GLOBAL EDITION

Engineering Software Products

An Introduction to Modern Software Engineering

Ian Sommerville

ENGINEERING SOFTWARE PRODUCTS

An Introduction to Modern Software Engineering

Global Edition

Ian Sommerville

Pearson

Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Dubai • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • São Paulo • Mexico City • Madrid • Amsterdam • Munich • Paris • Milan

Pearson Education Limited

KAO Two
KAO Park
Hockham Way
Harlow
CM17 9SR
United Kingdom

and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsonglobaleditions.com

Authorized adaptation from the United States edition, entitled Engineering Software Products, *1st Edition, ISBN 978-0-13-521064-2 by* Ian Sommerville, *published by Pearson Education ©2020.*

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN 10: 1-292-37634-1
ISBN 13: 978-1-292-37634-9
eBook ISBN 13: 978-1-292-37635-6

PREFACE

Software products, such as stand-alone programs, web apps and services, and mobile apps, have transformed our everyday life and work. There are tens of thousands of software product companies, and hundreds of thousands of software engineers are employed worldwide in software product development.

Contrary to what some people may think, engineering software products needs more than coding skills. So, I've written this book to introduce some of the software engineering activities that are important for the production of reliable and secure software products.

Who is the book for?

The book has been designed for students taking a first course in software engineering. People thinking about developing a product who don't have much software engineering experience may also find it useful.

Why do we need a software engineering book that's focused on software products?

Most software engineering texts focus on *project-based* software engineering, where a client develops a specification and the software is developed by another company. However, the software engineering methods and techniques that have been developed for large-scale projects are not suited to software product development.

Students often find it difficult to relate to large, custom software systems. I think that students find it easier to understand software engineering techniques when they are relevant to the type of software that they constantly use. Also, many product engineering techniques are more directly relevant to student projects than project-oriented techniques.

Is this a new edition of your other software engineering textbook?

No, this book takes a completely different approach and, apart from a couple of diagrams, does not reuse any material from *Software Engineering*, 10th edition.

What's in the book?

Ten chapters cover software products, agile software engineering, features, scenarios and user stories, software architecture, cloud-based software, microservices architecture, security and privacy, reliable programming, testing, and DevOps and code management.

I've designed the book so that it's suitable for a one-semester software engineering course.

How is this book different from other introductory texts on software engineering?

As I said, the focus is on *products* rather than *projects*. I cover techniques that most other SE texts don't cover, such as personas and scenarios, cloud computing, microservices, security, and DevOps. As product innovation doesn't come from university research, there are no citations or references to research and the book is written in an informal style.

What do I need to know to get value from the book?

I assume that you have programming experience with a modern object-oriented programming language such as Java or Python and that you are familiar with good programming practice, such as the use of meaningful names. You should also understand basic computing concepts, such as objects, classes, and databases. The program examples in the book are written in Python, but they are understandable by anyone with programming experience.

What extra material is available to help teachers and instructors?

1. An instructor's manual with solutions to exercises and quiz questions for all chapters
2. Suggestions how you can use the book in a one-semester software engineering course
3. Presentations for teaching (Keynote, PowerPoint, and PDF)

You can access this material along with additional material at: www.pearsonglobaleditions.com

Where can I find out more?

I've written a couple of blog posts that are relevant to the book. These provide more information about my thoughts on teaching software engineering and my motivation for writing the book.

"Out with the UML (and other stuff too): reimagining introductory courses in software engineering"

https://iansommerville.com/systems-software-and-technology/what-should-we-teach-in-software-engineering-courses/

"Engineering Software Products"

https://iansommerville.com/systems-software-and-technology/engineering-software-products/

Acknowledgments

I'd like to thank the reviewers who made helpful and supportive suggestions when they reviewed the initial proposal for this book:

Paul Eggert—*UCLA Los Angeles*
Jeffrey Miller—*University of Southern California*
Harvey Siy—*University of Nebraska Omaha*
Edmund S. Yu—*Syracuse University*
Gregory Gay—*University of South Carolina*
Josh Delinger—*Towson University*
Rocky Slavin—*University of Texas San Antonio*
Bingyang Wei—*Midwestern State University*

Thanks also to Adam Barker from St. Andrews University for keeping me right on containers and to Rose Kernan who managed the production of the book.

Thanks, as ever, to my family for their help and support while I was writing the book. Particular thanks to my daughter Jane, who did a great job of reading and commenting on the text. She was a brutal editor! Her suggested changes significantly improved the quality of my prose.

Finally, special thanks to our newest family member, my beautiful grandson Cillian, who was born while I was writing this book. His bubbly personality and constant smiles were a very welcome distraction from the sometimes tedious job of book writing and editing.

Ian Sommerville

CONTENTS

1

Software Products

This book introduces software engineering techniques that are used to develop software products. Software products are generic software systems sold to governments, businesses, and consumers. They may be designed to support a business function, such as accounting; they may be productivity tools, such as note-taking systems; or they may be games or personal information systems. Software products range in size from millions of lines of code in large-scale business systems to a few hundred lines of code in a simple app for mobile phones.

We all use software products every day on our computers, tablets, and phones. I am using a software product—the Ulysses editor—to write this book. I'll use another editing product—Microsoft Word—to format the final version, and I'll use Dropbox to exchange the files with the publisher. On my phone, I use software products (apps) to read email, read and send tweets, check the weather, and so on.

The engineering techniques that are used for product development have evolved from the software engineering techniques developed in the 20th century to support custom software development. When software engineering emerged as a discipline in the 1970s, virtually all professional software was "one-off," custom software. Companies and governments wanted to automate their businesses, and they specified what they wanted their software to do. An in-house engineering team or an external software company then developed the software.

