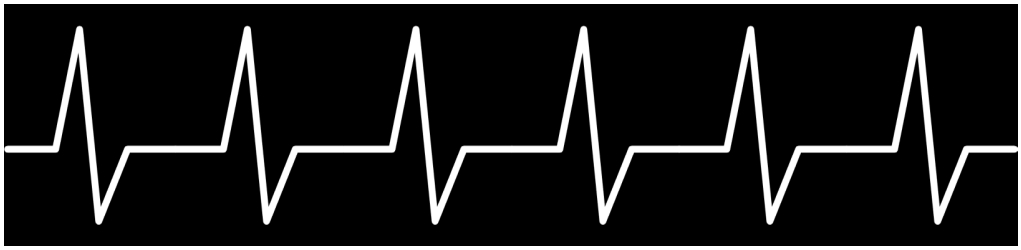
Helsinki University of Technology Software Business and Engineering Institute
Technical Reports 3

Teknillisen korkeakoulun ohjelmistoliiketoiminnan ja –tuotannon laboratorion
tekninen raportti 3

Espoo 2004 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 2004 HUT-SoberIT-C3

PACING SOFTWARE PRODUCT DEVELOPMENT:
A Framework and Practical Implementation Guidelines

Kristian Rautiainen Casper Lassenius (eds.)

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 9600, FIN-02015 HUT, Finland
Tel. +358-9-4511
Fax. +358-9-451 4958
E-mail: reports@soberit.hut.fi

© Kristian Rautiainen, Casper Lassenius

ISBN 951-22-7069-2
ISSN 1457-8050

Otamedia
Espoo 2004
2nd Printing

HELSINKI UNIVERSITY OF TECHNOLOGY
SOFTWARE BUSINESS AND ENGINEERING INSTITUTE
TECHNICAL REPORTS

ISBN 951-22-7069-2
ISSN 1457-8050

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.

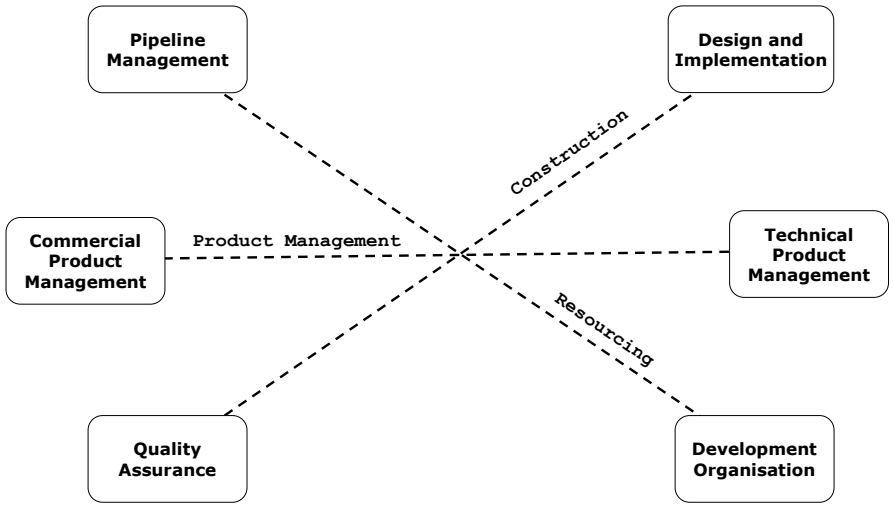


Figure 1.1: Components of a Software Engineering Management System (SEMS)

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.

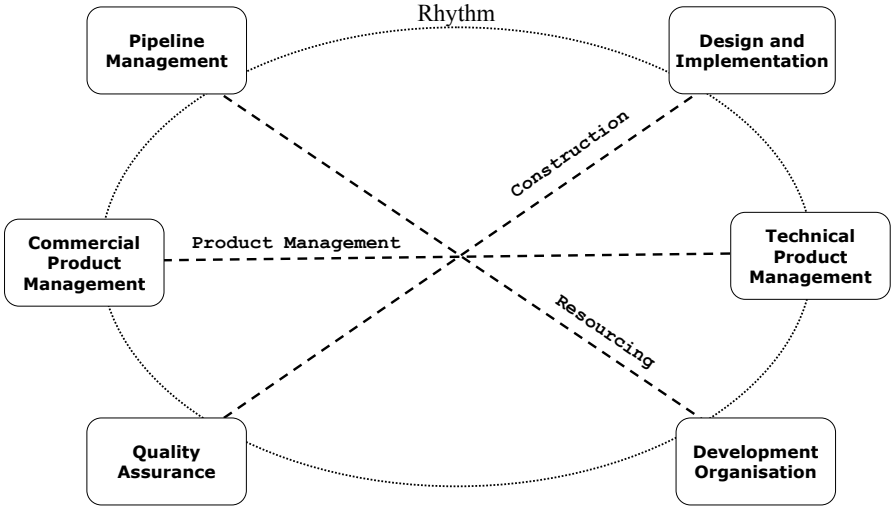


Figure 1.2: Rhythm as the coordinator of the SEMS components

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.

