

Introduction to System Engineering

Compendium

Table of Contents	Page
Craig Larman, "Applying UML and Patterns", Prentice Hall PTR, 3.ed.	
Chapter 5 Requirements, pp 54 – 57	2
Craig Larman, "Applying UML and Patterns", Prentice Hall PTR, 3.ed.	
Chapter 6 Use Cases, pp 61 – 100	7
James K. Peckol, "Embedded Systems Design, A Contemporary Design Tool", Wiley, ISBN 978-0-471-72180-2.	
Chapter 10 Hardware Test and Debug, pp 401 – 407	48
Stephen Biering-Sørensen, Finn Overgaard Hansen, Susanne Klim, Preben Thalund Madsen "Struktureret Program-Udvikling" (SPU), Teknisk Forlag, ISBN 87-571-1046-8.	
Vejledning i softwaretest, pp 171 – 207	57
Sandford Friedenthal, Alan Moore, Rich Steiner, "A Practical Guide to SysML", Morgan Kaufmann, ISBN 978-0-123-74379-4.	
Chapter 3 SysML Language Overview, pp 29 – 60	77
Stephen Biering-Sørensen, Finn Overgaard Hansen, Susanne Klim, Preben Thalund Madsen "Struktureret Program-Udvikling" (SPU), Teknisk Forlag, ISBN 87-571-1046-8.	
Vejledning i review, pp 215 – 236	110
Craig Larman, "Applying UML and Patterns", Prentice Hall PTR, 3.ed	
Chapter 9 Domain Models, pp 131 – 159	123
James K. Peckol, "Embedded Systems Design, A Contemporary Design Tool", Wiley, ISBN 978-0-471-72180-2.	
Chapter 9 System Design, pp 366 - 368 and pp 376 - 391	154
Frank Vahid/ Tony Givargis, "Embedded System Design, A Unified Hardware/Software Introduction", Wiley, ISBN 0-471-38678-2.	
Chapter 6 Interfacing, pp 137- 153	175
Chapter 6 Interfacing, pp 166-169	185
James K. Peckol, "Embedded Systems Design, A Contemporary Design Tool", Wiley, ISBN 978-0-471-72180-2.	
Chapter 16.7 Network Architecture, pp 627 - 637	188
Frank Vahid/ Tony Givargis, "Embedded System Design, A Unified Hardware/Software Introduction", Wiley, ISBN 0-471-38678-2.	
Chapter 1 Embedded Systems Overview, pp 1 – 11	201
Poul Staal Vinje, "Projektledelse af systemudvikling", Nyt Teknisk Forlag, ISBN 978-87-571-2457-5.	
Kapitel 5.8-5.9, 6.3, 10.1 – 10.7 (Udviklingsprocesser)	208
Poul Staal Vinje, "Projektledelse af systemudvikling", Nyt Teknisk Forlag, ISBN 978-87-571-2457-5.	
Kapitel 3.1-3.4, 3.7, 5.4, 5.5, 6.4, 6.5, 6.7 (Projektledelse)	247

Craig Larman

Applying UML and Patterns

Prentice Hall PTR, 3.ed.

Chapter 5, Requirements, pp 54 – 57

5.1 Definition: Requirements

Requirements are capabilities and conditions to which the system—and more broadly, the project—must conform [JBR99].

The UP promotes a set of best practices, one of which is *manage requirements*. This does not mean the waterfall attitude of attempting to fully define and stabilize the requirements in the first phase of a project before programming, but rather—in the context of inevitably changing and unclear stakeholder's wishes, this means—"a systematic approach to finding, documenting, organizing, and tracking the *changing* requirements of a system" [RUP].

In short, doing it iteratively and skillfully, and not being sloppy.

A prime challenge of requirements analysis is to find, communicate, and remember (that usually means write down) what is really needed, in a form that clearly speaks to the client and development team members.

5.2 Evolutionary vs. Waterfall Requirements

Notice the word *changing* in the definition of what it means to manage requirements. The UP embraces change in requirements as a fundamental driver on projects. That's incredibly important and at the heart of waterfall versus iterative and evolutionary thinking.