Examples of custom software that were developed around that time include:

- the U.S. Federal Aviation Administration's air traffic management system;
- accounting systems for all of the major banks;

- billing systems for utility companies such as electricity and gas suppliers;
- military command and control systems.

Software projects were set up to develop these one-off systems, with the software system based on a set of software requirements. The contract between the software customer and the software development company included a requirements document, which was a specification of the software that should be delivered. Customers defined their requirements and worked with the development team to specify, in detail, the software's functionality and its critical attributes.

This project-based approach dominated the software industry for more than 25 years. The methods and techniques that evolved to support project-based development came to define what was meant by "software engineering." The fundamental assumption was that successful software engineering required a lot of preparatory work before starting to write programs. For example, it was important to spend time getting the requirements "right" and to draw graphical models of the software. These models were created during the software design process and used to document the software.

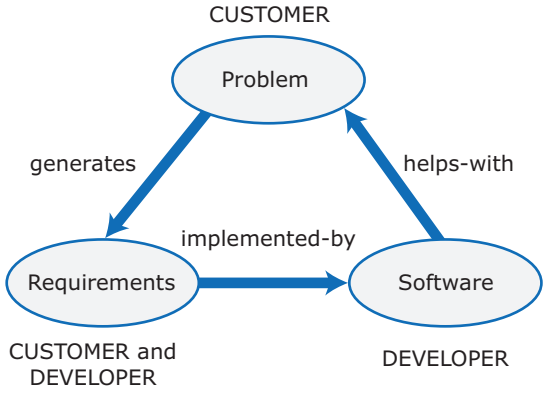
As more and more companies automated their business, however, it became clear that most businesses didn't really need custom software. They could use generic software products that were designed for common business problems. The software product industry developed to meet this need. Project-based software engineering techniques were adapted to software product development.

Project-based techniques are not suited to product development because of fundamental differences between project-based and product-based software engineering. These differences are illustrated in Figures 1.1 and 1.2.

Software projects involve an external client or customer who decides on the functionality of the system and enters into a legal contract with the software development company. The customer's problem and current processes are used as a basis for creating the software requirements, which specify the software to be implemented. As the business changes, the supporting software has to change. The company using the software decides on and pays for the changes. Software often has a long lifetime, and the costs of changing large systems after delivery usually exceed the initial software development costs.

Software products are specified and developed in a different way. There is no external customer who creates requirements that define what the software

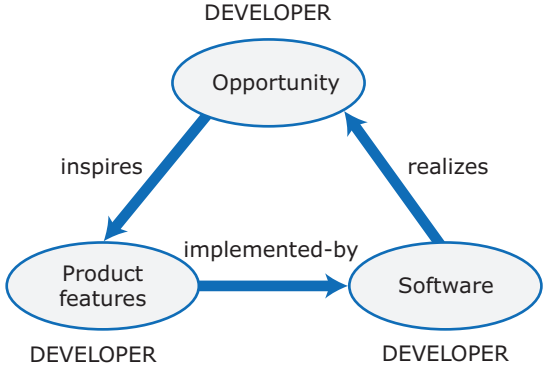
Figure 1.1 Project-based software engineering



must do. The software developer decides on the features of the product, when new releases are to be made available, the platforms on which the software will be implemented, and so on. The needs of potential customers for the software are obviously considered, but customers can't insist that the software includes particular features or attributes. The development company chooses when changes will be made to the software and when they will be released to users.

As development costs are spread over a much larger customer base, product-based software is usually cheaper, for each customer, than custom software. However, buyers of the software have to adapt their ways of working to the software, since it has not been developed with their specific needs in mind. As the developer rather than the user is in control of changes, there

Figure 1.2 Product-based software engineering



is a risk that the developer will stop supporting the software. Then the product customers will need to find an alternative product.

The starting point for product development is an opportunity that a company has identified to create a viable commercial product. This may be an original idea, such as Airbnb's idea for sharing accommodations; an improvement over existing systems, such as a cloud-based accounting system; or a generalization of a system that was developed for a specific customer, such as an asset management system.

Because the product developer is responsible for identifying the opportunity, they can decide on the features that will be included in the software product. These features are designed to appeal to potential customers so that there is a viable market for the software.

As well as the differences shown in Figures 1.1 and 1.2, there are two other important differences between project-based and product-based software engineering:

1. Product companies can decide when to change their product or take their product off the market. If a product is not selling well, the company can cut costs by stopping its development. Custom software developed in a software project usually has a long lifetime and has to be supported throughout that lifetime. The customer pays for the support and decides when and if it should end.
2. For most products, getting the product to customers quickly is critical. Excellent products often fail because an inferior product reaches the market first and customers buy that product. In practice, buyers are reluctant to change products after they have invested time and money in their initial choice.

Bringing the product to the market quickly is important for all types of products, from small-scale mobile apps to enterprise products such as Microsoft Word. This means that engineering techniques geared to rapid software development (agile methods) are universally used for product development. I explain agile methods and their role in product development in Chapter 2.

If you read about software products, you may come across two other terms: "software product lines" and "platforms" (Table 1.1). Software product lines are systems designed to be adaptable to meet the specific needs of customers by changing parts of the source code. Platforms provide a set of features that can be used to create new functionality. However, you always have to work within the constraints defined by the platform suppliers.

Table 1.1 Software product lines and platforms

Technology	Description
Software product line	A set of software products that share a common core. Each member of the product line includes customer-specific adaptations and additions. Software product lines may be used to implement a custom system for a customer with specific needs that can't be met by a generic product. For example, a company providing communication software to the emergency services may have a software product line where the core product includes basic communication services such as receive and log calls, initiate an emergency response, pass information to vehicles, and so on. However, each customer may use different radio equipment and their vehicles may be equipped in different ways. The core product has to be adapted for each customer to work with the equipment that they use.
Platform	A software (or software+hardware) product that includes functionality so that new applications can be built on it. An example of a platform that you probably use is Facebook. It provides an extensive set of product functionality but also provides support for creating "Facebook apps." These add new features that may be used by a business or a Facebook interest group.