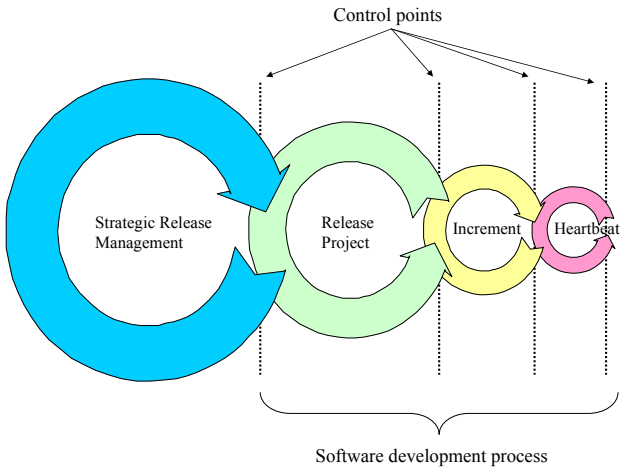


Figure 1.3: Cycles of Control building blocks

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.

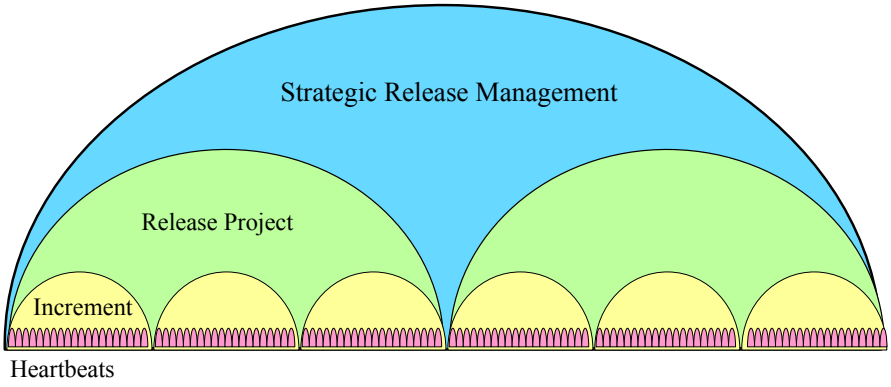


Figure 1.4: Cycles of Control on a timeline

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.

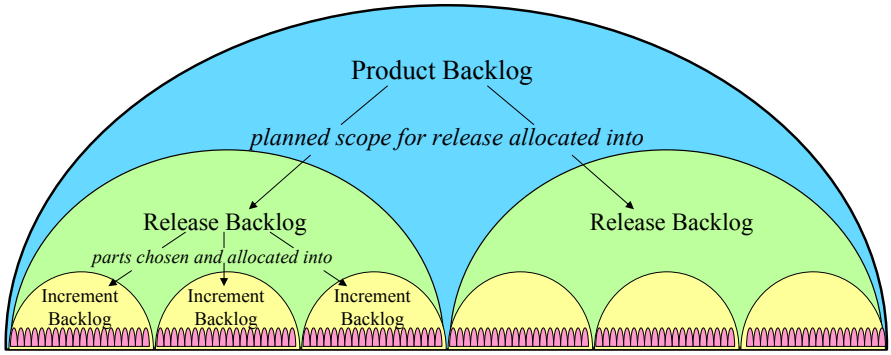


Figure 1.5: Managing requirements with backlogs

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.

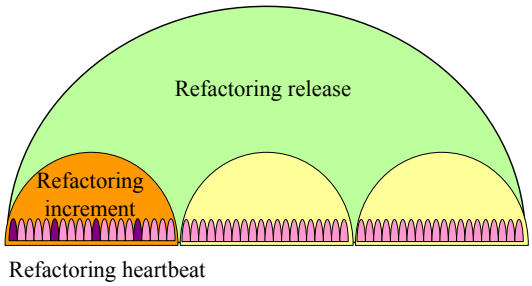


Figure 1.6: Different approaches to refactoring

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

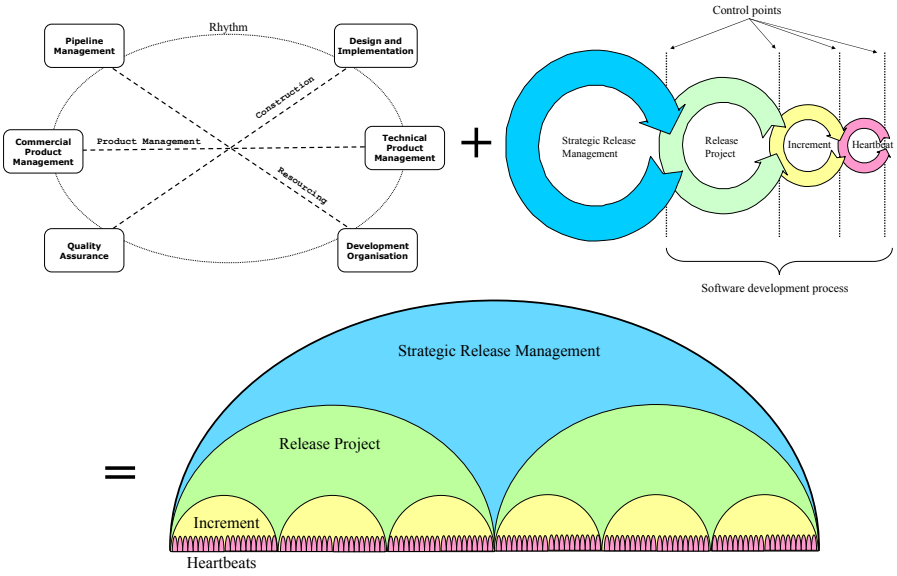


Figure 1.7: Putting it all together

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`