers have poor motivation to develop the system further. This in turn results in poor maintainability. In a way, the belief of the scary legacy system becomes a self-fulfilling prophecy. To prevent this problem from happening, you should make sure that more than one developer knows each part of the software, e.g., by practicing pair programming. Also make sure that the developers that are new to the system really understand the system before the old developers leave. Gaining deep understanding of a complex software system can take more than a year. There is really no two-week course to educate your new developers to become masters of a large software product.

The following factors affect the decision to rewrite software:

- Knowledge gained from previous product generations
- Technology and system changes
- Poor internal software quality, e.g., poor extendibility and maintainability
- Lack of internal qualities, e.g., reliability or security can only be achieved by reimplementing the product
- Size of the existing system to be replaced
- The black box effect — the developers' lack of knowledge of the existing system

4.5 Pacing in Software Design and Implementation

In time paced development, the most noticeable difference to your previous way of working is the explicit rhythm and the absoluteness of the schedule through that rhythm. Instead of finishing a task and being a week late because of it, you have to accept and learn to drop tasks, to get the product ready by the end of the increments and the release project. This puts requirements to the way you design and implement your product, which is discussed in this section. First we discuss the time horizons and control points of the CoC from the design and implementation viewpoint.

The CoC from the Design and Implementation Viewpoint

As you can recall from Chapter 1, there are four time horizons in the basic configuration of the CoC: strategic release management, release project, increment, and heartbeat. Strategic release management plans the product roadmap and provides a guiding product vision for the developers and designers. The input of the Chief Technical Officer (CTO) and the Chief Architect is important in deciding the technological aspects of the product vision and planning the technology roadmap (see Section 4.4) of the product.

A release project can last from 1 to 6 months, depending, e.g., on the release type (for more information on release types, see Chapter 2). In the release planning control point Business (i.e. the stakeholders that are responsible for the commercial aspects of the software product, such as sales & marketing) share their vision and goals for the product release with the development team. The development team is responsible for

making preliminary effort estimations and assessing the technology and design risks, including quality assurance considerations (see Chapter 5). As part of release project planning, we recommend planning themes for the increments. This is discussed in detail in the following section. The release project ends with a release demo of the release candidate, which the development team is responsible for.

Increments can last between 1 to 4 weeks, depending, e.g., on the product and its release type. In the increment planning meeting Business sets the goals for the increment and suggests what should be implemented. The development team then considers the goals and the required functionality, and plans what tasks are needed to implement the required functionality, not forgetting quality assurance (see Chapter 5). Designing the product so that tasks can be planned for short increments is demanding, which is discussed more below. Work effort estimations are then made for the tasks by the development team. The tasks and their effort estimations are then given to Business, whose responsibility is to consider the availability of resources (see Chapter 3), prioritise the tasks, and possibly redefine the increment goals to reflect the prioritisation, so that rescoping during the increment cycle is possible for the development team based on these prioritisations. This means that not all tasks can be of critical importance! In the end control point of the increment, the development team demonstrates its achievements by showing the working software to all interested. This gives, e.g., the salesmen a good chance to see what has been done, how it works, and ask questions. The demonstration also shows the progress of the release project and is the basis for the planning of the following increment.

The heartbeat rhythm is the tightest rhythm on which progress is tracked and managed. We recommend a daily heartbeat rhythm, but at least the heartbeats should not exceed 1 week. The short heartbeat meeting is the control point of the heartbeat. In this meeting the developers (and testers) discuss three issues: what each person has done since the last meeting, what problems each person has encountered, and what each person is going to do until the next heartbeat. The meeting should not exceed 15 minutes, so when problems are identified, the solutions are not discussed at the meeting, only who will help solving the problems. The heartbeat meetings are a good way of following development (and testing) progress. Another way of monitoring progress is using a burn down graph. The burn down graph has two dimensions: calendar time on the X-axis and the cumulative estimated effort left to complete the open tasks on the Y-axis. In an ideal increment the burn down graph will look like a straight line from the upper left corner to the down right corner of the graph, but this rarely happens, as is illustrated in the example graph in Figure 4.1. On Wednesday 7.4. the development team decided to adjust the scope of the increment by removing tasks, which is evident from the steep slope of the graph.

Increment Themes

We recommend that you specify themes for increments when planning a release project. Specifying themes help you identify suitable goals and implementation activities for each increment. At the same time you should consider quality assurance tasks and themes, which might also influence your goals and choices of implementation activities. Your project will likely consist of 3-6 increments. In our example in Figure 4.2 the project consists of 4 increments the themes of which we have labelled *Early*, *Middle*, and *Late*. We have also identified continuous activities that are not specific to any increment. The following sections discuss the increment themes and their content, including the continuous activities.

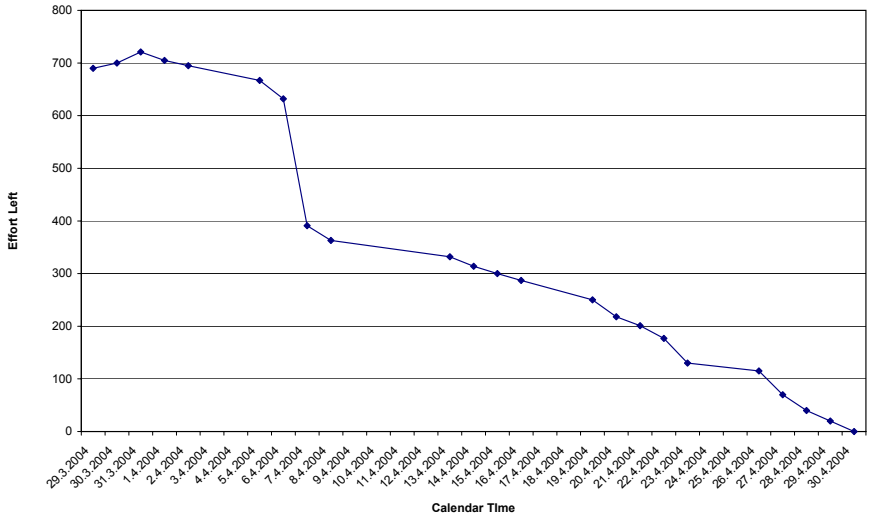


Figure 4.1: Example burn down graph

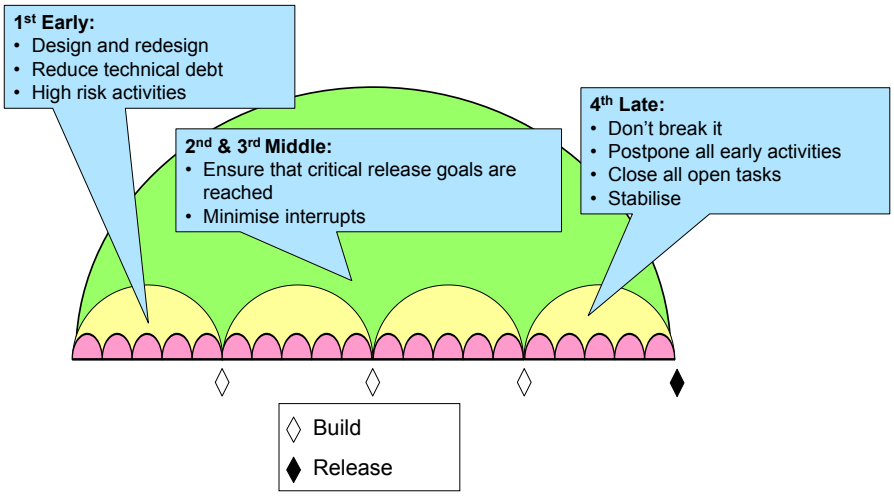


Figure 4.2: Themes for increments and their contents

Early

In the early increment(s) you want to start working on high-risk tasks that can compromise the release goals. From a design and implementation perspective high-risk tasks are often implementation of new features that have to be checked for implementation feasibility or features that are otherwise considered difficult or laborious. High-risk tasks can also include tasks that need involvement of people outside of the development team. For example, in order to improve the usability of the product, the development team might want to ask for input from the actual users. The timing of the input cannot be fully controlled by the development team, so such tasks possess a high risk. Also, the input provided by the users might require such big changes to the product that they cannot be completed, if they are not started early in the release project. Outsourcing is another issue that can be considered risky. Naturally, this depends on the confidence you have in your supplier. Still, outsourcing involves a third party that cannot be controlled by the development team. Therefore, starting to work with a supplier as early as possible reduces the risk that a delay at the supplier end compromises the release goals.

The second type of tasks that you want to focus on in early increments is tasks that get easily neglected or postponed. These tasks are important, but when the release date is closing in, it makes more sense to delay these activities in order to reach the primary goals. Therefore, the tasks should be performed in the early increments, when there is no immediate release deadline approaching. An example of such a task is software refactoring, which keeps the internal structure of the software clean and makes future development activities easier. Microsoft has a rule called “20% tax for rewriting code”, which means that 1/5 of the development effort should be spent on improving the software structure (Cusumano and Selby 1995). Other tasks that are important but might get postponed in late increments are code commenting and software design documentation. For example, undocumented software design can cause problems, because developers do not understand the product that they are developing. This makes them less efficient. It will also cause them to implement features in a way that is not in harmony with the product architecture, which again will make the software less maintainable and extendable.

The third type of activities that should be done in the early increment is design and redesign of the product. You need to figure out how the new features needed to fulfil the release goals will be implemented in the software. This may cause a need to redesign the product architecture, which could also be classified as a risky task. Both the tasks that get postponed in late increments and the need to design and redesign the product architecture can be considered as technical debt, a concept introduced in Section 4.4.

As an example of an early increment’s activities, you might start with updating the software architecture documentation. After updating the documentation you start to design the implementation of the new features, realising that you need to restructure the software architecture a little bit to make the implementation of the new features easier. However, before you can implement the changes to the architecture you need to do some refactoring to reduce the couplings between the classes you are trying to separate. Or you can start with refactoring some poorly structured code and then realise that certain functionality could be collected to a common utility library package, which could then be used for implementing the new features, and so on.

Middle

In the middle increments you should make sure that you are on schedule with implementing the critical features of the project. Preferably you should finish the implementation of these features before the late increment(s). Consider limiting the features that need external involvement, since they increase the risk of interruptions in your implementation pace.

Late

In the late increment(s) you should make sure that you do not break the system and risk the release date, which would break your release rhythm and complicate your life. Therefore, any activity that unnecessarily increases the risk of deteriorating the external quality of the software should be postponed to the next release project's early increment. You certainly do not want to implement risky architecture reconstructions in the late increment. When you plan the last increment you should make sure that it is possible to close all the tasks selected for that increment. This way there are no features that are left open, when the release project is complete. You should also realise that when debugging it is possible to spend days trying to locate a single bug. Therefore it might be a good idea to have a total feature freeze in the last increment and just concentrate on finding and fixing bugs to stabilise the product. It is also possible that in the last increment some development personnel is needed to help sales and marketing in producing white papers and other technical material, which will reduce the time spent on the actual software development. These issues should be considered in pipeline management.

Continuous Activities

We suggest that when developing software you continuously identify areas for improvement. Often it is not possible to immediately fix the discrepancies as you spot them. Therefore, you should at least make a note of them so that they are not forgotten, e.g., as tags in the code or improvement suggestion entries in your requirements management system. In the early increments you should allocate time to fix discrepancies that possess the most serious risk to the project, the product, and the organisation. These risks can mean problems in implementing necessary functionality in the future. These risks can also mean quality problems where fixing one bug will cause two new bugs to appear.

Design and Redesign in Paced Development

Pacing should facilitate good software design, because in order to develop software in increments you have to design the software in parts that can be implemented in each increment. Since the length of an increment should not exceed one month, this should effectively force you to make the development tasks relatively small, promoting a modular system design. As each increment ends with a demonstration of the increment's achievements, you should not be able to over design the system with too massive or detailed designs. This helps you avoid the anti pattern called Analysis Paralysis.

Ideally each developer implements one task at a time. This way a minimum amount of tasks are simultaneously under work and the rest of the tasks are either

completed or not started. This in turn makes increment progress tracking quite transparent. Unfortunately, this works only if the tasks are isolated from each other. In reality there are dependencies between the tasks that prevent completing one task before another one is completed. These dependencies can lead to a situation where there are several open tasks, because some tasks are blocked by others. It also forces the developers to start new tasks before the old ones are completed. These issues are dealt with in the heartbeat meetings. To reduce the problems coming from several dependencies between the tasks you can try to organise the tasks in a matrix where the task dependencies are marked. A dependency matrix will help determine the order in which tasks should be implemented. Instead of implementing the most critical tasks first you start with the tasks that are blocking the most critical ones. Creating a complete dependency matrix between tasks is virtually impossible, since new dependencies will emerge during the implementation. Still, it should help you in planning and designing the increment's contents.

Another design issue you must consider is the testability of the work products. You should design the system in such a way that makes testing the increment's work products easy. An important principle is that no tasks should be considered completed before they are tested. For a more detailed discussion of testing and quality assurance, see Chapter 5.

Redesign is an important part of paced iterative and incremental software development. The reasons for and issues of redesign were discussed in Section 4.4. Basically, there are two reasons for redesigning the product. The first one stems from the situation where the product's design is no longer in sync with the product requirements. The product was originally designed when all of the future requirements could not be anticipated. The second reason stems from developing the software in a hurry. Often, when developing a new feature you have neither the time nor the vision for planning how the particular feature should best be implemented. In such cases it is perfectly okay to implement the feature in a patch style. However, later you must take the time to redesign the product to get rid of the patches. Otherwise your product will become a complete mess that cannot be further developed and whose external quality will also start to suffer. A wise decision is to redesign and refractor continuously a little bit at a time rather than performing a big bang reengineering or rewriting that involves bigger risks.

Can all software be developed in an incremental fashion? This issue has been somewhat debated in the software engineering literature. To our knowledge it should be possible to develop all kinds of software, even including operating systems and compilers, incrementally. Basili and Turner (1975) report how a compiler was developed incrementally. First, the baseline architecture was created and then it was extended in the following increments. In an operating system the idea of a micro kernel, which emphasises small modules, should make the incremental approach feasible to use. However, with a pure micro kernel approach you might have too much overhead from the communication between the kernel modules. Even with a more monolithic type of kernel like Linux there is a modular structure that makes the incremental approach feasible. So we conclude that the claim *"this software cannot be developed iteratively and incrementally, because..."* is as false as the claim that "*this software cannot be tested, because..."*

A paced iterative and incremental process promotes good design.

A dependency matrix can help you reduce the number of open tasks.

The work products need to be testable.

You need to allocate time for redesigning the software.

All software can be developed in an incremental fashion.

References

Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angle. 1977. *A Pattern Language.* New York: Oxford University Press.

Ambler, S. W. and L. Constantine. 2000. *The Unified Process Construction Phase: Best Practices in Implementing the UP*. Lawrence: CMP Books.

Ambler, S. W. 2002. *Agile Modeling*. New York: John Wiley & Sons.

Barbacci, M. R. 2003. Integrating Analysis and Design Methods for the Software Life Cycle. In *News @ SEI* 6 (4). [electronic journal]. [cited April 13^{th}, 2004]. Available at `http://interactive.sei.cmu.edu/news@sei/columns/the_architect/architect.htm`

Basili, V. R. and A. J. Turner. 1975. Iterative Enhancement: A Practical Technique for Software Development. In *IEEE Transactions on Software Engineering* 1 (4): 390-96.

Beck, K. 2002. *Test-Driven Development by Example.* Boston: Addison-Wesley.

Berard, E. V. 2000. *Abstraction, Encapsulation, and Information Hiding*. itmWEB Media Corporation. [electronic article]. [cited April 13^{th}, 2004]. Available `http://www.itmweb.com/essay550.htm`.

Bianchi, A., D. Caivano, V. Marengo and G. Visaggio. 2003. Iterative reengineering of legacy systems. In *IEEE Transactions on Software Engineering* 29 (3): 225-41.

Bosch, J. 2000, *Design & Use of Software Architectures.* Harlow: Addison-Wesley.

Briand, L.C., J.W. Daly, and J. K. Wüst. 1997. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Proceedings of the Fourth International Software Metrics Symposium*: 43-53.

Briand, L.C., J. W. Daly and J. K. Wüst. 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. In *IEEE Transactions on Software Engineering* 25 (1): 91-121.

Brooks, F. P. Jr. 1999, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Reading: Addison Wesley Longman.

Brown, W. J., R. C. Malveau, H. W. McCormick and T. J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: Wiley.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Chichester: John Wiley & Sons.

Chan, T. Z., S. L. Chung, and T. H. Ho. 1996. An economic model to estimate software rewriting and replacement times. In *IEEE Transactions on Software Engineering* 22 (8): 580-98.

Chidamber, S. R., C. F. Kemerer. 1994. A Metric Suite for Object Oriented Design. In *IEEE Transactions on Software Engineering* 20 (6): 476-93.

Cockburn, A. 2002. *Agile Software Development.* Boston: Addison-Wesley.

Cockburn, A and L. Williams. 2000. The Costs and Benefits of Pair Programming. In *Proceedings of the XP2000 Conference*: 223-44.

Cunningham, W. 1992. The WyCash portfolio management system. In the *Addendum to the proceedings on Object-oriented programming systems, languages, and applications:* pp. 29-30.

Cusumano, M. A. and R. W. Selby. 1995. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People*. New York: The Free Press.

Cusumano, M.A. and D. B. Yoffie. 1998. Design Strategy. In *Competing on Internet Time*. New York: The Free Press.

DeMarco, T. and T. Lister. 1999. *Peopleware: Productive Projects and Teams.* 2nd ed. New York: Dorset House.

Fowler, M. 2000. *Refactoring: Improving the Design of Existing Code.* Canada: Addison-Wesley.

Refactoring is an important technique in preventing code decay in order to keep the software maintainable and extendable.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns — Elements of Reusable Object-Oriented Designs*. Reading: Addison Wesley.

A classic book about Design Patterns, which should be included in every professional software developer's library.

Hohmann, L. 2003. *Beyond Software Architecture.* Boston: Addison-Wesley.

Humphrey, W. S. 2003. Some Programming Principles: People. In *News @ SEI* 6 (4). [electronic journal]. [cited April 13^{th}, 2004] Available `http://interactive.sei.cmu.edu/news@sei/columns/watts_new/watts-new.htm`.

Jacobson, I., G. Booch, and J. Rumbaugh. 1998. *The Unified Software Development Process.* 2nd ed. Reading: Addison-Wesley.