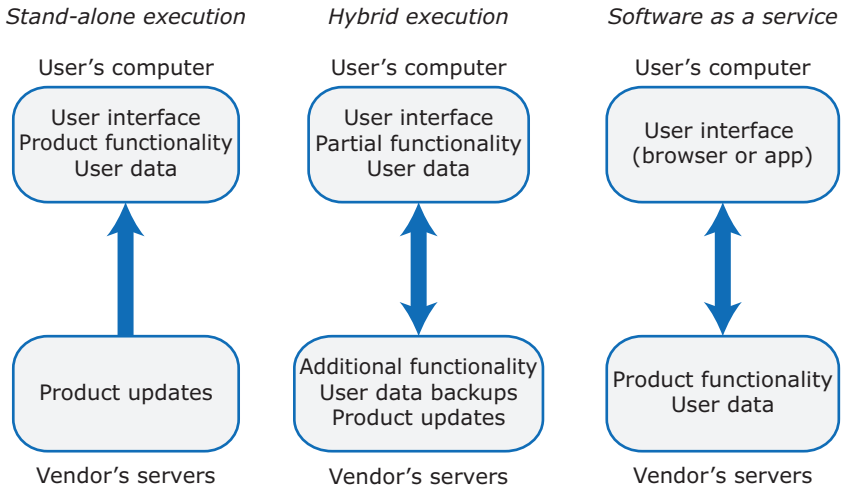
When software products were first developed, they were delivered on a disk and installed by customers on their computers. The software ran on those computers and user data were stored on them. There was no communication between the users' computers and the vendor's computers. Now, customers can download products from either an app store or the vendor's website.

Some products are still based on a stand-alone execution model in which all computation is carried out on the product owner's computers. However, ubiquitous high-speed networking means that alternative execution models are now available. In these models, the product owner's computers act as a client, with some or all execution and data storage on the vendor's servers (Figure 1.3).

There are two alternatives to stand-alone software products:

1. *Hybrid products* Some functionality is implemented on the user's computer and some on the product vendor's servers that are accessed over the Internet. Many phone apps are hybrid products with computationally intensive processing offloaded to remote servers.
2. *Service-based products* Applications are accessed over the Internet from a web browser or an app. There may be some local processing using

Figure 1.3 Software execution models



Javascript, but most computation is carried out on remote servers. More and more product companies are converting their products to services because it simplifies product updating and makes new business models, such as pay-as-you-go, feasible. I cover service-oriented systems in Chapters 5 and 6.

As I have said, the key characteristic of product development is that there is no external customer who generates software requirements and pays for the software. This is also true for some other types of software development:

1. *Student projects* As part of a computing or engineering course, students may be set assignments in which they work in groups to develop software. The group is responsible for deciding on the features of the system and how to work together to implement these features.
2. *Research software* Software is developed by a research team to support their work. For example, climate research depends on large-scale climate models that are designed by researchers and implemented in software. On a smaller scale, an engineering group may build software to model the characteristics of the material they are using.
3. *Internal tool development* A software development team may decide that it needs some specific tools to support their work. They specify and implement these tools as "internal" products.

You can use the product development techniques that I explain here for any type of software development that is not driven by external customer requirements.

There is a common view that software product engineering is simply advanced programming and that traditional software engineering is irrelevant. All you need to know is how to use a programming language plus the frameworks and libraries for that language. This is a misconception and I have written this book to explain the activities, apart from programming, that I believe are essential for developing high-quality software products.

If your product is to be a success, you need to think about issues other than programming. You must try to understand what your customers need and how potential users can work with your software. You need to design the overall structure of your software (software architecture) and know about technologies such as cloud computing and security engineering. You need to use professional techniques for verifying and testing your software and code management systems to keep track of a changing codebase.

You also need to think about the business case for your product. You must sell your product to survive. Creating a business case may involve market research, an analysis of competitors, and an understanding of the ways that target customers live and work. This book is about engineering, however, not business, so I don't cover business and commercial issues here.

1.1 The product vision

Your starting point for product development should be an informal "product vision." A product vision is a simple and succinct statement that defines the essence of the product that is being developed. It explains how the product differs from other competing products. This product vision is used as a basis for developing a more detailed description of the features and attributes of the product. As new features are proposed, you should check them against the vision to make sure they contribute to it.

The product vision should answer three fundamental questions:

1. *What* is the product that you propose to develop? What makes this product different from competing products?
2. *Who* are the target users and customers for the product?
3. *Why* should customers buy this product?

The need for the first question is obvious—before you start, you need to know what you are aiming for. The other questions concern the commercial viability of the product. Most products are intended for use by customers outside of the development team. You need to understand their background to create a viable product that these customers will find attractive and be willing to buy.

If you search the web for "product vision," you will find several variants of these questions and templates for expressing the product vision. Any of these templates can be used. The template that I like comes from the book *Crossing the Chasm* by Geoffrey Moore.[1] Moore suggests using a structured approach to writing the product vision based on keywords:

- FOR (target customer)
- WHO (statement of the need or opportunity)
- The (PRODUCT NAME) is a (product category)
- THAT (key benefit, compelling reason to buy)
- UNLIKE (primary competitive alternative)
- OUR PRODUCT (statement of primary differentiation)

On his blog *Joel on Software*, Joel Spolsky gives an example of a product described using this vision template:[2]

> *FOR a mid-sized company's marketing and sales departments WHO need basic CRM functionality, THE CRM-Innovator is a Web-based service THAT provides sales tracking, lead generation, and sales representative support features that improve customer relationships at critical touch points. UNLIKE other services or package software products, OUR product provides very capable services at a moderate cost.*

You can see how this vision answers the key questions that I identified above:

1. *What* A web-based service that provides sales tracking, lead generation, and sales representative support features. The information can be used to improve relationships with customers.