In the UP and other evolutionary methods (Scrum, XP, FDD, and so on), we start production-quality programming and testing long before most of the requirements have been analyzed or specified—perhaps when only 10% or 20% of the most architecturally significant, risky, and high-business-value requirements have been specified.

What are the process details? How to do partial, evolutionary requirements analysis combined with early design and programming, in iterations? See "How to do Iterative and Evolutionary Analysis and Design?" on page 25. It provides a brief description and a picture to help explain the process. See "Process: How to Work With Use Cases in Iterative Methods?" on page 95. It has more detailed discussion.

Caution!

If you find yourself on a so-called UP or iterative project that attempts to specify most or all of the requirements (use cases, and so forth) before starting to program and test, there is a profound misunderstanding—it is not a healthy UP or iterative project.

EVOLUTIONARY VS. WATERFALL REQUIREMENTS

In the 1960s and 1970s (when I started work as a developer) there was still a common speculative belief in the efficacy of full, early requirements analysis for software projects (i.e., the waterfall). Starting in the 1980s, there arose evidence this was unskillful and led to many failures; the old belief was rooted in the wrong paradigm of viewing a software project as similar to predictable mass manufacturing, with low change rates. But software is in the domain of new product development, with high change ranges and high degrees of novelty and discovery.

*change research
p. 24*

Recall the key statistic that, on average, 25% of the requirements change on software projects. Any method that therefore attempts to freeze or fully define requirements at the start is fundamentally flawed, based on a false assumption, and fighting or denying the inevitable change.

Underlining this point, for example, was a study of failure factors on 1,027 software projects [Thomas01]. The findings? Attempting waterfall practices (including detailed up-front requirements) was the single largest contributing factor for failure, being cited in 82% of the projects as the number one problem. To quote the conclusion:

... the approach of full requirements definition followed by a long gap before those requirements are delivered is no longer appropriate.

The high ranking of changing business requirements suggests that any assumption that there will be little significant change to requirements once they have been documented is fundamentally flawed, and that spending significant time and effort defining them to the maximum level is inappropriate.

Another relevant research result answers this question: When waterfall requirements analysis is attempted, how many of the prematurely early specified features are actually useful in the final software product? In a study [Johnson02] of thousands of projects, the results are quite revealing—45% of such features were never used, and an additional 19% were “rarely” used. See Figure 5.1. Almost 65% of the waterfall-specified features were of little or no value!

These results don’t imply that the solution is to start pounding away at the code near Day One of the project, and forget about requirements analysis or recording requirements. There is a middle way: iterative and evolutionary requirements analysis combined with early timeboxed iterative development and frequent stakeholder participation, evaluation, and feedback on partial results.

5 – EVOLUTIONARY REQUIREMENTS

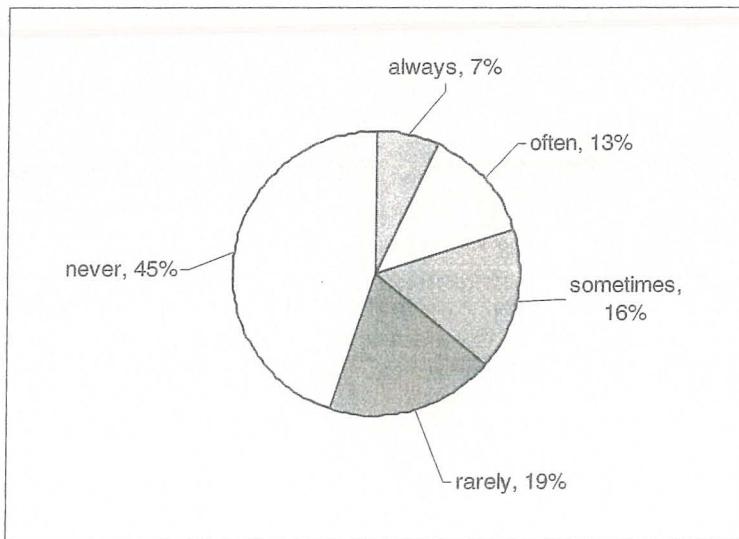


Figure 5.1 Actual use of waterfall-specified features.