Larman, C. 2002. *Applying UML and Patterns.* Upper Saddle River: Prentice Hall.

This book covers the topics UML, object-oriented analysis and design, and the unified process. In addition to explaining how to use different UML diagrams, it also describes when to use them in the different phases of the unified software development process. This book also presents some of the most important design patterns and principles. These patterns and principles, which offer tried and tested solutions, can help the developers to achieve high design quality.

Lehman, M. M. 1980. On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. In *Journal of Systems and Software.* 1: 213-21.

Lieberherr, K. J., I. M. Holland. 1989. Assuring good style for object-oriented programs. In *IEEE Software* 6 (5): 38-48.

MacCormack, A., C. F. Kemerer, M. Cusumano, and B. Crandall. 2003. Trade-offs between productivity and quality in selecting software development practices. In *IEEE Software* 20 (5): 78-85.

MacCormack, A., R. Verganti, and M. Iansiti. 2001. Developing products on "Internet time": The anatomy of a flexible development process. In *Engineering Management Review* 29 (2): 90-104.

McConnell, S. 1996. *Rapid Development.* Redmond: Microsoft Press.

An excellent book that deals with the managerial issues of software development, such as risk management, life cycle planning, estimation, scheduling, developer motivation, teamwork, and team structure. This book is in many ways a predecessor to the agile software development books and it contains many of the ideas that are also presented in agile software development methods.

McConnell, S. 1993. *Code Complete*. Redmond: Microsoft Press.

Although the book comes from the era of procedural programming, it still is an excellent book on software development. The book covers software design, data organisation, program control, and other important issues that every software developer and development manager should be aware of. The book views software development as similar to building construction. In my opinion this analogy to building construction is not very solid as there really are several differences, the key difference being the fact that software is abstract and buildings are concrete. Also, the idea of using PDL (program description language) before coding sounds like an unnecessary side step.

Meyer, B. 1992. Applying design by contract. In *Computer* 25 (10): 40-51.

Nosek, J. 1998. The case for Collaborative Programming. In *Communications of the ACM* 41 (3): 105-08.

Parnas, D., L. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. In *Communications of the ACM* 15 (12): 1053-58.

Spolsky, J. 2000. *Things You Should Never Do, Part I*. [electronic article]. [cited April

13^{th}, 2004]. Available at:
`http://www.joelonsoftware.com/articles/fog0000000069.html`.

Stevens, W., G. Myers, and L. Constantine. 1974. Structured Design. In *IBM Systems Journal* 13 (2): 115-39.

Weinberg, G. M. 1998. *The Psychology of Computer Programming.* Silver Anniversary Edition. New York: Dorset House Publishing.

Weinberg, G. M. and E. L. Schulman. 1974. Goals and Performance in Computer Programming. In *Human Factors* 16 (1): 70-77.

Since this article is hard to get the same issue is covered in chapter seven of (Weinberg 1998).

Williams, L and R. Kessler. 2002. *Pair Programming Illuminated*. Boston, USA: Addison Wesley.

The book explains what pair programming is, the reasoning behind the usefulness of pair programming, and several good practices for using pair programming.

Williams, L and R. Kessler. 2000. Strengthening the Case for Pair Programming. In *IEEE Software* 17 (4): 19-25.

Chapter 5

Quality Assurance

Juha Itkonen

5.1 Introduction

This chapter discusses how to arrange quality assurance (QA) in time-paced software development, with a focus on describing how quality assurance practices can be combined with a CoC-style iterative and incremental software development process. In this chapter, quality assurance is an overarching term referring to all dynamic (e.g., testing) and static (e.g., reviews) activities and practices that are applied to improve the quality of the software product as part of the development process. The term testing is used in the meaning of dynamic evaluation of the software product against some criteria, which includes both defect detection and evaluation against non-functional criteria. Quality assurance includes both preventive and corrective actions.

This chapter does not address software process improvement in general. A discussion on SPI can be found in a separate appendix. Specific testing and quality assurance practices are not described in detail, due to the large number of potentially useful practices and techniques. There are several books covering such techniques, and we point the reader to them where appropriate.

In this chapter, we first discuss the fundamental concepts of software quality assurance from the viewpoint of iterative and incremental development. Following this, we discuss the concepts of quality and good enough quality, as well as the ideas of test mission and test strategy. These concepts are useful when planning an overall approach to quality assurance. The core of the chapter is the description of the different time horizons of the Cycles of Control framework from the quality assurance point of view. We describe how viewing and constructing QA approaches through those time horizons can help you see quality assurance as a part of the iterative and incremental development process — not as a phase at the end of the development process.

5.2 Software Quality

In this section, we discuss some basic concepts of quality assurance and testing. We first discuss what is meant by software quality in general, and propose a stakeholder-based approach. Then, we present the idea of *good enough quality* followed by a discussion of various quality dimensions. The section ends with a discussion of the use of *quality risk analysis* as a means of communicating high-level quality goals.

Defining Software Quality

Software quality is a very complex, subjective, and abstract concept and it is hard to explicitly define what software quality actually means, as the following quote shows:

> Quality is a complex and multifaceted concept. It is also a source of great confusion — managers frequently fail to communicate precisely what they mean by the term. The result is often endless debate, and an inability to show real progress on the quality front. (Garvin 1984)

Garvin (1984) presents an extensive discussion of different viewpoints and dimensions of product quality in his article *What Does "Product Quality" Really Mean?* Some other authors have tried to give short and explicit definitions for quality. A few examples include:

- The degree to which a system, component or process meets specified requirements.
- The degree to which a system, component, or process meets customer or user needs or expectations. (IEEE 1990)
- Quality is the absence of errors.
- Quality is the value to some person. (Weinberg 1991)

The problem with these short definitions is that only some of the stakeholders in any given project can agree to, for example, the definition that quality is absence of errors or conforming to specifications. There are some fundamental reasons to the confusion that the concept of software quality arouses. First, different stakeholders usually have different views of the quality of a software product. Second, software quality is difficult to measure, even if we could achieve consensus of what we actually mean by it. Therefore, it is important to think about what quality means for different people or stakeholders of a software development project (e.g. end users, customers, administrators, developers, testers, product managers). More quality for one person might mean less quality for another, so you need to decide whose opinion matters the most. For example, if the software architect's view of good quality is different from the view of the end users, whose opinion is more important? Naturally, this depends on the quality attribute at hand. You might, e.g., appreciate your software architect's opinion on maintainability, while the end users opinion might be important when it comes to usability.

Because quality can mean many different things to different people, it is not sensible to try to formulate a generic definition of high quality. Instead, you should try to find out who the important stakeholders of your software product are and whose opinion on quality to listen to. Also, consider the risks of providing unacceptably low quality for some minority of stakeholders.

> Do not let the concept of quality confuse you — what has value to you and your stakeholders is important.

Quality Dimensions

Quality dimensions describe different views of quality and a certain test type typically concentrates only on one or two quality dimensions. A common classification of quality dimensions can be found in the ISO 9126 quality standard in which software

quality is defined with six quality characteristics; *functionality, reliability, usability, efficiency, maintainability,* and *portability*. Each of these characteristics is further divided into 3-5 quality attributes as shown in Figure 5.1. Another example of quality dimensions is found in (Garvin 1984) where Garvin describes eight dimensions of product quality.

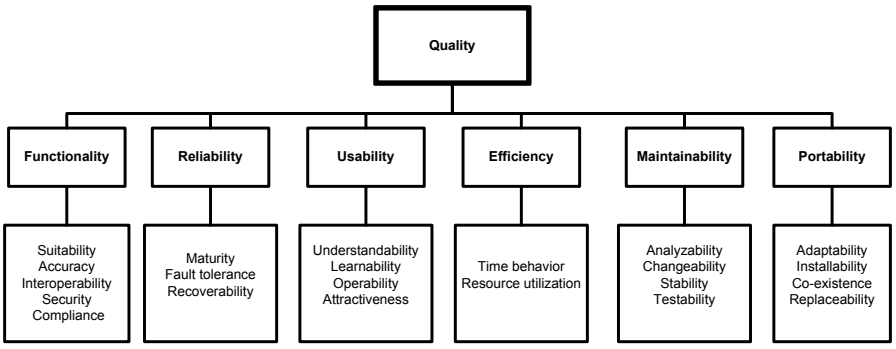


Figure 5.1: Quality characteristics and attributes in ISO 9126

These quality dimensions or characteristics and attributes can serve as a useful checklist of dimensions of quality that might be worth considering when planning a quality assurance approach in your development process. However, these kinds of classifications do not provide help for the actual QA and testing work in practice. It is easy to say that we aim for high efficiency, faultless functionality and good usability in our products, but making these aims show in your development process and in the everyday work of the developers and testers is what really matters. In addition, you have to prioritise your quality goals to be able to concentrate improving the most important dimensions of quality.

> Quality dimensions act as a checklist of aspects of quality to consider
>
> However, you need more concrete goals to guide work in practice

Good Enough Quality

By concentrating on the viewpoints of the most important stakeholders, you do not have to strive for perfect quality. Instead, your goal is *good enough quality.* Good enough quality means that your goal is to do everything that is economically feasible for the quality of your product in order to satisfy the needs of your stakeholders as well as possible. This does not mean that you skip testing, ship the product, and trust your luck. It means that when you have performed quality assurance and testing activities to the extent that doing, e.g., more testing would cause more harm than help, then you have achieved good enough quality (Bach 1997).

> The product is good enough when further improvement would be more harmful than helpful.

Quality Criteria and Risks

You can use *quality criteria* to define your quality goals and use them when evaluating the product quality. Quality criteria describe the desired quality level and thus, in order to be useful they should be as concrete and measurable as possible. Often it is rather challenging to define good high-level quality criteria that could actually be used to measure and assess the quality of a product release. A common problem is that the quality criteria are defined at a far too abstract level, which makes it impossible to say if the product actually meets the criteria or not. In addition, too general quality criteria do not provide any guidance or goals to the software developers and architects. Quality criteria are often specified by outlining various quality dimensions and attributes and describing their target values. An alternative approach is to concentrate on identifying and prioritising quality risks. This can make it easier to get a more concrete grasp of quality and to find concrete actions to improve and measure the quality level.

> Good quality criteria can be used to evaluate and track the quality of a product during the development project.

Quality Risk Analysis

In his book, *Critical Testing Processes* Rex Black (2003) presents an approach and a process for analysing quality risks. The idea is to collect the views of quality from the different stakeholders of the development project in the form of quality risks. All important stakeholders are surveyed and the information of their ideas of the biggest quality risks are gathered, together with their assessments of the impact and priority of each quality risk. In this phase it is useful to use some checklist covering the most important quality dimensions to ensure that risks on all important quality dimensions are considered. The most important stakeholders who should participate in this risk analysis are those whose perceived quality matters the most and those who have insight into the desired quality dimensions and the potential quality risks of the system. The book by Gerrard and Thompson (2002) is another source of information on risk-based testing.

There are several techniques available for gathering quality risks, for example, informal brainstorming, the use of ISO 9126 or other quality standards as checklists, using cost of exposure or failure mode and effect analysis (Black 2003).

Regardless of the technique or method used for collecting quality risks, you should prioritise the risks and classify them, e.g., into major, tolerable and minor risks. The major risks should be paid attention to and addressed extensively. You might choose to address the tolerable risks to some extent, but you probably will find that you have to live with many of the risks in this category. The minor risks can typically be ignored.

If you can successfully identify your most important quality risks, they help you get a concrete feel for the issues when you plan your quality assurance strategy. The primary goal of your QA strategy is to address the major quality risks. The secondary goal is to address the tolerable and minor risks as efficiently as possible.

Quality risks can be used as a tool to create and communicate the quality criteria and goals.

Identified quality risks provide concrete goals for the quality assurance activities.

When you have identified your high-level quality goals and perhaps performed some sort of quality risk analysis, you have a concrete idea of what quality means to you and what level of quality you are aiming for. The next step is to define a quality assurance approach consisting of a quality mission, a quality strategy, as well as quality assurance goals for each increment. How to do this is the subject of the next sections.

5.3 Defining Your Quality Assurance Approach

Quality Mission

The quality mission sets the high level purpose of your QA activities. Kaner, Bach and Pettichord (2002) state that a test mission describes the essential problems that you as a tester must solve to be seen as successful by your clients. You can see the quality mission as an extension or application of this definition of test mission.

The quality mission is a statement that communicates how, and in what role you are using quality assurance activities to achieve your quality goals. By this mission statement, you describe what the role and goals of the people who perform the testing and quality assurance tasks in your organisation are. If you have identified several different quality goals and risks, it might be useful to have several quality missions. The quality mission for developers is probably also different from the quality mission of dedicated test specialists. Some examples of a quality mission include:

- Find the most important defects early
- Provide useful information on risks of the release promptly at the end of each increment cycle
- Demonstrate the existence of the business requirements in the software
- Measure and evaluate the performance and reliability of the software
- Find the maximum amount of concurrent users and provide performance measures for different user loads.
- Rigorously follow certain methods or instructions
- Minimise the testing time and cost

The quality mission should include success metrics by which you can judge the success of your quality assurance activities. If your clients will judge your testing efforts by detailed conformance to rigorous and defined test procedures then it should show in your test mission. A different case would be if the goal of your company were to release the new product as soon as possible by taking some quality risks. In that case, your quality mission should somehow reflect that goal.

> HardSoft had not defined their quality missions for either of their products, so Jeff, the CEO, asked the product managers Jeremy and Jay to write down the quality mission for the products, Widget and Gadget. Jeremy and Jay started to work on the quality mission with the Development Team Leader Jill, Jenny, the Chief Architect, Senior Developer Jack and Jake, the Quality Engineer. After a

couple of iterations they had formulated two high level quality missions for each of the products:

Widget

- Find all critical, severe and moderate severity defects before the end of the release cycle.
- Ensure that the performance and reliability of the product meet the goals by providing measurements and risk assessment at the end of each development increment.

Gadget

- Demonstrate that all implemented features function correctly and find all defects that would be relevant to the majority of users.
- Produce high quality code that is easy to read and understand, well documented and structured and easy to develop further, extend and maintain.

As you can see, the quality missions for the two products are quite different. The main reason for this is the fact that Widget and Gadget are very different products and therefore have differences in the importance of various quality dimensions, as well as different quality risks.

Widget, the older product, is a traditional server side application that is close to databases and implements a lot of domain specific business logic. Widget is a server that is required to handle up to 10000 daily users and has high performance and reliability requirements. The release cycle of Widget is rather long — three months — and commercial releases are made at the most twice a year. The core functionality and the interfaces of the product are quite stabile. A typical release of Widget does not include many new features, but the implementation and testing of those features can be quite complicated. This makes reliability, performance, as well as functionality the most important quality dimensions for Widget.

Gadget is a new client side application that utilises the services of Widget, but that also provides a large amount of client side functionality covered with a modern and attractive graphical user interface. Gadget is a new product that has a rapid one-month release cycle. Lots of new features are included in each release. This means that the most important quality dimensions for Gadget are functionality, usability and maintainability.

A quality mission outlines, at a high level, the purpose of your QA activities.

Quality Assurance Strategy

The next step towards a well-understood quality assurance process is defining the quality assurance strategy. When you have found your quality goals and your quality mission, you probably find it necessary to define a QA strategy for each of the quality missions. The same strategy does probably not work with different missions if the missions aim at solving very different problems.

The QA strategy concretises and specifies the relationships between the quality criteria, the quality mission and the testing process. The QA strategy describes the concrete methods, techniques and activities that the development team and testers will use to build quality into the software, as well as to test the resulting products. The QA strategy describes at least the following issues:

- How will you cover the product to satisfy your quality mission and reach the quality goals or address the identified quality risks?
  - The highest priority requirements, functional areas or components of the product?
  - New functionality and changed code?
  - Areas that are the most likely to have problems?
  - The most used areas, functionality and components?
- What techniques or methods do you use to assure the quality of each area of the product?
  - What specifically will you test — and what specifically will you not test?
  - What techniques will you use to create and execute the tests?
  - What is you automation strategy
- How will you recognise defects when they occur?

The QA strategy is a very concrete asset that describes the actual techniques and activities that are performed in order to assure software quality. You should plan the QA strategy separately for each product, in order to achieve a concrete enough level of strategy to be useful in practice. The test automation strategy is an essential part of the QA strategy and more information about test automation can be found, e.g., in the book *Software Test Automation* by Fewster and Graham (1999).

> The CEO, Jeff, was pleased to notice that Jeremy and Jay had put together simple and clear quality missions for both products. Now, he wanted to know how development was actually going to fulfil this mission and asked for their quality strategy. Jay and Jill had actually already thought about this and were proud to present their first draft of the QA strategy for Gadget. Here is their short description of the quality strategy for Gadget. They had divided the strategy according to the QA mission into two different approaches:
>
> **Demonstrate that all implemented features function correctly and find all defects that would be relevant to the majority of users:**
>
> - Automated unit tests for every component and feature using the JUnit framework (and NUnit later after porting the Gadget into .Net)
> - Executing the unit tests every night and require that 100% pass
> - Code reviews for all production code
> - Documented functional tests for each feature that is implemented
> - Automated system level functional tests where appropriate
> - System testing in the 3 most important environments
> - Integration tests with the Widget server
>
> **Produce high quality code that is easy to read and understand, well documented and structured and easy to develop further, extend and maintain.**
>
> - Clear coding standards that are continuously maintained
> - Peer reviews to enforce coding standards
> - Certain static code metrics and analysis using tools to prevent bad design
> - Clear thresholds for code metrics and defined procedure for handling the deviations from these thresholds
> - Using test-driven development to document design by test-cases ensure testable code

The quality strategy states what you are going to do to assure the quality of the product.

Planning the Quality Goals of each Release and Increment

The third step when defining your quality assurance approach is to define the quality goals for each product release and increment. When doing this, you should make sure that all quality assurance and testing practices reflect the high-level quality goals of the organisation. In addition to these goals, each individual release has its own, detailed quality goals. However, it is necessary to break these down into increment-level quality goals in order to make the quality goals visible also on the heartbeat level. This is important, since this helps directing the everyday work of the development organisation in the right direction from a quality point of view. The following fictional anecdote highlights some common problems with high-level quality goals.