[1] Geoffrey Moore, *Crossing the Chasm: Marketing and selling technology products to mainstream customers* (Capstone Trade Press, 1998).

[2] J. Spolsky, Product Vision, 2002; http://www.joelonsoftware.com/articles/JimHighsmithonProductVisi.html

Table 1.2 Information sources for developing a product vision

Information source	Explanation
Domain experience	The product developers may work in a particular area (say, marketing and sales) and understand the software support that they need. They may be frustrated by the deficiencies in the software they use and see opportunities for an improved system.
Product experience	Users of existing software (such as word processing software) may see simpler and better ways of providing comparable functionality and propose a new system that implements this. New products can take advantage of recent technological developments such as speech interfaces.
Customer experience	The software developers may have extensive discussions with prospective customers of the product to understand the problems that they face; constraints, such as interoperability, that limit their flexibility to buy new software; and critical attributes of the software that they need.
Prototyping and "playing around"	Developers may have an idea for software but need to develop a better understanding of that idea and what might be involved in developing it into a product. They may develop a prototype system as an experiment and "play around" with ideas and variations using that prototype system as a platform.

2. *Who* The product is aimed at medium-sized companies that need standard customer relationship management software.
3. *Why* The most important product distinction is that it provides capable services at a moderate cost. It will be cheaper than alternative products.

A great deal of mythology surrounds software product visions. For successful consumer software products, the media like to present visions as if they emerge from a "Eureka moment" when the company founders have an "awesome idea" that changes the world. This view oversimplifies the effort and experimentation that are involved in refining a product idea. Product visions for successful products usually emerge after a lot of work and discussion. An initial idea is refined in stages as more information is collected and the development team discusses the practicalities of product implementation. Several different sources of information contribute to the product vision (Table 1.2).

Table 1.3 A vision statement for the iLearn system

FOR teachers and educators *WHO* need a way to help students use web-based learning resources and applications, *THE iLearn system* is an open learning environment *THAT* allows the set of resources used by classes and students to be easily configured for these students and classes by teachers themselves.
UNLIKE Virtual Learning Environments, such as Moodle, the focus of iLearn is the learning process rather than the administration and management of materials, assessments, and coursework. *OUR* product enables teachers to create subject and age-specific environments for their students using any web-based resources, such as videos, simulations, and written materials that are appropriate.

Schools and universities are the target customers for *the iLearn system* as it will significantly improve the learning experience of students at relatively low cost. It will collect and process learner analytics that will reduce the costs of progress tracking and reporting.

1.1.1 A vision example

As students, readers of this book may have used Virtual Learning Environments (VLEs), such as Blackboard and Moodle. Teachers use these VLEs to distribute class materials and assignments. Students can download the materials and upload completed assignments. Although the name suggests that VLEs are focused on learning, they are really geared to supporting learning administration rather than learning itself. They provide some features for students, but they are not open learning environments that can be tailored and adapted to a particular teacher's needs.

A few years ago, I worked on the development of a digital environment for learning support. This product was not just another VLE but was intended to provide flexible support for the process of learning. Our team looked at existing VLEs and talked to teachers and students who used them. We visited different types of school from kindergartens to colleges to examine how they used learning environments and how teachers were experimenting with software outside of these environments. We had extensive discussions with teachers about what they would like to be able to do with a digital learning environment. We finally arrived at the vision statement shown in Table 1.3.

In education, the teachers and students who use learning environments are not responsible for buying software. The purchaser is a school, university, or training center. The purchasing officer needs to know the benefits to the organization. Therefore, we added the final paragraph to the vision statement in Table 1.3 to make clear that there are benefits to organizations as well as individual learners.

1.2 Software product management

Software product management is a business activity focusing on the software products that are developed and sold by the business. Product managers (PMs) take overall responsibility for the product and are involved in planning, development, and marketing. They are the interface between the software development team, the broader organization, and the product's customers. PMs should be full members of the development team so that they can communicate business and customer requirements to the software developers.

Software product managers are involved at all stages of a product's life—from initial conception through vision development and implementation to marketing. Finally, they make decisions on when the product should be withdrawn from the market. Mid-size and large software companies may have dedicated PMs; in smaller software companies, the PM role is likely to be shared with other technical or business roles.

The job of the PM is to look outward to the customers and potential customers of the product rather than to focus on the software that is being developed. It is all too easy for a development team to get caught up in the details of "cool features" of the software, which most customers probably don't care about. For a product to be successful, the PM has to ensure that the development team implements features that deliver real value to customers, not just features that are technically interesting.

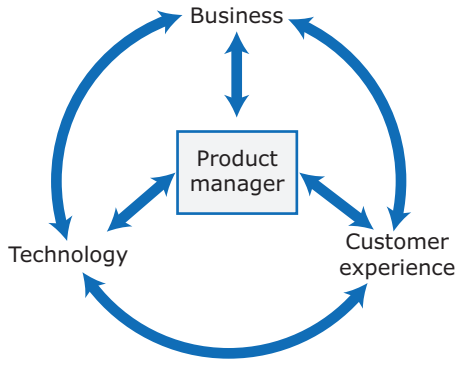
In a blog post, Martin Eriksson[3] explains that product managers have to be concerned with business, technology, and user experience issues. Figure 1.4, which I based on Martin's diagram, illustrates these multiple concerns.