5.3 What are Skillful Means to Find Requirements?

To review the UP best practice *manage requirements*:

...a systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system. [RUP]

Besides *changing*, the word *finding* is important; that is, the UP encourages skillful elicitation via techniques such as writing use cases with customers, requirements workshops that include both developers and customers, focus groups with proxy customers, and a demo of the results of each iteration to the customers, to solicit feedback.

The UP welcomes any requirements elicitation method that can add value and that increases user participation. Even the simple XP “story card” practice is acceptable on a UP project, if it can be made to work effectively (it requires the presence of a full-time customer-expert in the project room—an excellent practice but often difficult to achieve).

5.4 What are the Types and Categories of Requirements?

In the UP, requirements are categorized according to the **FURPS+** model

WHAT ARE THE TYPES AND CATEGORIES OF REQUIREMENTS?

[Grady92], a useful mnemonic with the following meaning:¹

- **Functional**—features, capabilities, security.
- **Usability**—human factors, help, documentation.
- **Reliability**—frequency of failure, recoverability, predictability.
- **Performance**—response times, throughput, accuracy, availability, resource usage.
- **Supportability**—adaptability, maintainability, internationalization, configurability.

The “+” in FURPS+ indicates ancillary and sub-factors, such as:

- **Implementation**—resource limitations, languages and tools, hardware, ...
- **Interface**—constraints imposed by interfacing with external systems.
- **Operations**—system management in its operational setting.
- **Packaging**—for example, a physical box.
- **Legal**—licensing and so forth.

It is helpful to use FURPS+ categories (or some categorization scheme) as a checklist for requirements coverage, to reduce the risk of not considering some important facet of the system.

Some of these requirements are collectively called the **quality attributes**, **quality requirements**, or the “-ilities” of a system. These include usability, reliability, performance, and supportability. In common usage, requirements are categorized as **functional** (behavioral) or **non-functional** (everything else); some dislike this broad generalization [BCK98], but it is very widely used.

architectural analysis p. 541

As we shall see when exploring architectural analysis, the quality attributes have a strong influence on the architecture of a system. For example, a high-performance, high-reliability requirement will influence the choice of software and hardware components, and their configuration.

1. There are several systems of requirements categorization and quality attributes published in books and by standards organizations, such as ISO 9126 (which is similar to the FURPS+ list), and several from the Software Engineering Institute (SEI); any can be used on a UP project.

Craig Larman

Applying UML and Patterns

Prentice Hall PTR, 3.ed.

Chapter 6 Use Cases, pp 61 – 100

Chapter

6

USE CASES

The indispensable first step to getting the things you want out of life: decide what you want.

—Ben Stein

Objectives

- Identify and write use cases.
- Use the brief, casual, and fully dressed formats, in an essential style.
- Apply tests to identify suitable use cases.
- Relate use case analysis to iterative development.

Introduction

*intermediate use
case topics p. 493*

Use cases are text stories, widely used to discover and record requirements. They influence many aspects of a project—including OOA/D—and will be input to many subsequent artifacts in the case studies. This chapter explores basic concepts, including how to write use cases and draw a UML use case diagram. This chapter also shows the value of analysis skill over knowing UML notation; the UML use case diagram is trivial to learn, but the many guidelines to identify and write good use cases take weeks—or longer—to fully digest.

What's Next?

Having introduced requirements, this chapter explores use cases for functional requirements. The next covers other requirements in the UP, including the Supplementary Specification for non-functional requirements.