> Jeff, the CEO, had a nice chat with a few new sales prospects about his company's newest product under development. Based on first hand customer feedback Jeff announced that the company's highest priority quality goals for the next release were to be able to handle 10 000 concurrent users and to provide sub 0,1 second response time for every request, and to have 24/7 reliability — of course. The developers tried to ask him, what he actually meant with concurrent users and explained that 10 000 concurrent users is impossible to achieve — even 10 000 daily users would be hard to achieve with their current product. They also asked why every request had to have a response time below 0,1s, especially when even Jeff should know that their system currently took more than 5 seconds to respond to most nontrivial queries. Anyway, Jeff just replied that this was what he had sold and it will be the goal for the next release.
>
> After the first two increments when the new architecture prototype was completed, the performance measures showed that the system could support around 10 concurrent users with an average response time of 3,7 seconds. Too general and unachievable quality goals did not help the developers improve the performance of the system. However, the performance was quite satisfactory for all their current customers. Actually, at their biggest customer, the system had over 1 000 users, but because the system was not used frequently by any single user, the performance was quite satisfactory.
>
> After discussing this learning experience with Jermaine, the sales director, and Jeremy, the product manager, Jill, the development team leader, replaced Jeff's high level quality goals with one goal that stated that the product should provide performance that was satisfactory for an installation of 10 000 users. Based upon this high level goal she then chose suitable subgoals for the next release. The release goal was to make the system process 10 selected (most frequently used) requests per second and to have a response time for the most important requests below 1 second under this load of 10 requests per second. Now, the quality goals of each increment could be made measurable. This helped the developers concentrate on specific performance issues that could be constantly measured and tracked.
>
> The new goals that Jill stated did not guarantee performance in all situations, but as they were concrete and measurable, they helped the developers to concentrate on improving the performance of the most important functions of the system.

The quality goals on the release and increment level should help you track and measure the selected quality attributes during the development project, and to make sure that the product actually satisfies the release quality goals at the end of the release. Setting these goals is a task performed in the release and increment planning control points. The goals are tracked using the test results from each increment and used to control the contents and themes of the subsequent increments.

The goals might include, for example, measurable performance goals for the first or second increment, in order to evaluate the ability of the pure system architecture to provide the required performance. No matter how wild and vague the promises of the marketing people are, the development quality goals must be concrete, measurable and achievable in order to be useful for tracking and directing the development work.

> Quality goals must be concrete and measurable in order to be useful.
>
> Vague quality goals might be good for advertising, but they are useless in directing the development work.

Quality Debt

The field of software design uses the concept of *technical debt*, which means that the architecture and design of an evolving system needs constant maintenance and if this maintenance is neglected, it can be thought as a technical debt that we are taking. At some point in time, we must repay the debt if we want to be able to further develop the system. The same analogy holds for quality assurance: we might be able to skip testing and other QA activities for a short period of time, but this leads to an increasing quality debt — an increasing number of bugs in the software and decreasing quality. This debt, as debts usually, has a high interest rate. If it grows too big, we end up into a situation in which all our efforts go to fixing bugs and trying to repair and patch the existing versions of the system instead of developing new features and modules. In other words, we can barely cope with the interest payments, and the quality of the system does not improve.

This increasing high interest debt is one reason why leaving quality assurance activities until the end of the development project is not an optimal solution. In iterative and incremental development, we do not want to let the debt increase, not even during a single release or even one increment. Therefore, we must include quality assurance as part of the heartbeat activities. This helps keep both the debt and the interest low enough that we can afford to pay them at the end of the release cycle.

> Ignoring or performing quality assurance sloppily is like taking debt.
>
> The interest is paid by patching the defects after the release.

Next, we will discuss how to arrange quality assurance in iterative and incremental development in more detail.

5.4 Quality Assurance in IID

Sequential vs. Concurrent Testing

Testing is usually seen as a separate, destructive activity that takes place after the constructive phase of software development at the end of the development life cycle. In iterative and incremental development (IID) several separate testing phases at the end of each increment cycle correspond to this idea. In this book we take a different viewpoint to QA in IID. We see QA as an important part of the software development

process that cannot be separated into a single phase of the development life cycle. Instead, we think that it should be an integral part of each development phase and activity. This means that testing and development adopt the same rhythm, and must be well synchronised and connected. Successful synchronisation of QA activities with the development rhythm has a significant effect on the overall software development process.

Traditionally, testing activities are seen as a straightforward process of sequential tasks, e.g., in TMap (Pol, Teunissen, and Veenendaal 2002) testing is divided into the five tasks of planning and control, preparation, specification, execution, and completion. This viewpoint of testing is shown in Figure 5.2.

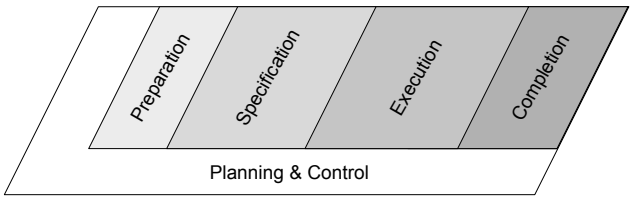


Figure 5.2: The TMap testing process (Pol, Teunissen, and Veenendaal 2002)

These activities form a sequential process that is usually performed in a single phase in the software development life cycle. However, in IID the testing activities are typically performed concurrently. Different functions and features are developed concurrently, and are in different phases of testing. There are not as clear project level phases as when using a sequential development life-cycle model. This means that the testing activities become disconnected from any specific development process phases. Instead, the testing activities are performed according to the status of the functionality or component the quality with which they are concerned.

The concurrency of testing activities does not mean merely sliding the testing phases slightly on top of each other. In IID, you must think of each of these testing activities separately and connect them with the time horizons of the IID process. For example, it might be necessary to separate test case specification from execution and integrate test case specification as part of the implementation tasks. Test execution might take place, for instance, after the integration of the functionality developed in the increment. Furthermore, the execution phase is often performed more than once, which does not necessarily mean that the specification phase needs to be performed several times.

The primary goal of software testing has traditionally been to find errors. A classic definition of software testing is "*the process of executing a program with the intent of finding errors*" (Myers 1979). However, testing has another important goal when using IID. As IID strives for short development cycles and frequent releases of stable software that "grows" in small increments, the role of testing as a provider of status information is emphasised. Testers are actually the best people to continuously provide information on the status of the development work and the quality of the software. Perhaps the most important goal of testing in IID is to provide useful information of the risks of releasing the product. This information should be provided promptly and timely — not only at the end of the development cycle when nothing can be done to mitigate or prevent those risks, but continuously at the organisational heartbeat pace. In addition, this information should be provided in an accurate, useful and understand-

able form.

Testing can be organised in many ways. Testers can be assigned either to a separate organisational unit or be integrated into the development team. In fast paced iterative development, we think that it is hard to totally separate testers from the development team. Most testing textbooks do emphasise that independency is a crucial requirement for the successful execution of testing activities and recommend using independent test teams. However, these books also point out the value of early involvement of testers. Thus, taking this advice to heart, it can be tempting to isolate testing as a separate function and organisational unit. Our experience with small software product companies shows that this is not a good approach, because the responsibility for quality cannot be outsourced. A separate testing unit cannot put quality into the product — it can only evaluate the software and point at the missing quality characteristics. We therefore think that relying purely on a separate testing unit is hardly feasible in IID. Instead, testing needs to be integrated into the development process at all levels. Some testing tasks might require special expertise or independence that make them better suited for assignment to a separate testing unit than to people that are close to development. In such cases, there is additional value in having an independent quality assurance organisation. However, we do not think that this removes the need for testers that are integrated into the development team.

The Tester and the Development Organisation

Making testing work is demanding also from the organisational perspective. Perhaps the most common problem is the "us-versus-them" mentality between testers and developers. There are several other aspects that make testing organisationally challenging. For example, it is often the testers who have to deliver bad news, the work of testing teams has many dependencies to the work and schedules of other teams, in many cases developers and managers do not understand what testers actually do. When time pressure builds up, the time scheduled for testing is often cut to get the product to market sooner or to give the developers more time to add features. The many people-oriented problems of testing organisations are addressed for example in the book *Surviving the Top 10 Challenges of Software Testing* (Perry and Rice 1997).

The role of testers in a development organisation can be very different depending on the size and type of the project and organisation. The tester's role in a development project depends primarily on the development process used. On a high level, there are at least three basic cases:

1. Testers form a separate organisational unit, a testing team.
2. The development team has separate tester roles
3. All developers are collectively responsible for testing, with no explicit testing roles defined or allocated.

Traditionally it has been argued that testers should be independent to guarantee that they can objectively evaluate the quality of the product. Even if there is no separate testing unit, at least developers should not be the only testers of their own code. On the other hand, testers and developers should collaborate and work together daily in order to facilitate communication and avoid the us versus them mentality and "throwing over the wall" effect. Thus, a balance has to be struck between the need for independent testing and the need to collaborate with developers.

Generally, testers can have many different goals in a development organisation, and these goals as well as the overall purpose of the testers help to find the right organisational structure. Some examples of goals for a tester are to:

- Find defects early and get them fixed together with the developers
- Provide useful information regarding the essential quality attributes and risks
- Help the developers to produce high quality code
- Provide an independent and formal assessment of the software quality

The Tester as a Professional

Testing is a profession, not a punishment. It is common to hear that some individuals were transferred to testing because they were poor programmers or sloppy designers. This is not a good approach. Both testing professionals and experienced software developers know that testing is a task that requires a lot of skill, knowledge, and experience. Testing is different from programming, and thus requires different skills. However, testers must know software development and programming intimately to be able to find the most important faults. Testing is not a boring and repetitive task that anybody can perform. Testing is much more than simply performing previously documented test cases. Testing is a task for creative, diligent and experienced people, who can continuously find new ways of identifying problems in the system and new ways of automating the laborious, repetitive parts of testing. Testing requires good communication skills and strong understanding both of software development and of the application domain of the software product.

Testing is not for everyone — testing tasks typically require many persistent actions, some level of interest in details and careful investigation of different scenarios and test results. One of the most important qualities of a tester is the ability to work with a destructive mindset — the ability to show that things do not work. A tester should also be good with people because he always has to tell people about their errors and criticise other people's work. Thus, it is usually a good idea to recruit people directly to quality assurance, and to look for other personality traits than when hiring coders.

Modelling and Understanding the Testing Process

Testing is a crucial part of incremental software development, but it still lacks good and useful models and frameworks. There are plenty of development models and methodologies for IID, but testing is all too often included merely by pasting an arbitrary phase called 'testing' at some suitable place in the development life cycle. General testing models or processes for IID are rare and the most common model that is used to describe, educate, and understand testing also in the context of IID is the V-model, depicted in Figure 5.3.

The V-model for testing was originally a simple extension to the waterfall model, and it is still most often used to describe the testing phases in development projects applying this life-cycle model. However, the V-model is not good for describing testing in IID, because it lacks the idea of continuous and repetitive increments in software development and testing.

The V-model describes *test levels*, which refer to different levels of abstraction, and the size of the target of testing. Commonly used test levels are unit testing, integration testing, system testing, and acceptance testing. In addition, the V-model

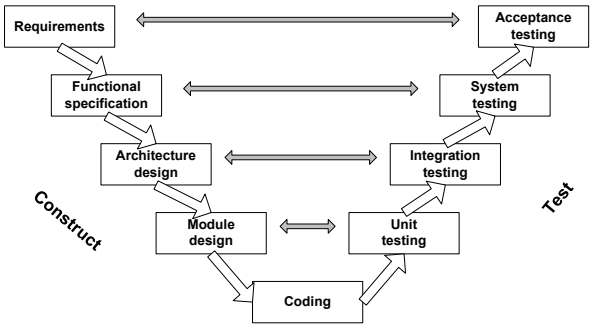


Figure 5.3: The V-model of software testing

includes both planning of testing during each development phase and validating the intermediate artefacts of each phase. However, using the V-model to describe the dynamic behaviour of the testing process is hardly possible in IID. If the V-model is used to describe the dynamic behaviour, it translates to a sequential waterfall life cycle, with different test levels describing sequential phases of the testing at the end of the development process. However, the CoC framework is designed with IID in mind, and we therefore use it to understand also the testing process. To aid us in this, we will need the concept of test levels taken from the V-model, as well as those of *test types*, and test *complexity levels*. It is important and useful to understand that these concepts refer to different dimensions of testing, all of which help us understand different aspects of the testing activities that we perform when developing software. We can perform different types of testing on different test levels and with an approach of different level of complexity. In the next sections we shortly describe the concepts of test levels, test types and the complexity levels of testing. For more information of the fundamental concepts of software testing we refer to the books of Kaner, Falk, and Nguyen (1999), and Burnstein (2003).

Test Levels

Test levels describe the level of abstraction of testing and the size of the target of testing. The most common test levels are:

Unit testing

The lowest level of testing (also called component testing, module testing, or program testing). Testing the individual components in isolation without the rest of the system. The concepts *component* or *unit* can refer to anything from a single class to a whole subsystem.

Integration testing

Testing two or more components together. The focus is on testing the interfaces and how the components interact.

System testing

Testing the whole system. Concentrates on the system level functionality and the non-functional features of the system.

Acceptance testing

The customer or end user tests the system in its real operating environment. Acceptance testing can be a contractual phase, in which the customer verifies the formal

requirements previously agreed upon. In product development it is often hard to distinguish acceptance testing from system testing.

Each test level has a different approach to testing and on each level the testing efforts concentrate on finding different types of defects. The names of the test phases in the waterfall or V-model are usually same as the test levels, but it is important to note that these two concepts are different. Test levels do not necessarily specify any process or sequential phases. In particular, you cannot use the test levels to describe phases of a development project in iterative and incremental development.

Test Types

Test types describe the testing activities that are coordinated in order to evaluate the software from one point of view; usually the goal is to evaluate one or a few quality dimensions of the software (see Section 5.2). Another goal of a test type can be finding certain classes or types of faults in the system.

Test types are traditionally classified into black-box and white-box testing techniques. Black-box techniques consider the system under test as a black-box and do not utilise any information about the internal structure or logic of the system. White-box techniques are based on the knowledge of the internal structure and, usually, the source code of the system. Examples of test types include:

- Functional testing
- Load testing
- Stress testing
- Reliability testing
- Usability testing
- Performance testing
- Security testing
- Configurations testing
- Documentation testing

Testing Complexity Levels

Testing complexity levels describe the complexity of the designed and executed test cases. Tests of lower complexity are simple to design and execute. Low complexity tests are, in addition to being simple, typically not very powerful in finding new defects in the software. Tests of higher complexity are more difficult to design and execute, but also more powerful in revealing defects and provide more valuable information about the software. An example of complexity levels is (Kaner, Bach, and Pettichord 2002):

> **Level 0 — Smoke testing**. Simple tests that show that the product is ready for serious testing. A sanity check that ensures that further efforts are not wasted in trying to perform more complex testing on a totally unstable build.

> **Level 1 — Capability testing**. Tests that verify the capability of each function of the product. The goal is to make sure that each function is capable of performing its task.

Level 2 — Function testing. Tests that examine both the capability and basic reliability of each individual function and subfunction of the product. Data coverage and complex test result evaluation methods. Boundary, stress and error handling tests.

Level 3 — Complex testing. Tests that involve interactions and flow of control among groups of functions to conform complex scenarios. The focus of evaluation is expanded and may include performance assessment, compatibility, resource contention, memory leaks, long-term reliability, and other types of quality criteria.

5.5 Using Time Horizons to Manage Testing

As we have seen in the previous sections of this chapter, in iterative and incremental development testing cannot be a phase that occurs at the end of the development life cycle. Instead, QA activities and practices have to be included as part of the incremental development that proceeds in short cycles with a defined rhythm. QA is a natural part of all development activities. We cannot view QA as belonging to some single phase or level of the development life cycle. As an example of this, we next consider how QA shows in the context of each time horizon of the CoC framework.

Common Testing Approaches

In this section we describe a few common approaches to testing in IID. These approaches illustrate typical ways of managing testing in an iterative development processes.

Automated Regression Testing

Frequent changes to the same code base is one of the main reasons behind the challenges in managing and organising testing activities in iterative and incremental software development. The most widely proposed solution to this problem is automated regression testing. It means that the frequently changing code is instrumented with automated tests that can be updated and executed whenever the code changes. These automated tests are used to build confidence in the system and to provide a fast feedback loop for developers who work with the code, i.e., implement new features, fix defects and refactor modules.

The automated regression testing approach is common in many agile methodologies. For example, in Extreme Programming (XP), automated unit and functional (acceptance) tests play a very important role in enabling the agile way of working.

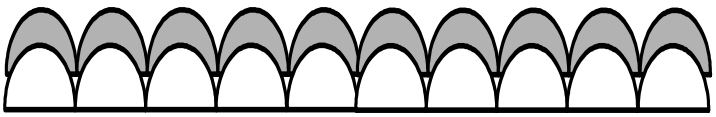


Figure 5.4: The automated regression testing approach

Figure 5.4 illustrates how this approach connects testing activities into each development heartbeat. Each development heartbeat is accompanied with a testing heartbeat (the grey cycles) that includes implementing automated tests for each developed functionality and executing these automated tests continuously in the same pace as the development heartbeats.

Stabilisation Phase

In the stabilisation phase approach, testing is included as a separate phase in (the end of) each increment. The goal of this stabilisation phase is to test the new functionality implemented during the increment and to stabilise the software system as a whole. The idea of the stabilising phase is to ensure the stability of the system at the end of each increment, integrate the whole system, execute tests and fix defects.

This approach is widely used in many incremental software development processes. (E.g. Microsoft's Synch-and-stabilise and RUP)

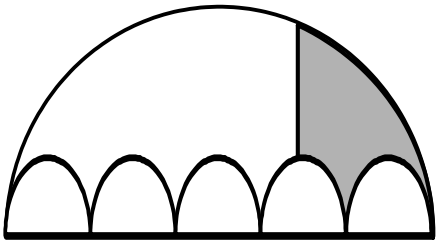


Figure 5.5: The stabilisation phase approach

Figure 5.5 illustrates how the stabilising phase approach can be viewed, for example, as a phase at the end of the increment cycle (the grey area).

Stabilisation Increment

The stabilisation increment approach is similar to the stabilisation phase approach. The difference is that in the stabilisation increment approach the whole increment, as depicted in Figure 5.6, is devoted to testing, integrating and fixing defects — no new functionality or new modules are implemented during the stabilisation increment.