Product managers have to be generalists, with both technical and communication skills. Business, technology, and customer issues are interdependent and PMs have to consider all of them:

1. *Business needs* PMs have to ensure that the software being developed meets the business goals and objectives of both the software product company and its customers. They must communicate the concerns and needs of the customers and the development team to the managers of the product business. They work with senior managers and with marketing staff to plan a release schedule for the product.

[3]Based on M. Erikkson, What, exactly, is a Product Manager, 2011; http://www.mindtheproduct.com/2011/10/what-exactly-is-a-product-manager/

Figure 1.4 Product management concerns



2. *Technology constraints* PMs must make developers aware of technology issues that are important to customers. These may affect the schedule, cost, and functionality of the product that is being developed.
3. *Customer experience* PMs should be in regular communication with customers to understand what they are looking for in a product, the types of user and their backgrounds, and the ways in which the product may be used. Their experience of customer capabilities is a critical input to the design of the product's user interface. PMs may also involve customers in alpha and beta product testing.

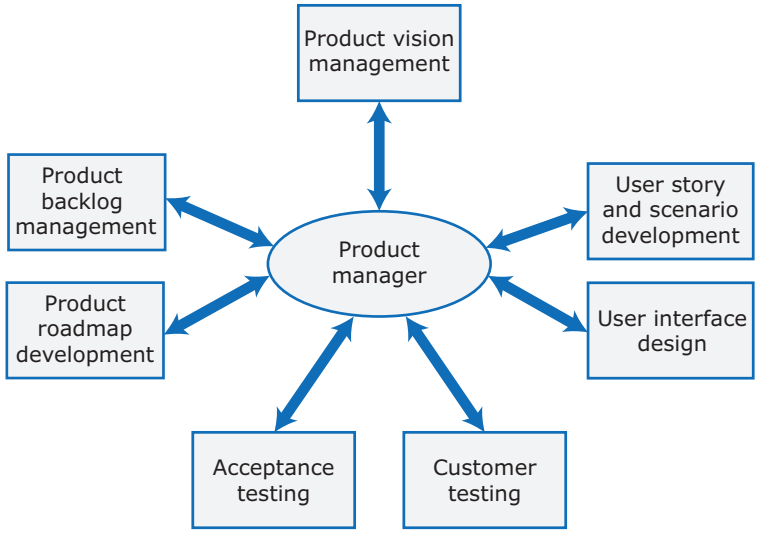
Because of the engineering focus of this book, I do not go into detail about the business role of product managers or their role in areas such as market research and financial planning. Rather, I concentrate on their interactions with the development team. PMs may interact with the development team in seven key areas (Figure 1.5).

1.2.1 Product vision management

Some writers say that the product manager should be responsible for developing the product vision. Large companies may adopt this approach, but it is often impractical in small software companies. In startups, the source of the product vision is often an original idea by the company founders. This vision is often developed long before anyone thinks about appointing a PM.

Obviously, it makes sense for PMs to take the lead in developing the product vision. They should be able to bring market and customer information to

Figure 1.5 Technical interactions of product managers



the process. However, I think all team members should be involved in vision development so that everyone can support what is finally agreed. When the team "owns" the vision, everyone is more likely to work coherently to realize that vision.

A key role of PMs is to manage the product vision. During the development process, changes are inevitably proposed by people from both inside and outside of the development team. PMs have to assess and evaluate these changes against the product vision. They must check that the changes don't contradict the ideas embodied in the product vision. PMs also have to ensure that there is no "vision drift," in which the vision is gradually extended to become broader and less focused.

1.2.2 Product roadmap development

A product roadmap is a plan for the development, release, and marketing of the software product. It sets out important product goals and milestones, such as the completion of critical features, the completion of the first version for user testing, and so on. It includes dates when these milestones should be reached and success criteria that help assess whether project goals have been attained. The roadmap should include a release schedule showing when

different releases of the software will be available and the key features that will be included in each release.

The development of the product roadmap should be led by the product manager but must also involve the development team as well as company managers and marketing staff. Depending on the type of product, important deadlines may have to be met if the product is to be successful. For example, many large companies must make decisions on procurement toward the end of their financial year. If you want to sell a new product to such companies, you have to make it available before then.

1.2.3 User story and scenario development

User stories and scenarios are widely used to refine a product vision to identify features of the product. They are natural language descriptions of things that users might want to do with a product. Using them, the team can decide what features need to be included and how these features should work. I cover user stories and scenarios in Chapter 3.

The product manager's job is to understand the product's customers and potential customers. PMs should therefore lead the development of user scenarios and stories, which should be based on knowledge of the area and of the customer's business. PMs should also take scenarios and stories suggested by other team members back to customers to check that they reflect what the target users of the product might actually do.

1.2.4 Product backlog management

In product development, it is important for the process to be driven by a "product backlog." A product backlog is a to-do list that sets out what has to be done to complete the product development. The backlog is added to and refined incrementally during the development process. I explain how product backlogs are used in the Scrum method in Chapter 2.

The product manager plays a critical role as the authority on the product backlog items that should take priority for development. PMs also help to refine broad backlog items, such as "implement auto-save," in more detail at each project iteration. If suggestions for change are made, it is up to the PM to decide whether or not the product backlog should be rearranged to prioritize the suggested changes.

1.2.5 Acceptance testing

Acceptance testing is the process of verifying that a software release meets the goals set out in the product roadmap and that the product is efficient and reliable. Product managers should be involved in developing tests of the product features that reflect how customers use the product. They may work through usage scenarios to check that the product is ready to be released to customers.

Acceptance tests are refined as the product is developed, and products must pass these tests before being released to customers.

1.2.6 Customer testing

Customer testing involves taking a release of a product to existing and potential customers and getting feedback from them on the product's features, its usability, and the fit of the product to their business. Product managers are involved in selecting customers that might be interested in taking part in the customer testing process and working with them during that process. They have to ensure that the customer can use the product and that the customer testing process collects useful information for the development team.