6 – USE CASES

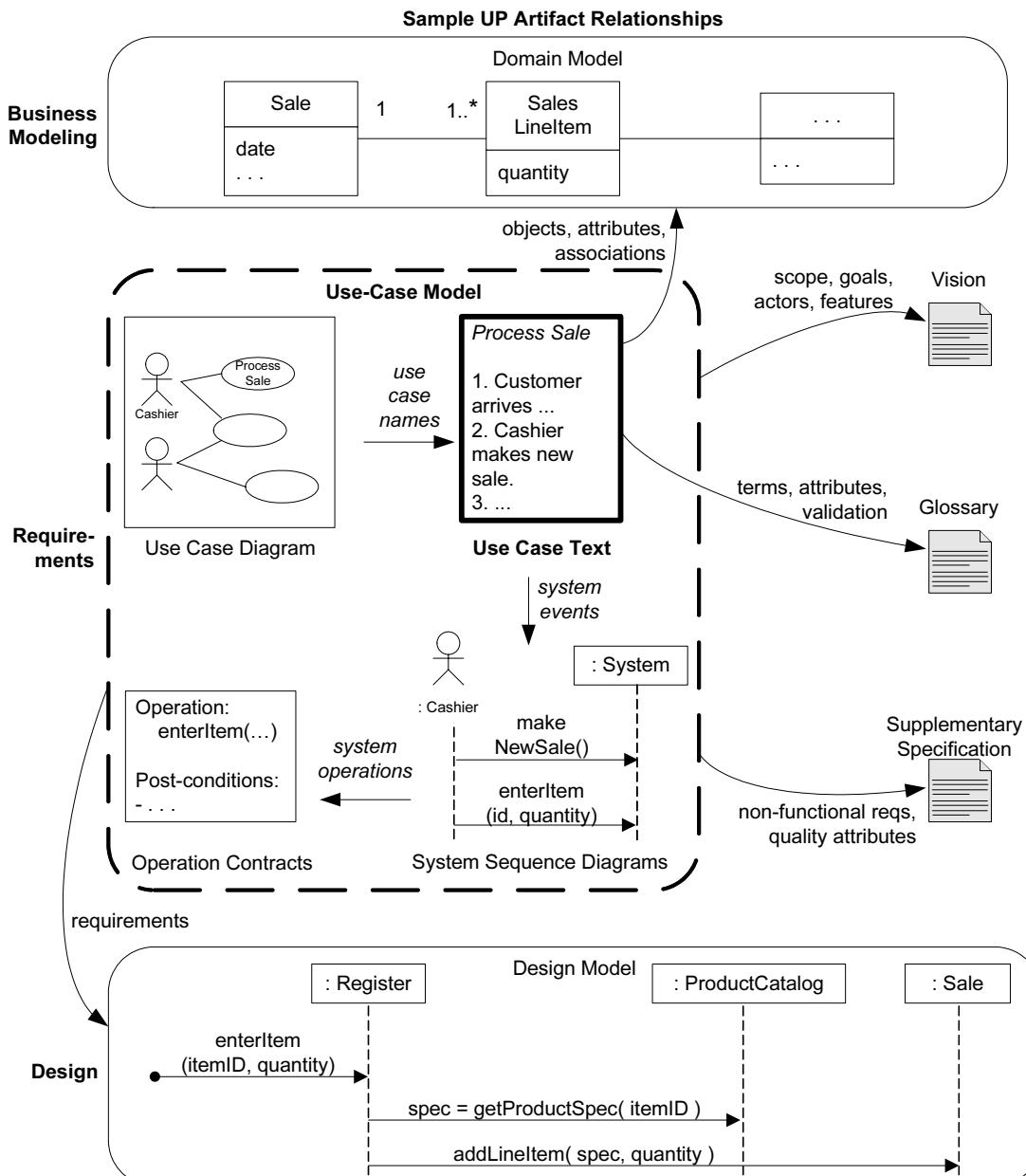


Figure 6.1 Sample UP artifact influence.

The influence of UP artifacts, with an emphasis on text use cases, is shown in Figure 6.1. High-level goals and use case diagrams are input to the creation of the use case text. The use cases can in turn influence many other analysis, design, implementation, project management, and test artifacts.

EXAMPLE

6.1 Example

Informally, use cases are *text stories* of some actor using a system to meet goals. Here is an example *brief format* use case:

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

UML use case diagrams p. 89

Notice that **use cases are not diagrams, they are text**. Focusing on secondary-value UML use case diagrams rather than the important use case text is a common mistake for use case novices.

Use cases often need to be more detailed or structured than this example, but the essence is discovering and recording functional requirements by writing stories of using a system to fulfill user goals; that is, *cases of use*.¹ It isn't supposed to be a difficult idea, although it's often difficult to discover what's needed and write it well.

6.2 Definition: What are Actors, Scenarios, and Use Cases?

First, some informal definitions: an **actor** is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

A **scenario** is a specific sequence of actions and interactions between actors and the system; it is also called a **use case instance**. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.