This approach is close to the traditional approach with a system testing phase at the end of a waterfall-type software development life cycle. However, in IID the stabilising increment may still be necessary: for performing specific types of testing that cannot be carried out before the full functionality of the release is completed, for testing that takes a long time and requires that the product is rather stable (load and stability testing), or if extensive testing in rarely available environments is required. It might be a good idea to have a testing phase at the end of the release cycle, but you cannot rely purely on that — you have to consider the other time horizons of the development process as well.

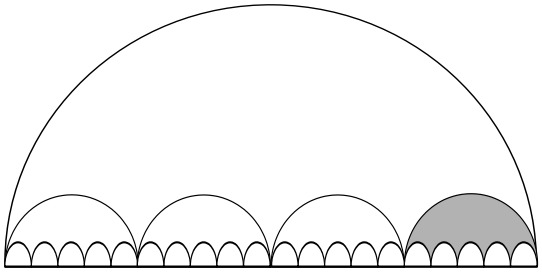


Figure 5.6: The stabilisation increment approach

Separate System Testing

> The separate system testing approach views testing activity as a separate function that is not directly part of the development team and does not have the same rhythm as development (see Figure 5.7). The testing rhythm is synchronised with the development rhythm by certain control points (e.g., at the end of each increment). In these control points the development hands off a version of the system to testing. Testing follows its own heartbeat rhythm, but the feedback to development (test results, issues, defect reports, etc.) can, and should, continuously flow to the development team.

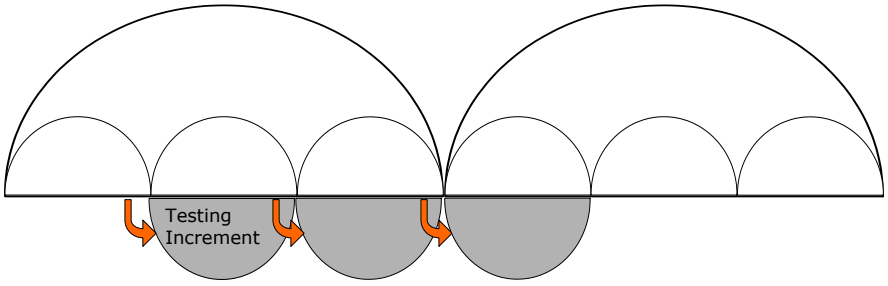


Figure 5.7: The separate system testing approach

Finding a Suitable Quality Assurance Approach

Any real world iterative and incremental software development life cycle will need more than one approach to testing and quality assurance. The above-mentioned base approaches provide several different viewpoints to testing, but any one of them alone is probably insufficient for assuring software quality. These sample approaches help recognise the needs and possibilities for quality assurance at the context of the different time horizons of the Cycles of Control framework.

The approaches often proposed for testing in IID usually take one point of view without considering the actual needs or restrictions of a particular product, project and situation. By constructing your QA approach using all of the time horizons of the CoC framework you can better identify methods and techniques that suit your needs. You will also have a more complete and accurate picture of the opportunities

for improving QA in your current development process. When we look at the QA activities through the different time horizons, we talk about *heartbeat, increment* and *release horizon quality assurance*. Each of these testing time horizons may include unit testing, integration testing and system testing — and, of course, different types of testing and testing of different complexity levels.

Different types and levels of testing catch different kinds of defects and provide different information. You need different testing approaches to implement your testing strategy and mission. You cannot place your trust purely, for example, in automated unit tests. It is helpful to think in terms of different test types and test levels, but these concepts do not make it clear who does the testing, when it is done, how often and how it relates to the overall development process. This is why the time horizons and structuring quality assurance through them is so important.

In the next subsections we describe how quality assurance can be viewed through the different time horizons of the CoC framework. The basic configuration of the framework with four cycles (strategic, release, increment, and heartbeat cycle), as described in Part I of this book, is assumed in the discussion.

Heartbeat Horizon Quality Assurance

The heartbeat time horizon covers all activities that are performed according to the heartbeat rhythm during the design, implementation, and testing of one single piece of functionality. For QA this means the activities that are performed for each piece of functionality before it can be deemed as implemented and completed. These activities are part of the implementation work and tightly follow the daily rhythm of the development activities. In the heartbeat time horizon, the QA activities are not delayed to any later testing phase. Instead, these activities are performed during implementation, as part of the coding tasks. The testing activities give instant feedback to developers and, therefore help drive development in the right direction.

It is particularly important to include the quality assurance tasks into the heartbeat time horizon. By applying and tracking QA activities as a part of the heartbeat level progress tracking, we can assure that the quality of every task that gets completed is good enough. This way, we can avoid the illusion of completed functionality that, in fact, needs another two weeks of reviewing, testing, debugging and fixing before it can be released for any serious use.

QA activities in the heartbeat time horizon apply to all heartbeat level implementation tasks and the tasks are not complete before these activities are performed. Note, however, that the QA activities can be different for different products and product areas. For example, the same QA practices are not suitable for GUI and back-end code. Since the QA tasks are tracked according to the heartbeat rhythm they automatically included in the lowest level progress tracking.

Heartbeat QA includes developer testing and debugging, but is by no means restricted to the testing tasks of the developers. Heartbeat QA tasks can include, for example, design document and code reviews, writing and executing unit tests, designing and automating system level functional tests, executing regression tests, reporting test results and bugs, verifying bug fixes, using static code analysis tools, and so on. Rhythm is once again key: heartbeat activities are managed and tracked according to the heartbeat rhythm and the different roles must communicate and synchronise their work in every heartbeat. The following anecdote highlights the idea of tracking the work of developers and testers at a heartbeat pace.

A new challenging week is about to start and the development team is gathered together at the Monday morning heartbeat meeting. Junior developer Jo has just reported that she has to increase her last week's estimate for the feature she currently is implementing, because the algorithm used has turned out to be harder to understand than she had expected. Jake, the quality engineer, reports that the functional tests for Jo's algorithm are completed and suggests that he could move forward to design tests for the next feature on Jo's list. However, Jill, the development team leader, smells trouble and asks Jake to improve the tests for Jo's algorithm, because the difficulties in implementation obviously increase the risk that the implementation might contain nasty bugs. She also suggests that Jake should review Jo's code in addition to the normal peer review of Jo, Joe (Junior Developer) and Jack (Senior Developer). In addition, Jill points out that, depending on how long it actually will take to get the algorithm done, it might be necessary to drop some tasks from Jo's list to ensure that all the important tasks on Jack's list get done and the increment goals are achieved.

In the next heartbeat meeting on Wednesday afternoon, the situation was better. Jo explained that she had solved all the problems and the algorithm component was done, so she would be able to move on to the next task on her list. Jill almost agreed to this but asked anyway if Jake had found any problems in his review. Jake then replied that actually he had not yet reviewed the algorithm, because it had taken so much effort for him and Jo to even get the component to pass the basic functional tests. Joe and Jake said that they were just about to submit their review notes to Jo before this meeting and that the notes are quite extensive and they recommend a major rework on that component. Now, Jill was glad that she assigned some extra effort to that task on Monday, because now she had much better idea of the situation and she decided to drop the less important features from Jo's list.

Next Monday the situation was over. Jo had fixed the algorithm implementation based on the review notes of Joe and Jack and, thanks to the Jake's new extensive set of partly automated tests for the algorithm, the verification of the rewritten code was straightforward. Everybody had enjoyed a relaxing weekend and was eager to continue with their new challenging tasks.

In the example above, we saw how heartbeat level tracking prevented Jill from thinking that some features were completed even if they were not, because the quality assurance was not done. She was also able to direct the QA resources to get the problematic algorithm done, instead of designing tests for features that eventually got dropped out of the increment. The communication on the heartbeat level brought up problems that could have easily been unnoticed until the late testing phases without including the QA activities as part of the heartbeat time horizon. The last but not the least lesson to learn from this example is that in spite of unexpected problems and overrun effort estimation with that one task, the team managed to get the problems sorted out and reprioritise the tasks. This enabled the team to continue working effectively, with a sustainable pace, without having to bear the stress of a slipping schedule.

Heartbeat quality assurance is part of the implementation task.

Heartbeat quality assurance is about building quality into the product.

Heartbeat level tracking reveals real progress and obstacles.

The natural flow of testing activities is cyclic and iterative. Common testing processes proceed through iterative cycles of testing activities (designing tests, executing

tests, reporting defects, fixing defects, verifying fixes, testing new and changed areas, and regression testing other areas). One major challenge in planning testing processes and projects is the difficulty of predicting how many successive test-and-fix cycles are needed during a certain testing phase. The number of test cycles depends on how good the implemented software is, i.e., how many and how severe defects and issues are found during the test cycles.

Kaner, Bach, and Pettichord (2002) propose that you should "treat test cycles as the heartbeat of the test process". In the context of the CoC framework, you should treat test cycles as part of the heartbeat of your development process. At least at the lowest level testing and development heartbeats (i.e. cycles) should not be separated. Instead, testing should be part of the development cycle. This means that the cycle is not completed until all tests are completed and the faults are fixed. Having testing tasks as part of the heartbeat time horizon makes the test cycles easier to manage. Estimating and tracking testing and fixing efforts can be handled due to the small granularity of the development tasks and the tight rhythm of tracking and controlling actions. However, on longer time horizons the difficulty of estimating test cycles remains.

Increment Horizon Quality Assurance

The increment time horizon includes all tasks and activities that are performed in order to successfully complete one development increment. While in the heartbeat time horizon we concentrate on getting one single piece of functionality or development task completed, the goal of the increment time horizon activities is to fulfil the increment goal. This includes all implementation, testing and review activities that are needed to ensure that the end product of the increment has good enough quality.

Testing and quality assurance on the increment time horizon is concerned with activities that are not performed for each individual implemented feature at the heartbeat pace. Instead, these activities are controlled and tracked at the increment level. Thus, the goal of increment testing is not to track at the heartbeat level whether a certain functionality or component is tested. Instead, the goal is to track whether, for example, the planned performance tests and functional system tests for the increment can be performed in time and the required results are available at the end of the increment. Quality assurance tasks on the increment time horizon can be realised by having a stabilising phase at the end of the increment, but it is not the only viable approach. The term increment horizon testing does not indicate that the tasks should occur at the end of the increment.

If an organisation has separate test specialists, their tasks are good candidates for increment time horizon tasks. These specialists write and execute tests during the whole increment and they must synchronise their work with the developers, e.g., by participating in development heartbeat meetings. However, their tasks are not directly synchronised with each development task. Instead, they have their own tasks that they perform as a part of the development team, in order to achieve the common increment goals. The synchronisation hand-offs can be, for example, the daily or weekly builds that the developers deliver. The testers can then synchronise their work with these builds.

On the heartbeat level the idea is to do the heartbeat level tasks and complete all reviews and tests before the feature is completed and the progress can be tracked. The increment level tasks are time-boxed, because the length of an increment is fixed. On the increment level testers have to prioritise their work. This means that it is

very important to track the progress of the work and communicate the situation to development leaders constantly. Otherwise, it is very easy to just ignore the fact that the increment level quality assurance lags behind development progress. This leads to short cutting the increment level quality assurance and increases quality risks, while the correct action would be to reduce the scope of the increment in order to get the increment level quality assurance activities performed.

Typical increment testing tasks are regression testing, performance testing, usability testing, and testing of quality dimensions that cannot be easily tested as a part of the testing of individual functions or components. In practice, all planned quality assurance activities for new functionality and code cannot be done at the heartbeat rhythm. If you want to delay these activities and pay back your quality debt during later increments, you can plan and track these activities through the increment time horizon. These QA tasks belong to the increment time horizon, because the activities are not related to the heartbeats of new functionality. In addition, testing tasks that require specific expertise or long periods of setup or execution time may be best managed at the increment level. Testing in different operating environments and in different combinations of environments are examples of tasks that usually cannot be included in heartbeat testing.

Increment quality assurance is evaluating the quality and providing information at the increment level.

Increment quality assurance is time boxed and tracked in the context of an increment.

Increment quality assurance is not a separate phase.

Release Horizon Quality Assurance

The release time horizon covers the tasks and activities for completing the whole product release. The goal of release testing practices and activities is to ensure the quality of the release at the release level. This might include more thorough testing of certain quality dimensions and final regression testing following completion of the final features or bug fixes. A common way of including release testing is to have a separate stabilisation increment at the end of the release project. However, this is a quite simplistic solution, and in many cases it might be better to describe the release testing tasks as tasks that are not tied to any single increment.

> For example, the release testing tasks could be planning and implementing a performance testing framework for the product, and use the deliverable of each increment as a prototype when creating the framework and test cases. All results that are gained during the design and execution of the performance tests are fed back to development even right after getting the first test to work with the first increment. Then the design and execution of the performance tests continue through the whole release project.

The release testing activities are typically dependent on the release type. Internal development releases and emergency patches might have different quality assurance practices than a major commercial release.

Release quality assurance evaluates the quality of the combined functionality of all increments prior to product release.

Release quality assurance is not a phase.

Strategic Horizon Quality Assurance

The strategic time horizon does not cover any specific QA practices. At the strategic time horizon high-level quality goals and risks are assessed, and the general quality strategy is developed and maintained. The quality strategy is executed by the quality assurance processes of the heartbeat, increment, and release time horizons.

An Example of Time Horizon-based Testing

The following example illustrates how quality assurance activities can be planned and implemented using the time horizons, as described above.

> HardSoft had defined the quality strategy for the Gadget product earlier and now they were ready to describe the quality assurance approach for each time horizon. The release cycle of Gadget is one month, which means rapid one-week increment cycles and an intensive one day heartbeat pace. This aggressive release pace is required to respond to the business needs. However, the rapid pace places restrictions on quality assurance. As many QA activities as possible must be included on the heartbeat horizon, because the rapid release cycle does not allow any unpredictable testing phase at the end of the release cycle.
>
> **Heartbeat quality assurance**
>
> - Developers write automated unit tests for all code using the test-driven development approach.
> - All unit tests are executed by the developer after implementing a piece of functionality, before the code is checked into the common repository (in addition, the tests are executed every night).
> - Clear coding standards are followed and continuously maintained by the developers.
> - Peer reviews are performed for all code and unit tests in order to find defects, enforce coding standards and ensure the quality of the unit tests.
> - Developers and quality engineers together document functional test cases for each piece of functionality.
> - All above tasks have to be performed before a piece of functionality is considered completed.
>
> **Increment quality assurance**
>
> - Jake, the quality engineer uses commercial tools and a custom test automation framework to automate the most important system level functional tests. These tests form the basic regression test suite that is executed for each increment.
> - Jake manually tests the new functionality of each increment in the three most important operating environments of Gadget. The developers help Jake in this task.
> - Jill, the development team leader, follows the standard set of source code metrics using static analysis tools. Based on the metrics and analysis of suspicious code modules she can include refactoring tasks the product backlog.

Release quality assurance

- Jack, the senior developer, and Jake perform integration tests with the Widget server. The scheduling, scope and complexity of these tests depend on the contents of the Gadget release.

5.6 Planning Your Quality Assurance Processes

The starting point in planning your QA processes is your QA strategy. The QA strategy expresses what you are going to do to verify and validate your work in order to achieve the quality goals, avoid risks and fulfil your increment and release goals. What the QA strategy does not tell you is when these quality assurance activities are done, who performs these activities and how often. The most important part of the QA processes is that these describe the role of the QA activities as a part of the whole development process.

Time Horizon-Specific Quality Assurance Processes

Designing the Overall Process

You will probably need a specific quality assurance process for each of the CoC time horizons. It is useful to plan your quality assurance processes by focusing on one time horizon at a time and thinking it through. In the context of CoC you have at least three time horizons that you need to deal with — the heartbeat, increment and release horizons. Each of these horizons corresponds to a particular cycle of control. The word "control" is important. When you perform some tasks at the heartbeat time horizon, it means that you control and track these tasks at the heartbeat pace. It does not mean that the tasks always are shorter than the heartbeat cycle, but you can coordinate, for example, heartbeat level development and testing tasks at the heartbeat pace. Similarly, when you do testing tasks at the increment time horizon it does not mean that testing happens only at the end of the increment — at the increment time horizon, you can test every day from the beginning to the end of an increment. At the increment time horizon your testing tasks are coordinated at the increment level, which means that you track these tasks against the increment goals and that the tasks are not necessary directly synchronised to the heartbeat level development tasks. On the increment and release time horizons QA tasks concentrate on evaluating the achieved quality and providing information about product quality and risks. This information is essential for tracking the progress of the project and helps identify necessary corrective actions.

As an example of viewing quality assurance approach through different time horizons we consider system testing that is usually viewed as a separate phase at the end of a release or increment cycle. This makes system testing particularly challenging because schedules are always tight and the time for system testing is squeezed. This easily leads to a situation in which there is not enough time for proper test design and execution, not to mention fixing the bugs found. By designing the testing approach from the time horizon point of view we can include system test case design as part of the heartbeat tasks as illustrated in Figure 5.8. This makes it possible to get developers and testers to prepare the test designs and cases together as a part of the design and implementation task. At the scope of the heartbeat time horizon the tests are tracked at the heartbeat pace and the progress of test design is also visible in the development progress tracking. Depending on the system, the first system test execution round can be part of the heartbeat (if the system is integrated and built frequently enough) or

increment time horizon as in Figure 5.8. By including testing tasks on the heartbeat level it is easier to track and control the development progress and ensure that the testing and quality assurance tasks get performed.

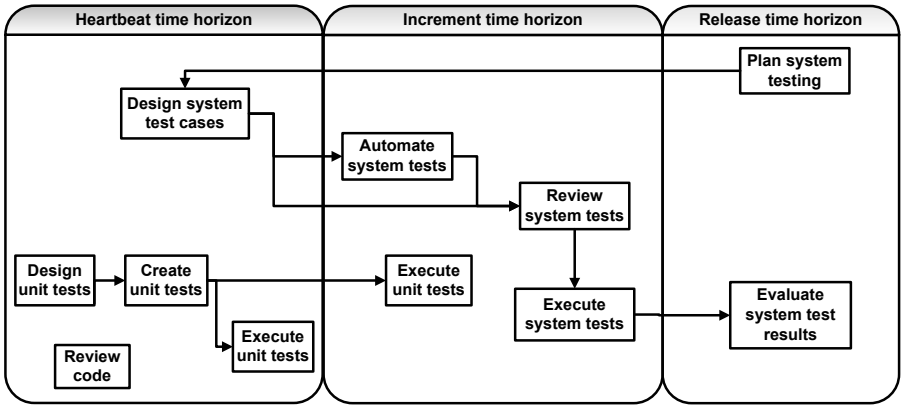


Figure 5.8: An example of viewing test levels and types through time horizons

You should think separately about what activities you could prepare, design and execute on the scope of each time horizon. For example, as illustrated in Figure 5.8, you can probably include some preparation and design of system level tests at the heartbeat time horizon, even if the execution of those tests is not possible in the heartbeat pace. Similarly, you can find ways of utilising unit level tests at increment and release time horizons, even if the unit tests are designed and created at heartbeat time horizon. In Figure 5.8 there is no timeline included; the figure only illustrates how the different test levels actually distribute among two or more time horizons.