1.2.7 User interface design

The user interface (UI) of a product is critical in the commercial acceptance of a software product. Technically excellent products are unlikely to be commercially successful if users find them difficult to use or if their UI is incompatible with other software that they use. UI design is challenging for small development teams because most users are less technically skilled than software developers. It is often difficult for developers to envision the problems that users may have with a software product.

Product managers should understand user limitations and act as surrogate users in their interactions with the development team. They should evaluate UI features as they are developed to check that these features are not unnecessarily complex or force users to work in an unnatural way. PMs may arrange for potential users to try out the software, comment on its UI, and assist with designing error messages and a help system.

1.3 Product prototyping

Product prototyping is the process of developing an early version of a product to test your ideas and to convince yourself and company funders that your product has real market potential. You use a product prototype to check that what you want to do is feasible and to demonstrate your software to potential customers and funders. Prototypes may also help you understand how to organize and structure the final version of your product.

You may be able to write an inspiring product vision, but your potential users can only really relate to your product when they see a working version of your software. They can point out what they like and don't like about it and make suggestions for new features. Venture capitalists, whom you may approach for funding, usually insist on seeing a product prototype before they commit to supporting a startup company. The prototype plays a critical role in convincing investors that your product has commercial potential.

A prototype may also help identify fundamental software components or services and test technology. You may find that the technology you planned to use is inadequate and that you have to revise your ideas on how to implement the software. For example, you may discover that the design you chose for the prototype cannot handle the expected load on the system, so you have to redesign the overall product architecture.

Building a prototype should be the first thing you do when developing a software product. Your goal should be to have a working version of your software that can be used to demonstrate its key features. A short development cycle is critical; you should aim to have a demonstrable system up and running in four to six weeks. Of course, you have to cut corners to do this, so you may choose to ignore issues such as reliability and performance and work with a rudimentary user interface.

Sometimes prototyping is a two-stage process:

1. *Feasibility demonstration* You create an executable system that demonstrates the new ideas in your product. The goals at this stage are to see whether your ideas actually work and to show funders and company management that your product features are better than those of competitors.

2. *Customer demonstration* You take an existing prototype created to demonstrate feasibility and extend it with your ideas for specific customer features and how these can be realized. Before you develop a customer prototype,

you need to do some user studies and have a clear idea of your potential users and scenarios of use. I explain how to develop user personas and usage scenarios in Chapter 3.

You should always use technology that you know and understand to develop a prototype so that you don't have to spend time learning a new language or framework. You don't need to design a robust software architecture. You may leave out security features and checking code to ensure software reliability. However, I recommend that, for prototypes, you should always use automated testing and code management. These are covered in Chapters 9 and 10.

If you are developing software without an external customer, such as software for a research group, it may be that a prototype system is all you need. You can develop and refine the prototype as your understanding of the problem develops. However, as soon as you have external users of your software, you should always think of your prototype as a "throw-away" system. The inevitable compromises and shortcuts you make to speed up development result in prototypes that become increasingly difficult to change and evolve to include new features. Adding security and reliability may be practically impossible.

KEY POINTS

- Software products are software systems that include general functionality that is likely to be useful to a wide range of customers.
- In product-based software engineering, the same company is responsible for deciding on both the features that should be part of the product and the implementation of these features.
- Software products may be delivered as stand-alone products running on the customer's computers, hybrid products, or service-based products. In hybrid products, some features are implemented locally and others are accessed from the Internet. All features are remotely accessed in service-based products.
- A product vision succinctly describes what is to be developed, who are the target customers for the product, and why customers should buy the product you are developing.
- Domain experience, product experience, customer experience, and an experimental software prototype may all contribute to the development of the product vision.

- Key responsibilities of product managers are to own the product vision, develop a product roadmap, create user stories and scenarios, manage the product backlog, conduct customer and acceptance testing, and design the user interface.
- Product managers work at the interface between the business, the software development team, and the product customers. They facilitate communication among these groups.
- You should always develop a product prototype to refine your own ideas and to demonstrate the planned product features to potential customers.

RECOMMENDED READING

"What is Product Line Engineering?" This article and the two linked articles provide an overview of software product line engineering and highlight the differences between product line engineering and software product development. (Biglever Software, 2013)

http://www.productlineengineering.com/overview/what-is-ple.html

"Building Software Products vs Platforms" This blog post briefly explains the differences between a software product and a software platform. (B. Algave, 2016)

https://blog.frogslayer.com/building-software-products-vs-platforms/

"Product Vision" This is an old article but an excellent summary of what is meant by a product vision and why it is important. (J. Spolsky, 2002)

http://www.joelonsoftware.com/articles/JimHighsmithonProductVisi.html

Agile Product Management with Scrum I generally avoid recommending books on product management as they are too detailed for most readers of this book. However, this book is worth looking at because of its focus on software and its integration with the Scrum agile method that I cover in Chapter 2. It's a short book that includes a succinct introduction to product management and discusses the creation of a product vision. (R. Pichler, 2010, Addison-Wesley)

The author's blog also has articles on product management.

http://www.romanpichler.com/blog/romans-product-management-framework/

"What, Exactly, is a Product Manager?" This excellent blog post explains why it's important that product managers work at the intersection of business, technology, and users. (M. Eriksson, 2011)

http://www.mindtheproduct.com/2011/10/what-exactly-is-a-product-manager/

PRESENTATIONS, VIDEOS, AND LINKS

https://iansommerville.com/engineering-software-products/software-products

EXERCISES

1.1. Briefly describe the fundamental differences between project-based and product-based software engineering.

1.2. What are three important differences between software products and software product lines.

1.3. Based on the example project vision for the iLearn system, identify the WHAT, WHO, and WHY for that software product.