This view can also be used as a tool to find ways of introducing the "test early, test often" approach. That is, you should find ways to move quality assurance activities from the scope of the release time horizon to the scope of the increment and from the increment to the scope of the heartbeat time horizon. That way you will be able to test early, because testing starts together with the design and implementation activities, and test often, because testing activities are part of the shortest heartbeat rhythm.

When you create a quality assurance process for each time horizon, you can use the following subsections as a reference.

Resourcing

You should plan who is responsible for QA activities and who performs the actual work and whose effort is required and how much. Another important aspect of resourcing is to decide when these activities are performed and how often. You also need to think about how these testing tasks are related to the other development tasks and how can you ensure that the required time and resources for the testing actually are available. Keep the high-level quality goals and the test strategy in mind to ensure that the actual test process at the level of each time horizon fulfils the test strategy and contributes toward achieving the quality goals.

- Who tests?
- When?
- How often?

Test Design and Execution

The core of the test process is the description of what types of tests are designed and executed, and how these tests are actually performed. It is useful to describe tests of different levels and different types, as well as the goal and purpose of each testing activity. A complexity description might be part of the goal description; describing a testing activity as smoke-test tells a lot of its goal and purpose.

You should describe separately what tests are designed, what design documents, test cases, scripts, etc. are created, and what tests are executed and reported. Usually it is useful to start test design as early as possible, but it is not always possible to execute the designed tests in the scope of the same heartbeat task. This splits the testing tasks onto several time horizons. For example, functional tests that are designed in the heartbeat time horizon might be executed on the increment time horizon (see Figure 5.8).

Tests that are designed

- Types and levels
- Test designs
- Test cases
- Test code (stubs, drivers, etc.)
- Test scripts
- Test data
- Test automation

Tests that are executed

- Types and levels
- Manual tests
- Automated tests
- Regression testing
- How execution is logged and reported

Other Quality Assurance Activities

Quality assurance is more than testing. When planning a quality assurance process for each time horizon, you should also identify other practices and activities that have to be performed to build and evaluate the quality of the software. These practices are usually good candidates for inclusion at the heartbeat time horizon, and thus work as a valuable tool for ensuring that the development organisation constantly maintains a sufficient level of quality.

- Reviews and inspections
- Static code analysis with tools
- Conventions
- Documentation
- Tracking the development progress

Reporting

By deciding and planning beforehand what results and metrics you want from the testing process, you can ensure that you actually are going to get these metrics. Plan what metrics you need to form a good enough view to your critical quality attributes. Plan what metrics you need in tracking the development progress. Plan also how are you going to utilise this information and how a task is considered to be completed. Remember also to consider the pass/fail criteria for each testing activity, and how the failed tests are handled.

- Defects and issues
- Performance, reliability, etc. metrics
- Progress metrics
- Pass/fail criteria

Maintenance of Testware

Several documents and data artifacts are created during testing activities and these need planning, configuration management and maintenance just like all other software development artifacts.

- Test environments
- Test designs, test cases, test code
- Test data
- Tools, configurations

The quality assurance process is specific to each time horizon.

The quality assurance process communicates:

- Who performs the QA activities and when
- What results are provided and how the results are used

Figure 5.9 summarises the quality assurance approach and the main concepts that were presented in this chapter. The high-level *quality goals and risks* define the quality level aimed for. The *quality mission* is a statement that highlights the high level purpose of the QA activities. The *QA strategy* is formulated based on the quality goals, risks and the quality mission. The strategy tells how, and with which means the goals are achieved, the mission fulfilled and the risks addressed. The *QA process* implements the QA strategy in the context of a CoC-style development process. The QA process is divided into *release*, *increment* and *heartbeat* time horizons and each time horizon has its own, specific quality assurance process. The quality goals and identified risks works as input for the *release goals* that are specified separately for each release cycle and further divided into *increment goals* for each increment. These goals help directing and controlling the QA process on the level of each time horizon. Note that the approach is product specific and feasible QA approaches for different products are different.

Using Test Information to Control Development

Testing has two major goals in iterative and incremental software development. The first goal is to find defects and other issues in the software. The second goal is to

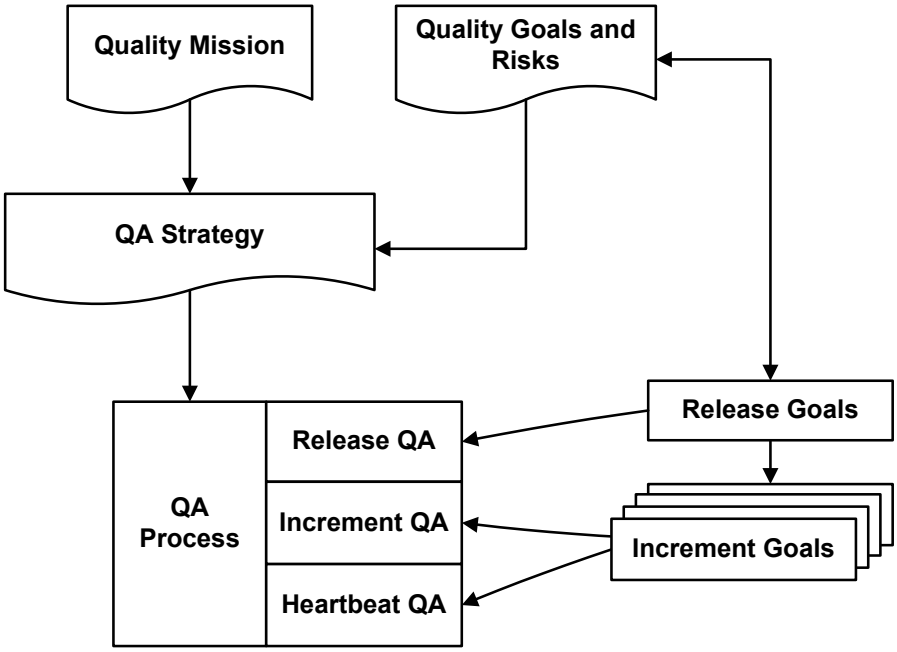


Figure 5.9: QA concepts and their relationships

provide useful information for the developers and managers. This information consists mainly of metrics of the progress of the development work and the quality attributes of the software under development. This information can be used to assess quality risks and to steer the development work.

One of the most important skills of a testing specialist is communication. A tester has to be able to communicate the results of his work to other people and other teams. In addition, the testers have to be able to communicate to managers the relevant information about the quality and risks of the software. They have to know how to present the important and relevant information in a comprehensible form that can be understood quickly and easily. They must be able to provide this information timely and accurately, without too much irrelevant noise or complicated metrics that managers or developers cannot understand.

You need to have a user for all reports and metrics that testing produces. You must also know for what purpose the results are used. The information must be timely available to assist managers to make decisions on the direction of the project.

In the Cycles of Control framework the information that testing activities produce is utilised in the control points of the development process, see Figure 5.10. You can see that the information that you use in controlling and steering development efforts in the time horizon of each cycle, must be produced and delivered at the pace of the smaller cycles.

> If you want, for example, to focus your efforts to stabilising the most important features of an increment and leave the “nice to have” functionality to upcoming increments, you need to be able to evaluate the quality level of the important features long before the end of the increment. This means that the information of the quality level of the most important features must be delivered in the heart-

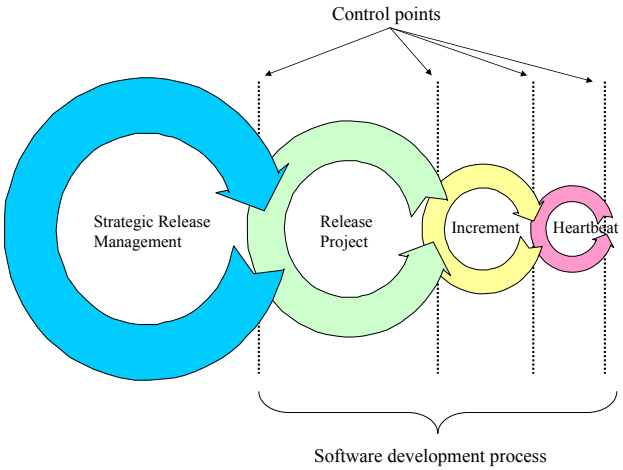


Figure 5.10: Control points of the CoC

beat rhythm in order to be useful in prioritising the tasks during the increment cycle. Similarly, if you want to utilise the results of performance testing when planning the contents of the next increments, the results have to be available at the increment pace.

The Cycles of Control — an Integrative Framework

There is no "one size fits all" solution for testing techniques, practices or tools. Therefore, you must select your testing techniques and practices based on the actual product and project context.

The different time horizons give you a good starting point when you think about different possibilities and solutions for quality assurance and testing. By considering the requirements and restrictions for QA activities on the level of each time horizon you are more prepared to make well considered and sensible decisions. As we described in the previous section, you must consider where and when you want to utilise the results of your testing activities. Depending on that you should put the activities in a certain time horizon, which again places certain restrictions to the activities.

By viewing QA practices as a part of the development process on the time horizons of CoC framework, you will be able to find new ways to apply QA practices earlier and more frequently in the development life cycle. This helps achieving a fast paced development rhythm without the common bottleneck effect of testing phase before each release.

The Cycles of Control framework and the viewpoint of the development rhythm gives you a good context in which you can combine different approaches and techniques for quality assurance. The cycles and time horizons of the CoC framework gives you advice about where in the development process each technique and practice belongs. It helps you put the separate techniques into a context where you can better assess their suitability to your needs. You can use the rhythm of each cycle to asses how realistic it is to get a certain technique or method included in the pace of

that cycle. Time horizons show when each method is applied and where the results or outputs of each applied method are available.

By describing quality assurance using the same cyclic framework as the development process makes their dependencies and relationships explicit, for example between testers work and developers work. When testing is not described as a phase, but rather as part of software development, it is easier to understand how developers and testers can together develop the product and be responsible for the quality of the work products.

References

Bach, J. 1997. Good enough quality: Beyond the buzzword. In *IEEE Computer* 30 (8): 96-98.

An article that describes the concept of Good Enough Quality. Describes the ideas behind the concept and presents a four part framework for good enough quality.

Black, R. 2003. *Critical testing processes.* Boston: Addison-Wesley.

Burnstein, I. 2003. *Practical Software Testing*. New York: Springer-Verlag.

A thorough text book on software testing. Describes all the fundamental software testing concepts and techniques. Also describes the Test Maturity Model (TMM), which is used as a framework throughout the book.

Crispin, L. and T. House. 2003. *Testing Extreme Programming*. Boston: Addison-Wesley.

Describes Extreme Programming from the tester's viewpoint. Presents the role of testers in XP and describes how testing should be done in an XP project.

Fewster, M. and D. Graham. 1999. *Software Test Automation*. New York: ACM Press, Addison-Wesley.

A basic guide to software test automation. Presents realistically the benefits and challenges of test automation, describes how test automation should be used to be useful, and gives guidelines for automation tool selection.

Garvin, D. A. 1984. What does "product quality" really mean?. In *Sloan Management Review* 26 (1): 25-43.

An interesting discussion on how product quality is viewed in different disciplines. Presents five different quality definitions and describes a framework for eight dimensions of quality.

Gerrard, P. and N. Thompson. 2002. *Risk-Based E-Business Testing*. Boston: Artech House Publishers.

Describes a risk-based approach to testing and test management. The book is written in the e-business domain, but the main ideas can be utilized in other domains as well.

IEEE. 1990. *IEEE Std. 610.12-1990, Standard Glossary of Software Engineering Terminology.* New York: IEEE.

Kaner, C., J. Falk and H. Q. Nguyen. 1999. *Testing Computer Software*. New york: John Wiley & Sons.

Kaner, C., J. Bach and B. Pettichord. 2002. *Lessons Learned in Software Testing.* Ne York: John Wiley & Sons.

A book representing the context-driven school of testing. Presents a lot of useful lessons, practices and ideas in the form of 293 short lessons to learn. A book that makes you to rethink many of your assumptions about software testing. Gives lots of good ideas for real world testing work.

Myers, G. J. 1979. *The Art of Software Testing.* New York: John Wiley & Sons.

The fundamental theories, techniques and concepts of software testing.

Perry, W. E. and R. W. Rice. 1997. *Surviving the Top Ten Challenges of Software Testing.* New york: Dorset House Publishing.

Describes the top 10 people issues of software testing and provides guidelines for addressing those.

Pol, M., R. Teunissen and E. van Veenendaal. 2002. *Software Testing A Guide To TMap Approach*. Boston: Addison-Wesley.

Weinberg, G. M. 1991. *Quality Software Management.* New york: Dorset House Publishing.

Chapter 6

Technical Product Management

Jari Vanhanen

6.1 Introduction

From the perspective of a software development organisation a software product consists of several types of items such as requirements, design models, source code, test case specifications, and development tools. All these items can have different types of relationships with each other. Some items are created based on the information in some other items, e.g., test cases are based on requirements. Some items are compositions of other items, e.g., a sub system is typically composed of several source code files. Realising that many things in this complex structure are under constant change is the reason for technical product management, which is the topic of this chapter. First, we discuss factors affecting technical product management and its goals. Then we discuss the most important areas for achieving these goals, namely version control, build management, change control, and release management.

6.2 Factors Affecting Technical Product Management

Several factors affect the complexity of technical product management. These factors are related to the product itself, to its development process and to the development organisation. The factors include at least the following:

- software size
- software complexity
- software quality
- number of variants
- amount of use and reuse
- number of changes
- development model
- development team

The degree of complexity related to each of these factors is determined by various business related product management decisions. Sometimes the effect can be understood easily beforehand, sometimes the effect is much more complicated. The factors are described in more detail below.

Software size. As the size of a software product grows, the number of items to be managed increases. The growth is not limited just to the source code, but realises also as an increasing amount of requirements, design models, test cases, manuals etc., which all need to be managed. Software size should be considered, whenever making a decision of adding one more new feature to the product. Sometimes it might even be advantageous to drop out some features from the product.

Software complexity. Much of the complexity of software comes from the various, difficult to understand dependencies between different items. Implementing new features or fixing bugs, dividing work between developers, and overall management of a complex product is harder. Clear architecture helps to decrease complex dependencies, but as most software products evolve, the decay of the architecture and increase of the complexity is a constant threat. Allocating resources for continuous refactoring of the code base is a good way to fight against the decay. Unfortunately, often all the resources are put into developing more and more features as long as it is somehow possible.

Software quality. The requirements for quality may be high due to the criticality of the software, but also due to the high cost of providing updates for mass products. The higher the requirements for quality, the more control is needed for development and quality assurance activities. The need for control increases the amount of information that needs to be collected and managed, such as code review and test results.

Number of variants. Several parallel versions, i.e., *variants* of a software product may be necessary, e.g., for supporting different platforms or specific customer needs. Managing several variants is complex, because there are awkward dependencies between variants. For example, when a bug is found in a piece of code, which exists as several variants, it is likely that the same bug exists also in the other variants. The bug may have to be fixed and tested to all of them. However, the use of variants is not a bad choice if variance cannot be avoided, because if different variants are considered separate products and managed totally separately the amount of overhead may be even higher. Well-planned version control practices help to manage variance, but confining the variance to as few places as possible in the code is still recommended.

Amount of use and reuse. The more installations a software product has, the harder it is to get them all updated, e.g., when critical fixes are made. If several subsequent releases are in use and supported, the fixes must be implemented separately to all these releases. If the software is re-used by some other development projects either using the copy-paste method, or as components, these parties are also interested about updates.

Number of changes. Managing changes to all levels of work products from requirements to source code and test cases, i.e., change control, is a crucial part of technical product management. A disciplined, but efficient process is needed for managing requests for change, in order to confirm that all requests are handled, prioritised, and tested after their implementation. When using iterative and incremental development (IID) it is natural that changes emerge. Handling a change request and its implementation almost always requires more effort than if the request would have been known before doing the work for the first time. The amount of overhead depends on several factors, such as when the change is found and implemented. Finding a problem in a requirement is much easier to fix before the requirement has been included in technical design, code, and test suites. Implementing changes late, just before a release, increases the risk of introducing bugs that are not detected before the release. It may also be more efficient to postpone the implementation of some less critical change request into the future increments, when using IID.

Development model. When using an IID model instead of a waterfall model, it must be possible to make quality releases frequently and thus with minimal overhead. IID allows changes to requirements later in the development making long-term plans for the software more vague and thus designing the system architecture harder. The architectural design may be changing even late in the development causing problems in managing the parts of the system and their dependencies.

Organisation. As the number of people in the development organisation grows, more formal processes are needed to facilitate communication and avoid disturbing other people's work. The need for, e.g., a large number of developers or scarce experts may lead to distributed development projects, which cause even more challenges for sharing the project's material and other required information.

When making decisions that increase the organisations flexibility for making business, the factors presented above typically get more complex. If short time-to-market is a necessity, then a larger development team should be used. If the strength of a company is in tailoring the product to each individual customer's needs, a high number of variants is accepted. Because of these reasons the complexity of the factors is typically not minimised. However, the additional costs and risk factors caused by the complexity of technical product management should be carefully analysed in parallel with the presumed business advantages. An organisation's capability of doing technical product management must also be taken into account. The situation should not be made too complex before the required capabilities have been achieved.

The role of technical product management is to minimise the required overhead and probability of risks with effective and efficient practices and tools. It must also allow maintaining the development rhythm, e.g., through nightly builds and smoke tests. The most important areas for achieving these goals are version control, build management, change control, and release management. The challenges and some solutions for each of these areas are described below.

> Business related decisions determine the complexity of technical product management.

6.3 Version Control

Almost all items of a software product change after their initial creation because they are typically created incrementally and refined or corrected later. Whenever an item is changed and stored a new *version* of the item is created. Various dependencies between the items necessitate the management of composite items in addition to the individual items. Typically, a well-defined set of items and their specific version is called a *configuration*. A configuration may have versions in a similar way as individual items. If the version of any item, or the set of items in a configuration changes, a new version of the configuration emerges.

In theory it is possible to manage different versions using file naming conventions and complex directory structures for files, but in practice a version control tool such as CVS is required. A version control tool allows creating and storing new versions of items, accessing older versions easily, defining and managing configurations, and also handling the problems caused by several people working simultaneously with the same items.

When deciding what to take under version control, all the items of a software product, including the development tools and 3^{rd} party components, such as a database management system, should be considered.

Tagging

The act of assigning a label to a certain version of an item is called *tagging*. By assigning the same label to all items of a configuration the whole configuration is tagged and can be referenced later. Tagging is a recommended practice for all configurations that have some important purpose, such as serving as an internal release in the end of an increment cycle or as an external release in the end of a release cycle. For example, if something goes wrong with the development of the increment, it is possible to discard changes and revert back to the configuration tagged in the end of the previous increment.

> Tagging is used to give a name to a version of an item or set of items (configuration) so that they can be referred to later.

Repository

When several developers are working with the same software items, the possibility of overwriting other people's changes or updating common items without notifying others should be avoided. The solution is to create a private workspace for everyone. Effectively a developer checks out a version from a common *repository* managed by a version control tool to his private workspace, modifies the items there, and only when he has finished some logical task, checks in the items as a new revision to the common repository. The changes become visible to others only after another developer makes a new check out from the repository. If two persons have simultaneously made changes to the same files in their private workspaces, *merging* the changed versions into a new version is required. It is also possible to deny simultaneous changes through *locking* the items in the repository during a check out.

The rules for making the check-outs and check-ins need to be defined. They actually define the implementation rhythm. A long cycle increases the effort required to integrate your own changes, because the versions in the repository have changed more. If locking is used, other developers cannot modify the locked files. A long cycle increases the time before other developers get access to the changes and decreases the visibility of the progress of development. A short cycle may cause checking in something that is not yet working well and disturbs other people's work, when they start developing on top of those items.

Sometimes a developer wants to store an incomplete version in order to revert back to it if something goes wrong. These versions should not be stored in the common repository where they may disturb other developers' work. Fortunately some IDEs and version control tools support private versions in the developer's private workspace. Another solution is to use private branches in the repository (see below).

Private workspaces integrated through a common repository allow developers to work without disturbing each other.

The frequency of integrating new code from private workspaces to the common repository defines the implementation rhythm.

Variants

If a new version of an item replaces the previous version, the new version is called a *revision*. Sometimes the new version does not replace the previous version, but presents an alternative version of the item, e.g., a source code module written for the Windows operating system is ported to Linux. The alternative versions are called *variants* of each other. In the repository a variant is called a *branch*. Each variant may have new revisions in its own branch. Variants may be *temporary variants*, e.g., for parallel development of two developers. A temporary variant is deleted later by merging it back to the originating branch as a new revision. Variants may also be *permanent variants*, e.g., when the software is tailored to a customer, or when several consecutive releases need to be maintained. Figure 6.1 illustrates the relationships between revisions and different types of variants.

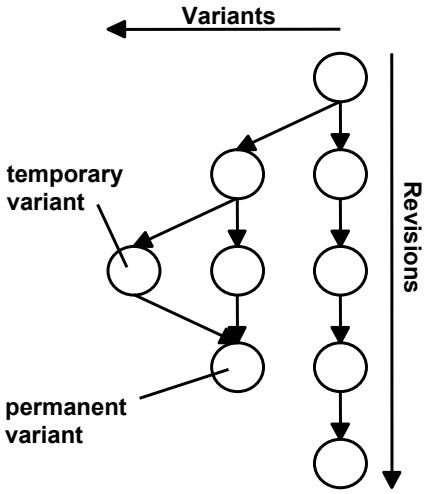


Figure 6.1: Versions, revisions and variants

When variants are created, the merging of changes between branches must be thought carefully. If an important change, e.g., a critical bug fix is made into one variant, it may be necessary to merge the change to other variants of the same item. If the change is a new feature to the new development branch, it is not necessary to merge it to the maintenance branches or previous releases.

In the source code level another way to manage variance is to embed all variants into the same file using, e.g., #if statements (in C/C++), which allow extracting the correct variant from the file before compilation. When the amount of variance is low,

this is a good solution, but as the amount of variant parts increases, the files become harder to understand.

In general, it is advisable to localise the variance to as few items as possible. For example, when developing software for different platforms, a common interface can be developed on top of platform-specific calls. Thus the variance is limited to the items implementing the interface for each platform and all other parts are similar because they can use the interface to access platform specific functions.

Managing variance in an optimal way needs careful consideration.

6.4 Change Control

Change control means having processes and practices for handling change requests related to any software item on any level of detail, such as a requirement or a piece of code. Being able to respond to change is an important characteristic of many development projects. Thus, change control must not hinder change, but on the contrary allow making changes quickly and flexibly but in a controlled way.

The purpose of change control is to avoid problems caused by a situation where anyone is able to change any software item without any communication with other involved parties. Lack of change control may lead, e.g., to lack of information on which changes have been implemented, or to sub-optimal implementation order of the change requests. The reasonable degree of control varies a lot depending on, e.g., the type of change, the target of change, the phase of the development cycle, and criticality of the software.

The term change must be interpreted in a loose way, because typically a software product is never ready. Often, when doing the initial commercial delivery, there is already a long list of ideas, more detailed requirements or even bugs based on which the next commercial releases will be developed. This means that changes are continuously implemented to the software.

Change requests may originate from several different sources. Users or sales personnel may request new and modified features and fixes to bugs found in the software. The company may internally change or refine its visions about the future of the software product. Testers and developers find bugs in the software during the development.

Generally all change requests should go through the same process. A change request should be received by quickly filtering critical bugs, “user errors” and duplicates. In a pre-defined cycle, e.g., before the next increment or release planning meeting, technically capable persons should estimate, at least roughly, the required effort and technical issues related to each new change request. Then in the next planning meeting, the new change requests are prioritised on the same scale with the existing development plans and implemented accordingly in the upcoming releases.

However, there are a couple of exceptions to the typical process. Forcing a developer to make a change request in order to fix a bug he finds during the development is typically unreasonable, but sometimes even in this case it may be justified. For example, if fixing a minor bug would take considerable effort, fixing it might mean that implementing some more important feature is delayed. Critical bugs are another ex-

ception and require different handling and probably delivering quickly an unplanned release (patch) of the product.

One special case when strict control is required is the last days before a commercial release. The development of new features should be stopped well before the release date in order to stabilise the product. During this time the product should be actively tested, found bugs fixed, and changed parts re-tested. The closer the release date, the less effort remains that will be spent on formal or informal testing of the product. Any change may introduce a new bug, and due to the late phase of the development the probability of detecting the bug is continuously decreasing. Therefore the number of late changes should be minimised and those made should be made with special care, e.g., using pair programming or code reviews.

If the stabilisation phase is long, it may be wise to make an own branch for it allowing the development of future releases to proceed in another branch. After the release the fixes made can be merged back to the originating branch, and the stabilisation branch may serve as the maintenance branch for the release made.

> Implementation of changes should be prioritised together with any other items in the backlogs.
>
> Just before a commercial release, the number of changes should be minimised, and the remaining changes made with extreme care.

6.5 Build Management

The executable version of a software product is called a *build*. The build is made from source code files using tools such as a compiler and linker. Builds are made for several different purposes. A developer needs a build in order to test how his new code works with the rest of the system, testers need builds to test the product (in parallel with the development and before a release), and a build is also the most crucial part of a software release.

Because making a build often includes several tools and steps, it is practically always automated by writing a command line script or using tools such as make or Ant. Automation allows making builds frequently and also adding additional steps, e.g., to verify the build's quality, generate reports of the build or to tag the configuration forming the build. A typical practice is to make the build automatically every night and then run some automated tests to check the build's stability. Tests can be created using, e.g., unit test frameworks such as JUnit.

If the build process can be optimised to last only a couple of minutes, builds and tests can and should be run every time a developer is checking in new code. Preferably the developer should integrate the latest code from the repository with his new code in his own workspace, build and test the system there, and only then check his new code into the repository. This practice minimises the possibility of disturbing other people's work with broken code.

> Builds are used for several different purposes during development.
>
> The build process should be automated in order to make the builds more quickly and in a more reliable way.

6.6 Release Management

When using IID, releases are made often and the importance of making them reliably and efficiently is emphasised. Just finishing and testing the code is not enough for making a release. A software release contains a whole lot of documentation (user's manual, installation guide, etc.), which must be produced and "tested" in parallel with other development work. In addition, installation of the software by the user may require creating installation scripts or programs.

Users' different environments are a challenge, because in practice it is impossible to test any product in all possible configurations of hardware and software, where the product is supposed to work. Testing effort should concentrate on the most typical environments, and the environment-dependent parts should be localised in the code already when designing the architecture of the software.

With less productised software, delivering the software to a customer may require making the installation at the premises of the customer and integrating the software with the customer's systems. Making checklists of typical installation steps and problems, and running test suites with the customer installations are good practices to decrease problems with the deliveries and thus decrease the required effort and increase the customer's satisfaction.

> When using IID, releases are made often. Therefore release related activities such as testing on various platforms, finishing documentation, and system installation should not require too much effort.

References

Bays, M. E. 1999. *Software Release Methodology*. Upper Saddle River: Prentice Hall.

This book discusses technical product management from the perspective of making software releases.

Berczuk, S. and B. Appleton. 2002. *Software Configuration Management Patterns*. Boston: Addison-Wesley.

This book describes twenty SCM patterns related to the daily work of software developers.

Fowler, M. Continuous Integration. [electronic article]. [cited April 13^{th}, 2004]. `http://www.martinfowler.com/articles/continuousIntegration.html`.

This article describes the advantages of integrating the work of different developers often, and how to create an automated building and testing environment.

Tool Support

Mikko Rusama

Introduction

In this chapter we discuss tool support in CoC deployment. We define *tool* in the meaning of *CASE tool*:

> A software tool that aids in software engineering activities, including, but not limited to, requirements analysis and tracing, software design, code production, testing, document generation, quality assurance, configuration management, and project management (IEEE 1995).

The number of different tools is enormous and there are many web sites and articles providing long lists of tools for different purposes. Thus, this chapter focuses on challenges in deploying the CoC framework and describes approaches for tackling these. The CoC framework does not dictate the usage of any particular tool type, but in practice several different tools are used in the companies which have deployed the framework. Thus, based on their experiences, we are able to give some examples of tools that have been found to be useful in the CoC context.

The Role of Tools

Typically, a tool has two different roles that are not mutually exclusive:

1. Vehicle for change — A tool works as an enabler of change, and helps in the deployment of new processes, practices, and methods in the organisation.
2. Process support — A tool supports the new or existing processes, practices, and methods

Software engineers are very often tool-oriented; they love different tools — especially development tools — and are eager to adopt new ones. The approach for choosing the right tool should, however, be solution-oriented, as illustrated in the example approach below[1]:

First, identify your problems. In many cases, a poorly defined process creates most of the problems and adopting new tools to the existing mess will only create new problems. "A fool with a tool is still a fool" — only bit more dangerous.

> Manual functional tests are often labour intensive, time consuming, inconsistent, boring, and lengthy, and comparison of test results is tedious and error prone.

Explore alternative solutions, not just tool-based solutions.

[1] This example is adopted from the book *Software Test Automation* by Fewster and Graham (1999).

> If manual tests are too lengthy, weed out ineffective and redundant tests. If testing takes too much time, recruit more testers. If tests are too labour intensive, redesign tests to require less effort per test e.g. by reorganising testing facilities.

A tool may be *one potential solution* to the problem. Tools as such should have no inherent value. In many cases, pen and paper might be just enough. In the words of Mushashi (a famous Japanese swordsman), "Do not develop an attachment to any one weapon or any one school of fighting". It means that you should not fall in love with a particular tool or process model; instead, look for the best possible solutions that have high-impact and low cost.

> In our example, part of the testing process may be automated with tools. For example, HttpUnit is a potential choice for creating automated functional tests for Web applications.

To really understand the benefits of tools, you should set measurable goals and success criteria for the tool. Formal measurement of benefits, e.g. calculating the ROI (Return on Investment), is difficult and rarely done.

Characteristics and Benefits of Tools

Tools vary in scope from supporting individual tasks to encompassing the complete development life cycle. Thus, tools can be divided into vertical and horizontal tools:

- *Vertical tools* support specific activities on a CoC time horizon, e.g., at the heartbeat level. Examples include modelling tools.
- *Horizontal tools* support activities across different CoC time horizons (release, increment, and heartbeat) and are typically used by many different stakeholders in the organisation. Examples include version control and requirements management tools.

Tools are typically designed to support a particular method, i.e., an orderly repeatable set of actions for accomplishing a well-specified task. Thus, tools can make work more systematic; they can improve consistency and uniformity of the development approach.

Tools allow repetitive, well-defined actions (methods) to be automated, giving developers more time on the creative aspects of the process. In many cases, automation improves the quality of work compared to manual work, and improves productivity.

Tools enable process scaling; if the development team grows bigger, some tool support is typically required. Tools also provide support for complicated or large tasks, such as compilation or synchronisation of parallel work (e.g., using version control).

Most tools store data in a digital format that is easier to manage, search, and copy than hand-taken notes.

Choosing the right tools — especially the right horizontal tools — is one of the most challenging and important issues in CoC deployment. Tools provide clear benefits in CoC implementation but the process should be first well understood.

Problems and Issues in SMEs Related to Tools

In many cases, the starting point for CoC deployment is that a company's processes are not clear or even undefined (ad hoc). It follows that there is no well-specified and standardised tool set in place. Typical tools-related problems are:

- *Finding the right tool*. The difficulty is to find the right tools to support the CoC implementation. Companies and their characteristics differ and there are plenty of tools available for the same purpose, making the selection difficult. In many cases, small companies do not have a process for evaluating and selecting tools (in addition that the whole development process is more or less ad hoc), and may end up acquiring useless tools (shelfware).
- *The tool does not solve the problem* as expected but may set unwanted constraints. In many cases, the process needs to be adapted and guidelines are needed for using the tool.
- *Tools are unnecessarily complex* — in many cases only a small subset of the available features is used and tools are used inefficiently. In some cases, developers lack a feature and look for a new tool without knowing that the old tool has the feature already implemented.
- *Platform incompatibilities.* Many companies have different platforms: Linux, Mac or Windows that they work with. Different platforms cause problems for tool usage; even programs written in Java — a platform independent programming language — do not necessarily work the same way on every platform.
- *Interoperability* issues are sometimes very difficult to solve; tools simply do not work together. Data formats are incompatible and there is no common user interface.
- *Tool heterogeneity.* Many different tools are used for the same purpose. For example, if two developers are familiar with different IDEs it is much harder to do pair programming efficiently. Problems may also arise from lack of interoperability between different tools, e.g., modelling tools use different file formats.
- *Tools are buggy*. Development platform and development tool bugs are especially difficult to work around.

Typically, both personnel and financial resources of an SME are scarce. SMEs have little time to learn new tools or technologies. Thus, tools should be simple to learn and use. The pure purchase price of the tool may be too high. It should also be remembered that the total cost for the tool also includes adoption costs related to training, customisation work, acquiring new hardware, and integration.

As a clear trend, there are more and more free open-source tools and Linux has become a viable alternative for Windows as a development platform. Nowadays, there are companies that mostly use free tools. However, in most cases, the adoption costs are much greater than the purchase price of a commercial tool. In the words of an IT expert, “all tools are free” — compared to the adoption costs.

Despite the problems, it seems that tools have not been a bottleneck in CoC deployment. The primary difficulty is to understand and adapt the CoC framework to the company’s needs. The selection of the right tools to support the process has been a secondary concern. However, it is important to understand the problems and issues described above for tool deployment in order to avoid common pitfalls.

Tools for CoC Deployment

In this section, we describe tool-related problems in CoC deployment and give examples of different approaches for tackling these problems. The CoC framework does not dictate the usage of any particular tool type but in practice, several different tools are used in the companies that have deployed the framework. We focus on the following areas and related tools that we consider important in CoC deployment:

- Requirements management
- Progress tracking
- Testing
- Communication

All tools for these areas (except testing) are horizontal, i.e., used by many different stakeholders of the company across the time horizons of the CoC framework. Before discussing the different tools we shortly review the characteristics of the CoC framework.

CoC Characteristics

CoC is a time boxed, iterative and incremental development process as described in Chapter 1. The goal is to maintain a constant rhythm with recurring control points. In its simplicity, CoC can be characterised as a process having two states: “Doing” and “Control point”.

In the “Doing” state stakeholders (e.g., developers) concentrate on the work at hand within the allocated time box. During “doing” the chosen set of features is frozen: if changes are required, the next control point should be waited for. The goal of each cycle (increment) should be reached even though the scope is flexible.

In the “Control point” both business and technical management can influence the direction of development. Frequent control points are part of risk management and the planning of the next “doing“ time box. In the “Control point” the goal for the next time box should be set and features should be prioritised.

The challenge of deploying CoC can be expressed as a single question “How to maintain the development rhythm?” To further clarify this, the following questions should be asked:

- How do we manage changing requirements?
- How do we deliver the right product at the right time?
- How do we build and maintain good enough product quality?

The CoC framework provides an approach for handling these challenges. In addition, several different types of tools can provide support for the various activities.

The following chapters describe some problems in the deployment of the CoC framework and give examples of different approaches for tackling these problems. We also give some examples of tools that can be used.

Requirements Management

Requirements management is a process that most companies find particularly puzzling. Typically, some kind of tool support is needed. One of the most common approaches when starting to do systematic requirements management is to use a simple spreadsheet. Perhaps the most widely used tool for this purpose is Excel.

In agile development, one of the main challenges of requirements management stems from the agile principle “Embrace Change”:

> Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.

The challenge is to handle the requirements pipeline from ideas to product releases. New requirements are continuously pouring in, requiring prioritisation and reprioritisation at the control points.

When using the CoC framework, we propose that *backlogs* (Schwaber and Beedle 2002) are used for requirements management. A backlog is a prioritised list of items such as ideas, features (functional and non-functional), change requests, or defect reports. These requirements are collected from several different sources, including customers, sales and marketing people, competitors, etc.

Backlogs are divided into three different types as depicted in Figure 1.