1.4. Why do software product managers have to be generalists, with a range of skills, rather than simply technical specialists?

1.5. You are a software product manager for a company developing educational software products based on scientific simulations. Explain why it is important to develop a product roadmap so that final product releases are available in the first three months of the year.

1.6. Why should you implement a prototype before you start developing a new software product?

2
Agile Software Engineering

Bringing a software product to the market quickly is critically important. This is true for all types of products—from simple mobile apps to large-scale enterprise products. If a product is released later than planned, a competitor may have already captured the market or you may have missed a market window, such as the beginning of the holiday season. Once users have committed to a product, they are usually reluctant to change, even to a technically superior product.

Agile software engineering focuses on delivering functionality quickly, responding to changing product specifications, and minimizing development overheads. An "overhead" is any activity that doesn't contribute directly to rapid product delivery. Rapid development and delivery and the flexibility to make changes quickly are fundamental requirements for product development.

A large number of "agile methods" have been developed. Each has its adherents, who are often evangelical about the method's benefits. In practice, companies and individual development teams pick and choose agile techniques that work for them and that are most appropriate for their size and the type of product they are developing. There is no best agile method or technique. It depends on who is using the technique, the development team, and the type of product being developed.

2.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to create good software was to use controlled and rigorous software development processes. The processes included detailed project planning, requirements

specification and analysis, the use of analysis and design methods supported by software tools, and formal quality assurance. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems. These were "one-off" systems, based on the customer requirements.

This approach is sometimes called plan-driven development. It evolved to support software engineering where large teams developed complex, long-lifetime systems. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is a control system for a modern aircraft. Developing an avionic system might take five to ten years from initial specification to on-board deployment.

Plan-driven development involves significant overhead in planning, designing, and documenting the system. This overhead is justifiable for critical systems where the work of several development teams must be coordinated and different people may maintain and update the software during its lifetime. Detailed documents describing the software requirements and design are important when informal team communications are impossible.

If plan-driven development is used for small and medium-sized software products, however, the overhead involved is so large that it dominates the software development process. Too much time is spent writing documents that may never be read rather than writing code. The system is specified in detail before implementation begins. Specification errors, omissions, and misunderstandings are often discovered only after a significant chunk of the system has been implemented.

To fix these problems, developers have to redo work that they thought was complete. As a consequence, it is practically impossible to deliver software quickly and to respond rapidly to requests for changes to the delivered software.

Dissatisfaction with plan-driven software development led to the creation of agile methods in the 1990s. These methods allowed the development team to focus on the software itself, rather than on its design and documentation. Agile methods deliver working software quickly to customers, who can then propose new or different requirements for inclusion in later versions of the system. They reduce process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used.

The philosophy behind agile methods is reflected in the agile manifesto[1] that was agreed on by the leading developers of these methods. Table 2.1 shows the key message in the agile manifesto.

[1]Retrieved from http://agilemanifesto.org/. Used with permission.

Table 2.1 The agile manifesto

We are uncovering better ways of developing software by doing it and helping others to do it. Through this work, we have come to value:

- individuals and interactions over processes and tools;
- working software over comprehensive documentation;
- customer collaboration over contract negotiation;
- responding to change over following a plan.

While there is value on the items on the right, we value the items on the left more.

All agile methods are based on incremental development and delivery. The best way to understand incremental development is to think of a software product as a set of features. Each feature does something for the software user. There might be a feature that allows data to be entered, a feature to search the entered data, and a feature to format and display the data. Each software increment should implement a small number of product features.

With incremental development, you delay decisions until you really need to make them. You start by prioritizing the features so that the most important features are implemented first. You don't worry about the details of all the features—you define only the details of the feature that you plan to include in an increment. That feature is then implemented and delivered. Users or surrogate users can try it out and provide feedback to the development team. You then go on to define and implement the next feature of the system.

I show this process in Figure 2.1, and I describe incremental development activities in Table 2.2.

Figure 2.1 Incremental development

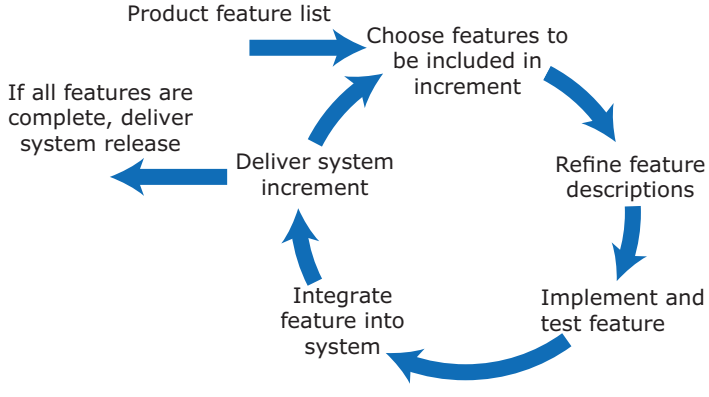


Table 2.2 Incremental development activities

Activity	Description
Choose features to be included in an increment	Using the list of features in the planned product, select those features that can be implemented in the next product increment.
Refine feature descriptions	Add detail to the feature descriptions so that the team members have a common understanding of each feature and there is sufficient detail to begin implementation.
Implement and test	Implement the feature and develop automated tests for that feature that show that its behavior is consistent with its description. I explain automated testing in Chapter 9.
Integrate feature and test	Integrate the developed feature with the existing system and test it to check that it works in conjunction with other features.
Deliver system increment	Deliver the system increment to the customer or product manager for checking and comments. If enough features have been implemented, release a version of the system for customer use.

Of course, reality doesn't always match this simple model of feature development. Sometimes an increment has to be devoted to developing an infrastructure service, such as a database service, that is used by several features; sometimes you need to plan the user interface so that you get a consistent interface across features; and sometimes an increment has to sort out problems, such as performance issues, that were discovered during system testing.