- *Product backlog* — Collects all product ideas. Can also include defect reports.
- *Release backlog* — The management team specifies, prioritises and allocates product backlog items into the product roadmap as different releases.
- *Sprint backlog* — Release backlog items are further broken down into tasks, prioritised, and allocated into the sprint backlog in sprint planning.

Typically, each backlog item has several properties such as:

- Description
- Priority
- Effort estimate
- Realised hours
- Responsible person (typically not used at the product backlog level)
- Status — status of the item, typically there are only two states: *open* or *closed*.

In addition, each item has a life cycle with transitions from the product backlog to the sprint backlog, as illustrated in Figure 1. The status of the item is also related to the life cycle; initially, the item is in the *open* state, and after it has been completed, it is in the *closed* or *completed* state (in the sprint backlog). There should not be too many states as it complicates the management of items.

The project manager is responsible for maintaining the sprint backlog. Typically, before the heartbeat control point, developers are responsible for recording 1) the effort spent on completing tasks and 2) estimates on how much effort is needed to complete the remaining items.

Excel as a Tool for Backlog Management

Technically, all backlogs can be maintained in one or more Excel files. Despite its limitations, Excel is a simple and effective tool for requirements management if the Excel template, perhaps including built-in macros, is well-designed. The problems of using Excel backlogs include:

- *Life cycle management of items is difficult.* As the scope of the release is flexible, the features to be implemented are moved from one (release or sprint) backlog to another. New tasks are spawn from original tasks making it difficult to track the life cycle of requirements.
- *Audit trail.* It is difficult to track what tasks were actually completed to implement a particular feature or to track what features were completed in a specific increment.
- *Reporting is difficult.* Excel is a very flexible tool, which enables the generation of almost any kinds of reports and graphs. However, automating report generation may be both difficult and laborious.

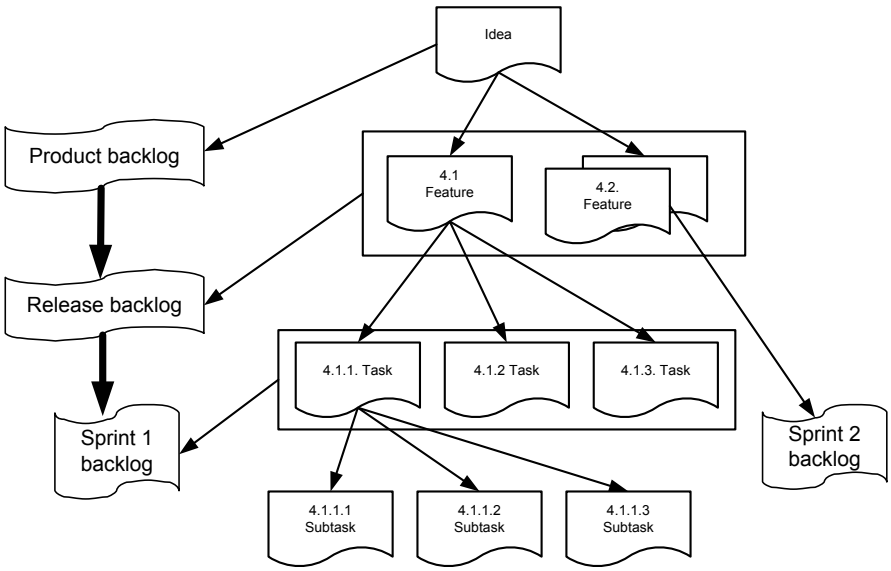


Figure 1: Backlog-based approach for managing requirements

- *Interoperability is poor.* Excel files do not transfer well, e.g., to the defect management system. Bugs in the defect management system (e.g., Bugzilla) generate new requirements and tasks that need to be tracked in the Excel backlog as specified above.
- *Concurrent editing is difficult.* When working in teams, it is difficult to share a single Excel file for updating the realised effort or marking tasks as completed. This problem is not Excel specific and can be partly resolved by using a version control system.

Backlogs and Defect Tracking

Many companies have a separate defect management system, in addition to their requirements management tool(s). For example, Bugzilla is used as a defect tracking system, but it integrates poorly with Excel backlogs. Defect fixing requires resources and the required effort should be indicated in the sprint backlog.

A practical solution is that during each sprint, some time is dedicated for fixing the bugs found in the previous sprint. There might be a generic “fix bugs” task in the sprint backlog, which is refined as bug fixing starts. Only larger, laborious, or critical defects generate separate tasks in the sprint backlog as they may jeopardise the achievement of the sprint goal.

Full-Fledged Backlog Management Systems

The next step from an Excel-based requirements management system is long. There are many tools for project, requirements and defect management. Examples of such tools that were used in the companies that utilise the CoC framework include:

- **Jira** (`http://www.atlassian.com/software/jira/`) is used for tracking and managing the issues and bugs that emerge during a project.
- **Devtrack** (at: `http://www.techexcel.com/Products/DevTrack/DTOverview.html`) is a project- and defect-tracking tool for product development organisations. It tracks and manages all product defects, change requests, and all other development issues.
- **Starteam** (at `http://www.borland.com/starteam/`) is a configuration and change management system with support for discussion groups. All items can be easily linked with each other.

There are many requirements for a good *requirements management system*. To name few, the requirements management system should be highly configurable supporting different issues (ideas, features, defects, suggestions etc.) and their potentially different life-cycles. Items should be easy to link with each other. Generating different reports for customer billing or progress tracking should be as easy as possible. In addition, the system should be easily extendable with well-defined external interfaces.

Progress Tracking

According to one of the agile principles:

Working software is the primary measure of progress.

The CoC framework supports effective monitoring of development progress due to the use of small increments and frequent deliveries. Along with following the progress of individual tasks, you should monitor the goals of the increment and release project. Typically, at least two questions should be asked when evaluating whether the goal of the increment has been reached:

- Are the customer requirements sufficiently fulfilled (functional and non-functional)?
- Is the quality good enough?

Heartbeat meetings and increment demonstrations are approaches to monitoring the progress of development. A heartbeat meeting is a short meeting lasting no more than 15 minutes. In the meeting the following three questions are asked:

- What have you done since the last meeting?
- What problems did you have? Do you have any obstacles?
- What are you going to do before the next meeting?

Typically, before or during the meeting, some data is recorded (e.g., to the sprint backlog):

- The features/tasks completed since the last meeting
- Hour reporting data
  - Estimated hours — time estimates for remaining and new tasks are updated. At the beginning of the increment, each team member estimates the number of hours it will take to complete each of the tasks they have been assigned for the next time box (increment).
  - Realised hours are assigned to projects, features, tasks

Each sprint backlog item should specify the amount of work remaining. Effort estimation during an increment focuses on updates to the estimated number of hours needed to complete a task. This should not be confused with time reporting. These values are updated continuously, typically in the heartbeat meetings. From this data, a burn

down graph is easily visualised (e.g. with Excel), showing the project progress (for an example, see Figure 4.1).

Another approach to monitoring the progress of the project is to follow the bug trends. If the number of fixed bugs is increasing and the number of new or open bugs is decreasing — or the gap between the open and closed bugs is closing, this can be an indication that the product is maturing, i.e., its quality is improving. Another explanation is that testing has been done poorly. Most defect tracking systems provide graphical views of bug trends automatically.

Testing

As previously discussed, the iterative, incremental and time boxed process used by the CoC framework creates challenges for testing. There is no long dedicated time for testing as in waterfall type processes. Instead, testing and quality assurance are continuous activities that face the challenge of frequent deliveries. As the software needs to be delivered often, it is important to effectively be able to guarantee "good enough" quality. We thus need to define "good enough" testing and "good enough" quality. According to James Bach (1997):

> "Good Enough testing is the process of developing a sufficient assessment of quality, at a reasonable cost, to enable wise and timely decisions to be made concerning the product."

Small companies do not necessarily even have a dedicated testing team. The challenge is to integrate quality assurance into the development process at a reasonable cost. Testing should also be automated as much as it is possible and reasonable.

Chapter 5 describes different approaches to testing. To shortly summarise, the following issues should be tackled at each time horizon:

- What tests should be designed and documented?
  - E.g., manual tests, automated tests, checklists of test ideas
- What tests should be executed and reported?
- What other quality assurance practices should be performed?
  - E.g., reviews, coding conventions

In addition, the following issues should be tackled:

- What is the goal and purpose of quality assurance in this time horizon?
- Who is responsible for performing all the defined actions (described above)?
- When and how often are these actions made?
- What tools can be used?

The following example clarifies the ideology of time horizons in the quality assurance:

> At the heartbeat level (heartbeat time horizon) developers write unit tests using JUnit (`http://www.junit.org`) before the actual code and start planning (for the increment time horizon) how to measure the performance of the product using JMeter (`http://jakarta.apache.org/jmeter/`). The coding conventions are also followed. Before the end of the increment (increment time horizon), some time is dedicated for running JMeter performance (load) tests. The release time horizon is considered designing automated functional tests using HttpUnit (`http://httpunit.sourceforge.net/`) tests that are integrated into the build and deployment processes.

Communication Tools

According to an agile principle:

> The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

In practice, arranging face-to-face meetings is not always possible and there needs to be a way to communicate and exchange documents efficiently. The challenge is to find the right communication channels and make the people use them. If there are too many communication channels, the monitoring overhead easily becomes too high. Also, some forms of communication — especially online chat — easily become pure entertainment that might be valuable as such but does not serve the purpose of supporting the daily work.

The SEMS approach as such does not dictate the usage of any communication tool. In addition to email, typical communication tools are:

- *Discussion groups* (for example newsgroups) are useful for documenting ideas, questions and answers. To view and post messages to a newsgroup, you need a news reader, a program that runs on your computer and connects you to a news server on the Internet.
  - Preliminary product ideas can be elaborated in an internal newsgroup from which they can be moved to a product backlog (Excel) and then further specified and prioritised.
- *Instant Messenger* is a type of communications service that enables creation of a kind of a private chat room with another individual in order to communicate in real time over the Internet, using text-based communication. Typically, the instant messaging system alerts you whenever somebody on your private list is online. You can then initiate a chat session with that particular individual.
- *Chat systems* enable real-time, text-based communication between two or more users via computers. Once a chat has been initiated, either user can enter text by typing on the keyboard and the entered text will appear on the other user's monitor.
  - Internet Relay Chat (IRC) is a multi-user chat system, where people meet on "channels" (rooms, virtual places, usually with a certain topic of conversation) to talk in groups, or privately. To join an IRC discussion, you need an IRC client and Internet access. The IRC client sends and receives messages to and from an IRC server. The IRC server is responsible for broadcasting messages to everyone participating in a discussion.
- *Wiki* is a collaborative Web site comprised of the collective work of many authors. Wiki allows anyone to easily edit, delete or modify content that has been placed on the Web site using a browser interface, including the work of previous authors. The term Wiki refers to either the Web site or the software used to create the site. There are many open-source Wiki softwares freely available on the Internet.

Others

There are also several other tool types that could also be covered, including development, configuration management, and documentation tools.

Development tools are necessary in software development — without a compiler life would be rather difficult (unless you write interpreted code). Development tools

differ greatly depending on the technological choices (programming language, operating system, product architecture etc.) of the company.

Documentation tools and especially lightweight documentation tools are especially important in the SEMS approach as heavy documentation is avoided. A special group of lightweight documentation tools are tools that semi-automatically generate documentation from the source code:

- **Javadoc** (`http://java.sun.com/j2se/javadoc/`) is a tool for generating documentation in HTML format from comments in source code.
- **Doxygen** (`http://www.doxygen.org`) is a documentation system for C++, C, Java, and many other programming languages.

Chapter 6 describes challenges of technical product management including version control. The most common tools for configuration management including version control and the build process are:

- **Concurrent Versions System (CVS)** (`http://www.cvshome.org/`) is the dominant open-source network-transparent version control system.
  - *TortoiseCVS* (`http://www.tortoisecvs.org/`) is a CVS client that lets you work with files under CVS version control directly from Windows Explorer.
  - *ViewCVS* `http://viewcvs.sourceforge.net/` can browse directories, change logs, and specific revisions of files. It can display diffs between versions and show selections of files based on tags or branches.
- **Ant** (`http://ant.apache.org/`) is a widely acknowledged and used Java-based build tool.
- **Make** is one of the most popular build tools, especially for C/C++ development. Make is included in the basic Linux distribution.

Tools in CoC Deployment: Case HardSoft Ltd.

This chapter illustrates an example CoC implementation in an imaginary software product company HardSoft. HardSoft has 20 employees, and specialises in Web-based time reporting and progress tracking software. Most product installations are based on a standard product configuration that is easy and fast to deploy. Approximately 1/5 of the customers require special customisations for the product. For this purpose, typically a one-month deployment project is set up. Seven developers work full-time on the core product (platform) and approximately three developers are assigned to customer deployment projects at any single point in time. The developers rotate in their roles, so all developers have participated in deployment projects.

All ten developers use the Java programming language. Most of the tools they use are open-source.

The following example mainly illustrates the work of HardSoft's core product development team. Only parts of the company's other processes and tools are described.

Requirements Management

HardSoft has an internal NNTP server hosting an internal newsgroup "product ideas" that is used as an idea pool for elaborating, discussing and documenting all ideas — also the crazy ones — related to new product releases. Ideas are reviewed once a month (at the control point before the next sprint) and suitable ideas are collected

to the product backlog. The product backlog is an Excel file in which the ideas are further specified and prioritised.

Prior to a release, management allocates preliminary resources for the release project and decides when it should start. Preliminary scheduling and resourcing is documented in the release backlog, which is an Excel file.

To create a release backlog, an Excel macro copies the selected (release) items from the product backlog to the release backlog. In the release backlog, items are then further allocated to increments (sprints) and features are split to tasks that are specified, prioritised and assigned to developers. Release and sprint backlogs are physically in the same Excel file. Each sprint has a dedicated "Excel tab", a worksheet under which all the tasks of the sprint are managed.

Developers are responsible for filling in the *actual hours* spent on the specified tasks and update their *effort estimates* for the remaining tasks in the sprint backlog before the Friday morning status check. The project manager has the ultimate responsibility for the release and sprint backlogs. Typical tasks of the project manager include:

- Adding new tasks to the backlogs
- Moving tasks from one sprint backlog to another
- Splitting original tasks into smaller ones
- Prioritisation and reprioritisation of tasks (in cooperation with Business)

Progress Tracking

Progress tracking is based on daily 10-minute scrum meetings where developers answer three questions:

- What did you do since the last meeting?
- What problems did you have? Do you have any obstacles?
- What are you going to do before the next meeting?

Problems are not solved in the meeting but discussed after the meeting with a smaller team — typically only two developers.

Once a week, on Fridays at 10 am, HardSoft has a longer, up to 2 hours long, status meeting in which project progress and all completed tasks are more closely discussed. Before the meeting, developers are forced to fill in the *actual hours* spent on the specified tasks and update their *effort estimates* for the remaining tasks to the sprint backlog. The sprint backlog, an Excel file, is under version control, and developers are asked to lock the file — not for longer than 20 minutes — when updating the required information. This should be done before 9 am on Friday. In the Friday morning meeting, all completed tasks are reviewed by the project team. In the status meeting, the following data is reviewed:

- The number of open, closed and red-flag tasks
  - Red-flags are unexpected tasks that have taken time from the original development tasks. A typical red-flag is caused by a critical bug in a customised customer installation that needs an urgent fix.
- The proportion (%) of red-flags to original development tasks
- The *burn down graph* visualising estimated hours to complete all the specified tasks. This is the only automatically generated graph in the Excel backlog.

- *The bug trend*, the distribution of open, new, and fixed bugs is viewed in Bugzilla, an open-source defect tracking system.

Based on this data, the project manager is able to estimate whether the goal of the increment can be reached, and is able to inform management about the progress of the project.

At the end of each increment (sprint), a sprint demo is arranged, in which both Business and Development participate and a working product increment is demonstrated. At the end of each release a similar demonstration is arranged but selected customers are also invited.

Development and Quality Assurance Tools

Developers use the Eclipse open source IDE that has a built-in support for JUnit (a unit and regression testing framework), CVS integration, and the Ant build tool. Developers use test driven development: JUnit tests are written before the actual code. Also, coding conventions (Sun's Java coding conventions) are enforced, e.g., all source code is documented using Javadoc. Eclipse has a *Jalopy* plugin that is able to format the code according to these agreed upon coding conventions.

The code is checked-in in the CVS version control system daily, and nightly builds are made using Ant and a cron scheduling script. The working version of the product is automatically installed on the internal test server and some automated functional tests are run using HttpUnit.

At the end of the release cycle, during the stabilising increment, most developers focus on testing, as there is no separate testing team. Most of the testing is done manually using predefined test cases. Test cases are specified in a Word file. From this Word file, test cases are automatically collected (using an Excel macro) into an Excel test log. Results of the tests are documented to the Excel test log. If a bug is found it is reported to the Bugzilla defect tracking system and Bugzilla's bug id is added to the Excel test log. If the manual test passes, the tester writes his initials in the test log. Regression testing is made with JUnit and HttpUnit. JMeter and Grinder are used for performance testing.

In addition to manual and dynamic performance testing, at the end of each sprint, CCCC is used for static code analysis. The following metrics are reviewed:

- Number of files (Java)
- Number of source code lines
- Number of comment lines
- McCabe's cyclomatic complexity
- Number of methods

The number of code and comment lines is compared to the results of the previous sprint to get some kind of understanding of the coding effort. The need for code refactoring is evaluated based on the CCCC reports and realised in the next increment if necessary.

Further Improvements

HardSoft is not completely happy with the Excel backlog approach and is looking for a more flexible Web-based tool for managing backlogs. Interoperability of Excel backlogs and the Bugzilla defect tracking system has proved to be clumsy. HardSoft

has identified two potential candidates for replacing Excel backlogs and Bugzilla: Jira and Devtrack are integrated issue and defect tracking systems.

Useful Tools

This chapter shortly lists all the tools mentioned in above, including some additional commonly used tools.

Requirements Management and Defect Tracking Tools

- Excel (`http://www.microsoft.com/excel/`) spreadsheets with macros are used for managing backlogs.
- Bugzilla (`http://www.bugzilla.org/`) is an open-source defect tracking system.
- Jira (`http://www.atlassian.com/software/jira`/) is used for tracking and managing the issues.
- Devtrack (see `http://www.techexcel.com/Products/DevTrack/DTOverview.html`) is a project- and defect-tracking tool for product development organisations. It tracks and manages all product defects, change requests, and all other development issues.