All agile methods share a set of principles based on the agile manifesto, so they have much in common. I summarize these agile principles in Table 2.3.

Almost all software products are now developed with an agile approach. Agile methods work for product engineering because software products are usually stand-alone systems rather than systems composed of independent subsystems. They are developed by co-located teams who can communicate informally. The product manager can easily interact with the development team. Consequently, there is no need for formal documents, meetings, and cross-team communication.

Table 2.3 Agile development principles

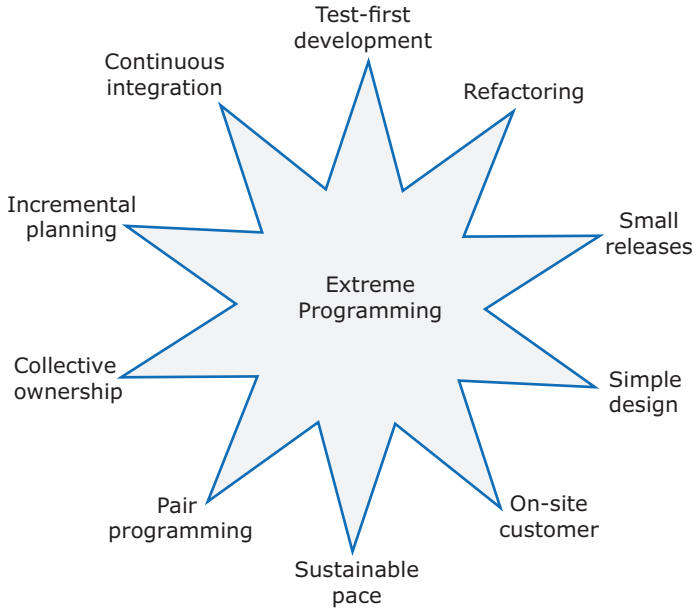
Principle	Description
Involve the customer	Involve customers closely with the software development team. Their role is to provide and prioritize new system requirements and to evaluate each increment of the system.
Embrace change	Expect the features of the product and the details of these features to change as the development team and the product manager learn more about the product. Adapt the software to cope with changes as they are made.
Develop and deliver incrementally	Always develop software products in increments. Test and evaluate each increment as it is developed and feed back required changes to the development team.
Maintain simplicity	Focus on simplicity in both the software being developed and the development process. Wherever possible, do what you can to eliminate complexity from the system.
Focus on people, not the development process	Trust the development team and do not expect everyone to always do things in the same way. Team members should be left to develop their own ways of working without being limited by prescriptive software processes.

2.2 Extreme Programming

The ideas underlying agile methods were developed by a number of different people in the 1990s. However, the most influential work that has changed the culture of software development was the development of Extreme Programming (XP). The name was coined by Kent Beck in 1998 because the approach pushed recognized good practice, such as iterative development, to "extreme" levels. For example, regular integration, in which the work of all programmers in a team is integrated and tested, is good software engineering practice. XP advocates that changed software should be integrated several times per day, as soon as the changes have been tested.

XP focused on new development techniques that were geared to rapid, incremental software development, change, and delivery. Figure 2.2 shows 10 fundamental practices, proposed by the developers of Extreme Programming, that characterize XP.

Figure 2.2 Extreme Programming practices



The developers of XP claim that it is a holistic approach. All of these practices are essential. In reality, however, development teams pick and choose the techniques that they find useful given their organizational culture and the type of software they are writing. Table 2.4 describes XP practices that have become part of mainstream software engineering, particularly for software product development. The other XP practices shown in Figure 2.2 have been less widely adopted but are used in some companies.

I cover these widely-used XP practices, in later chapters of the book. Incremental planning and user stories are covered in Chapter 3, refactoring in Chapter 8, test-driven development in Chapter 9, and continuous integration and small releases in Chapter 10.

You may be surprised that "Simple design" is not on the list of popular XP practices. The developers of XP suggested that the "YAGNI" (You Ain't Gonna Need It) principle should apply when designing software. You should include only functionality that is requested, and you should not add extra code to cope with situations anticipated by the developers. This sounds like a great idea.

Unfortunately, it ignores the fact that customers rarely understand system-wide issues such as security and reliability. You need to design and implement software to take these issues into account. This usually means including code to cope with situations that customers are unlikely to foresee and describe in user stories.

Table 2.4 Widely adopted XP practices

Practice	Description
Incremental planning/ user stories	There is no "grand plan" for the system. Instead, what needs to be implemented (the requirements) in each increment are established in discussions with a customer representative. The requirements are written as user stories. The stories to be included in a release are determined by the time available and their relative priority.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the previous release.
Test-driven development	Instead of writing code and then tests for that code, developers write the tests first. This helps clarify what the code should actually do and that there is always a "tested" version of the code available. An automated unit test framework is used to run the tests after every change. New code should not "break" code that has already been implemented.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system and a new version of the system is created. All unit tests from all developers are run automatically and must be successful before the new version of the system is accepted.
Refactoring	Refactoring means improving the structure, readability, efficiency, and security of a program. All developers are expected to refactor the code as soon as potential code improvements are found. This keeps the code simple and maintainable.

Practices such as having an on-site customer and collective ownership of code are good ideas. An on-site customer works with the team, proposes stories and tests, and learns about the product. However, the reality is that customers and surrogate customers such as product managers have many other things to do. It is difficult for them to find the time to be fully embedded in a development team.

Collective ownership discourages the individual ownership of code, but it has proved to be impractical in many companies. Specialists are needed for some types of code. Some people may work part-time on a project and so cannot participate in its "ownership." Some team members may be psychologically unsuited to this way of working and have no wish to "own" someone else's code.