Development Tools

- Eclipse (`http://www.eclipse.org/`) an open and extensible Integrated Development Environment (IDE) especially suitable for Java development.
  - Jalopy (`http://jalopy.sourceforge.net/`) is a source code formatter for the Sun Java programming language. Eclipse plugin is available too.
  - TeamInABox (see `http://www.teaminabox.co.uk/downloads/metrics/`) is a metrics plugin for Eclipse.
- Javadoc (`http://java.sun.com/j2se/javadoc/`) is a tool for generating documentation in HTML format from comments in the source code.
- Doxygen (`http://www.doxygen.org`) is a documentation system for C++, C, Java, and many other programming languages.

Testing Tools

- Word (`http://www.microsoft.com/word/`) is used for documenting manual test cases.
- Excel (`http://www.microsoft.com/excel/`) is used for managing test logs.
- JUnit (`http://www.junit.org`) is one of the most popular Java-based unit testing tools suitable for automated regression testing.
- HttpUnit (`http://httpunit.sourceforge.net/`) is a testing tool for Web applications. Written in Java, HttpUnit emulates the relevant portions of browser behaviour, including form submission, JavaScript, basic http authentication, cookies and automatic page redirection, and allows Java test code to examine returned pages either as text, an XML DOM, or containers of forms, tables, and links.

- Grinder (`http://grinder.sourceforge.net/`) is a Java load-testing framework.
- JMeter (`http://jakarta.apache.org/jmeter/`) is a pure Java desktop application designed to load test functional behaviour and measure performance.
- CCCC (`http://sourceforge.net/projects/cccc`) is a tool, which analyses C++ and Java files and generates a report on various metrics of the code.

Configuration Management and Build Tools

- CVS, Concurrent Versions System (`http://www.cvshome.org/`) is the dominant open-source network-transparent version control system.
  - TortoiseCVS (`http://www.tortoisecvs.org/`) is a CVS client that lets you work with files under CVS version control directly from Windows Explorer.
  - ViewCVS `http://viewcvs.sourceforge.net/` can browse directories, change logs, and specific revisions of files. It can display diffs between versions and show selections of files based on tags or branches.
- Ant (`http://ant.apache.org/`) is a widely acknowledged and used Java-based build tool.
- Make is one of the most popular build tools, especially for C/C++ development. Make is included in the basic Linux distribution.

Communication Tools

- ICQ (`http://web.icq.com/`) is an online instant messaging program.
- MSN Messenger (`http://messenger.msn.com/`) is Microsoft's popular instant messenger.
- MIRC (`http://www.mirc.com/`) is a popular IRC client program.
- *Wiki* allows anyone to easily edit, delete or modify content that has been placed on the Web site using a browser interface, including the work of previous authors. A number of Wiki engines can be found at `http://c2.com/cgi/wiki?WikiEngines`.

References

Bach, J. 1997. A Framework for Good Enough Testing. *Computer* 31 (10): 124-26.

Fewster, M. and D. Graham. 1999. *Software Test Automation : Effective Use of Test Execution Tools*. Boston: Addison-Wesley.

IEEE. 1995. *IEEE Std. 1348-1995, Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools*. New York: IEEE.

Schwaber, K. and M. Beedle. 2002. *Agile Software Development with Scrum*. Upper Saddle River: Prentice Hall.

Software Process Improvement Basics

Kristian Rautiainen

Introduction

The purpose of this chapter is to present and discuss software process improvement (SPI) basics. When you start improving your process(es), there are certain things you need to know and understand in order for you to have a better chance of succeeding. In theory, process improvement is about the ways of working, not the people doing the work, but in reality it is very much about the people and getting them motivated. This chapter covers some of the basic principles of SPI, presents a process framework that can be used when planning SPI, and discusses the motivators and de-motivators for SPI from different practitioner viewpoints.

Principles for SPI

SPI research has revealed basic principles that seem to apply to almost any SPI initiative. The following sections present these principles and discuss the application of them.

Management Commitment

A key success factor for a SPI initiative is management commitment. If top management is not 110 % committed to the SPI initiative, their actions easily hinder all efforts by others. By 110 % commitment we mean that just deciding that an SPI initiative is something that the organisation should undertake is not enough. Management must also participate in every possible way in the initiative itself, e.g., by planning their own management processes simultaneously with the development processes. We have experienced that otherwise the organisation's processes do not necessarily work, as the following example illustrates.

> At HardSoft Ltd the developers were fed up with the way things worked. Almost every day Joanna (Sales Director for Gadget) called Jill (the Development Team Leader) and told her about the latest changes to the release plans, because she had sold a new feature to a customer. The developers never got anything ready by the release deadline, and planning was becoming more or less impossible. Jill had just had lunch with Jeeves (a Process Consultant) that she knew from when she studied at the University, and he had told her about the SEMS approach and the CoC framework that were supposed link business and development in a controlled but yet flexible fashion. Jill got really exited and she convinced Jeff (the CEO of HardSoft) that HardSoft needed to improve their processes using the

SEMS approach. A meeting was arranged between Jeff, Jill, and Jeeves to discuss it further, and after the meeting Jeff agreed that this definitely was the thing to do. "I support you 100 %, go ahead with the SPI initiative", said Jeff and the meeting ended.

Jill was much exited and so were the developers as well when she told them. During the next month they all participated in specifying the processes for HardSoft's product development supported and coached by Jeeves. Some design and implementation practices were adopted from eXtreme Programming for the heartbeat time horizon activities, the increment length was specified to one month, and a strict requirements and resource freeze for the increment time horizon was decided and agreed with upper management. The control points were also specified for each cycle of control, including the roles and responsibilities of different stakeholders. A couple of training sessions were arranged to communicate and train the new way of working to all, including upper management.

The following month the new process was piloted in one of the release projects. To Jill's disappointment, Joanna still kept calling her almost daily, insisting on changing the increment plans, even though she had agreed to the new process with requirements freeze for the increments and she had participated in the increment planning meetings. Jill complained to Jeff, who was very sympathetic and promised her his support. However, during the following 6 months the situation did not get much better, not until John (the Visionary and vice CEO) was appointed as the head of the newly formed Product Management Board (PMB). The PMB was responsible for all the products of the company, including that the releases were successful and reflected the strategic ambitions of HardSoft. John used a couple of weeks to define the management processes for the PMB, and ended up with exactly the same processes as defined by Jill, the developers, and Jeeves, with the exception of some refinements and choices of words that he used from a management perspective. When these processes were applied in the following increment, things started to work out, improving in each subsequent increment, as the organisation learned and adapted its processes. Jill and the developers were finally satisfied.

Practitioner Buy-In

Although management commitment is crucial, because of the resourcing issues involved, practitioner buy-in is at least as important. If the developers are not motivated to change their current way of working, they resist change causing inertia to the SPI initiative. Motivational issues are discussed in more detail in Section 6.6.

Improvement Requires Investment

Any improvement or change is going to cost you, both directly and indirectly. The direct costs stem from the effort used to, e.g., planning, training, and documenting the improvement. Indirect costs stem from the performance downfall that is typical to any improvement or change. When the people learn new ways of working, their performance is at first decreased, but when they learn the new (and hopefully) better way of working, the rewards in performance increases should cover the costs. There is no guarantee, however, that this will happen, so there are always risks involved in improvement. But in our opinion, the risks of not improving far outweigh the risks of improving in most cases. One way to minimise the risks is to pilot the improvements in a smaller scale, e.g., only one development team tries out a new practice like pair

programming. When the piloting proves successful, the new practice can be rolled out to the whole development organisation.

First Understand the Current Process

A common mistake in SPI is to specify an ideal process for the organisation without considering the existing process. This way the ideal process easily is exactly that, an ideal that can never be reached and that discourages the practitioners from even trying to reach it, because it seems so far away from the current process. The key is to start by characterising the current process and identifying needs for improvement. A good framework for SPI that includes this idea is the Quality Improvement Paradigm (QIP), originally developed by Basili, Caldiera, and Rombach (1994) and shown in Figure 2.

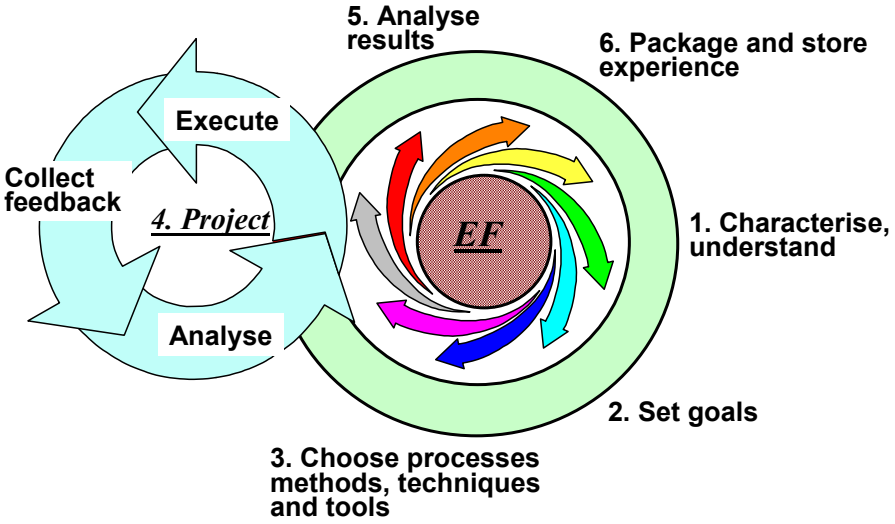


Figure 2: The QIP framework

The steps of the QIP seen in Figure 2 can be translated for SPI as follows:

1. *Characterise and understand* the current process(es) based upon available models, data, intuition, etc. Create a baseline for improvement using this characterisation.
2. *Set goals* for the improvement initiative based on the characterisation and understanding you have formed in step 1. Remember to consider what has strategical relevance to your organisation. The goals should be quantified (i.e. made measurable), so that you can assess the success of the SPI initiative based on how well the goals are reached. Do not set the goals too high or they might discourage your personnel. The idea is to improve the processes incrementally, so you should explicate short-term goals. A good idea is to include a long-term vision to help you in setting the short-term goals.
3. *Choose processes, methods, techniques and tools* for improvement on the basis of the characterisation and your set goals. It is advisable to choose smaller parts for improvement instead of trying to improve everything at once.
4. *Execute the project* with the chosen improved processes, *collect feedback,* and *analyse* the feedback after each increment to make further improvements.

5. *Analyse the results* of the improvement efforts when the project is finished. Use the data to evaluate the success of current practices, determine problems, document the findings, and make recommendations for further improvement.

6. *Package and store experiences* in the form of new or updated models, checklists, instructions or other forms of structured information. The QIP suggests using an Experience Factory (EF) repository for this, but you can use whatever way of storing the information you are comfortable with. The main thing is that the information is available to all stakeholders and that they are aware of its existence. Therefore a short training session for the improved parts of the process might be in order after each release project.

After this the QIP cycle starts from the beginning, with new areas of improvement.

Change Must Become a Way of Life

If processes are not constantly improved, they die or decay. That is why change and SPI must become a way of life. The QIP presented above is good also in this respect, since it promotes cyclical, continuous SPI. A good way to complement the QIP is to have regular reflection meetings, where different stakeholders gather to discuss what has been working and what has not. This way, new ideas for potential improvement can be gathered and everyone is kept involved with SPI, which also helps in creating better buy-in.

Institutionalising Improvement Requires Periodic Enforcement

The principle of periodically enforcing the improved process to make it part of the organisational culture, i.e., institutionalising it, is very close to the principle of change becoming a way of life. The main difference is that institutionalisation is needed to make the changes more permanent, and too much change can prevent this. The best way to enforce the processes is peer pressure. Everybody is responsible for pointing out lapses in process conformance, and if there is a need to change the process, it should be addressed at the reflection meetings. As with freezing the requirements for increments, the process should be frozen for increments as well.

SPI Must Have a Champion

Unless someone steps up as the leader and champion of SPI, other tasks easily take precedence over SPI tasks. The champion needs to be in a powerful position in order to have leverage over the others, and be able to hold his own in arguments with the CEO. The champion also needs to be respected by the others, otherwise they easily ignore him. That is why it is not a good idea to appoint the worst developer as the SPI officer. The champion is often the most vigilant person to reinforce the process, which can easily be observed as decay in process conformance when he goes on vacation. This is something we have witnessed in several companies.

Remember these 7 basic principles for SPI:

- Management commitment
- Practitioner buy-in
- Improvement requires investment
- First understand the current process
- Change must become a way of life
- Institutionalising improvement requires periodic enforcement
- SPI must have a champion

Motivators and De-Motivators of SPI

Baddoo and Hall have published articles of SPI motivators and de-motivators (Baddoo and Hall 2002; Baddoo and Hall 2003), which are used as the basis for this section. It is important to understand what motivates people for SPI if you want to succeed with it. The motivators and de-motivators in your company may differ from the ones found in the research done by Baddoo and Hall, but the content of this section can give you an idea of things to consider, when planning for your SPI initiative. The following sections present the main motivators and de-motivators of developers, project managers, and senior managers. Although, the connection between the motivators and de-motivators is often such that the lack of a motivator acts as a de-motivator, presenting the motivators and de-motivators separately makes the presentation clearer.

Motivators

Developers

Visible success is the strongest motivator for developers. Plan your SPI efforts in a way that provides visible success as soon as possible. This visible success can then be used as momentum for further process improvement. Provide the developers with the necessary *resources* for SPI. If you constantly remind them of how important finishing all functionality of the product for the release is, they are not motivated to use any time for SPI. If improving the process does not get the status and resources it deserves, do not even try, you will fail.

Bottom-up initiatives are motivating for developers. Involve your developers in SPI, e.g., through reflection meetings, and make sure their opinions and ideas count and are implemented. Of course, not all ideas have to be implemented at once nor the bad ones (that might endanger the visible success), only the most important ones, but include the developers or a trusted representative in the decision-making process. *Top-down commitment* was already discussed in Section 6.6. Management needs to show full commitment to SPI both in words and action, or the SPI initiative is compromised, because the developers make their own judgments on the importance of tasks based on how management behaves.

Feedback is very much appreciated by the developers. You should, e.g., reward developers for a job well done, also concerning process conformance. Another point is to create measurable goals for improvement and collect data during the improvement to see how it works. The results should also be presented to the developers and the implications discussed, e.g., in reflection meetings.

Project Managers

Visible success also motivates the project managers. They naturally take pride in being able to manage their projects better when the process is improved and works. Sufficient r*esourcing* is another motivator for project managers. One way to concretise this is allocating part of the overall effort of projects to SPI activities in advance. This means that less development tasks and thus features for the product get done, but this is what is meant by sufficient resources for SPI.

Project managers are also motivated by *empowerment*. They want to have the mandate to make SPI decisions within their projects, e.g., between increments in reflection meetings. The reflection meetings involve the developers too, ensuring that the bottom-up initiatives of developers are also considered. *Easy and maintainable processes* are the final main motivator of project managers. They want that the processes employed in their projects are easy to use and maintain. This is an important topic for reflection meetings, where you should think about what could have been done easier or what has caused obstacles in the current process.

Senior Managers

Not so surprisingly, *visible success* is also the main motivator of senior management as can be seen from the second most quoted motivator, *meeting targets*. These both motivators are closely interrelated and show the viewpoint taken by senior management, i.e., the importance of being successful and meeting targets, e.g., the release goals. This kind of success can very much determine the future of the company.

Sufficient *resources* motivate senior managers too, but we have many times witnessed that this is hard to put into action, as the example showed in section 6.6. Management commitment for providing the necessary resources is needed. A motivator very close to this one is *cost effectiveness* of the processes. If the processes are easy to use and do not force too much overhead, resources are also spared.

The most important motivators for SPI are:

- Visible success
- Sufficient resources
- Empowerment / bottom-up initiatives
- Feedback
- Top-down commitment

De-Motivators

Developers

Time and budget constraints are the main de-motivators of developers. These basically mean lack of resources, which was a strong motivator. *Inertia* is also mentioned as a de-motivator for developers. This can be interpreted as the inertia to change the process, i.e., the resistance to change that some or many developers and managers may display. Another interpretation is that change and improvement takes a long time, which can prevent visible success and frustrate the developers.

Cumbersome processes are de-motivating. Naturally, the developers like processes that are easy to use and maintain, which was one of the motivators of project managers.

If the process is cumbersome to use, developers are tempted to make shortcuts, which may result in bad quality product. It is advisable to discuss these matters in the reflection meetings, before the situation gets out of control. Finally, *lack of management commitment* is de-motivating for developers, as they are quick enough to draw their own conclusions of the importance and priority of things based on observed behaviour of the managers.

Project Managers

Time and budget constraints de-motivate project managers as well. If the release project goals are too tight and no resources are allocated for process improvement SPI does not get done. Making the ongoing project successful is much more important to the project managers (and it should be), unless time and resources have been allocated for SPI. This can only be done if SPI is perceived as important for the future success of the projects and the company. *Lack of evidence of benefits* of SPI discourages project managers from allocating resources for process improvement.

Senior Managers

Time and budget constraints also de-motivate senior managers. Since the ultimate decisions of resource allocation are made by the senior managers, this is concerning. If the senior managers do not see the benefits of SPI, they are likely to prioritise other activities, even if SPI would in fact be needed. Therefore it is important for the SPI champion to be able to motivate senior management, e.g., using arguments that are motivating to senior managers (see Motivators above). Also, SPI has to be done well, because *bad previous experiences* and *lack of SPI management skills* de-motivate senior managers. Therefore we hope that this chapter has provided you with good ideas of how to do SPI successfully.

The main de-motivators for SPI are:

- Time pressure
- Lack of resources
- Lack of evidence of benefits
- Bad previous experiences
- Lack of management commitment

References

Baddoo, N. and T. Hall. 2002. Motivators of Software Process Improvement: An Analysis of Practitioner's Views. *Journal of Systems and Software* 62 (2): 85-96.

Baddoo, N. and T. Hall. 2003. De-Motivators for Software Process Improvement: an Analysis of Practitioners' Views. *Journal of Systems and Software* 66 (1): 23-33.

Basili, V. R., G. Caldiera, and H. D. Rombach. 1994. The Experience Factory. In *Encyclopaedia of Software Engineering*, ed. J. J. Marciniak 1: 469-76. John Wiley & Sons.