Informally then, a **use case** is a collection of related success and failure scenarios that describe an actor using a system to support a goal. For example, here is a *casual format* use case with alternate scenarios:

Handle Returns

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios:

1. The original term in Swedish literally translates as “usage case.”

6 – USE CASES

If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external accounting system, ...

Now that scenarios (use case instances) are defined, an alternate, but similar definition of a use case provided by the RUP will make better sense:

A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor [RUP].

6.3 Use Cases and the Use-Case Model

The UP defines the **Use-Case Model** within the Requirements discipline. Primarily, this is the set of all written use cases; it is a model of the system's functionality and environment.

**Use cases are text documents,
not diagrams, and use-case modeling is
primarily an act of writing text, not drawing diagrams.**

other UP requirements p. 101

The Use-Case Model is not the only requirement artifact in the UP. There are also the Supplementary Specification, Glossary, Vision, and Business Rules. These are all useful for requirements analysis, but secondary at this point.

UML use case diagram p. 89

The Use-Case Model may optionally include a UML use case diagram to show the names of use cases and actors, and their relationships. This gives a nice **context diagram** of a system and its environment. It also provides a quick way to list the use cases by name.

There is nothing object-oriented about use cases; we're not doing OO analysis when writing them. That's not a problem—use cases are broadly applicable, which increases their usefulness. That said, use cases are a key requirements input to classic OOA/D.

6.4 Motivation: Why Use Cases?

We have goals and want computers to help meet them, ranging from recording

DEFINITION: ARE USE CASES FUNCTIONAL REQUIREMENTS?

sales to playing games to estimating the flow of oil from future wells. Clever analysts have invented *many* ways to capture goals, but the best are simple and familiar. Why? This makes it easier—especially for customers—to contribute to their definition and review. That lowers the risk of missing the mark. This may seem like an off-hand comment, but it's important. Researchers have concocted complex analysis methods that they understand, but that send your average business person into a coma! Lack of user involvement in software projects is near the top of the list of reasons for project failure [Larman03], so anything that can help keep them involved is truly desirable.

*more motivation
p. 92*

Use cases are a good way to help keep it simple, and make it possible for domain experts or requirement donors to themselves write (or participate in writing) use cases.

Another value of use cases is that *they emphasize the user goals and perspective*; we ask the question “Who is using the system, what are their typical scenarios of use, and what are their goals?” This is a more user-centric emphasis compared to simply asking for a list of system features.

Much has been written about use cases, and though worthwhile, creative people often obscure a simple idea with layers of sophistication or over-complication. It is usually possible to spot a novice use-case modeler (or a serious Type-A analyst!) by an over-concern with secondary issues such as use case diagrams, use case relationships, use case packages, and so forth, rather than a focus on the hard work of simply *writing* the text stories.

That said, a strength of use cases is the ability to scale both up and down in terms of sophistication and formality.

6.5 Definition: Are Use Cases Functional Requirements?

FURPS+ p. 56

Use cases *are* requirements, primarily functional or behavioral requirements that indicate what the system will do. In terms of the FURPS+ requirements types, they emphasize the “F” (functional or behavioral), but can also be used for other types, especially when those other types strongly relate to a use case. In the UP—and many modern methods—use cases are the central mechanism that is recommended for their discovery and definition.

A related viewpoint is that a use case defines a *contract* of how a system will behave [Cockburn01].

To be clear: Use cases are indeed requirements (although not all requirements). Some think of requirements only as “the system shall do...” function or feature lists. Not so, and a key idea of use cases is to (usually) reduce the importance or use of detailed old-style feature lists and rather write use cases for the functional requirements. More on this point in a later section.

6 – USE CASES

6.6 Definition: What are Three Kinds of Actors?

An **actor** is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems.² Primary and supporting actors will appear in the action steps of the use case text. Actors are roles played not only by people, but by organizations, software, and machines. There are three kinds of external actors in relation to the SuD:

- **Primary actor**—has user goals fulfilled through using services of the SuD. For example, the cashier.
 - Why identify? To find user goals, which drive the use cases.
- **Supporting actor**—provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
 - Why identify? To clarify external interfaces and protocols.
- **Offstage actor**—has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
 - Why identify? To ensure that *all* necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

6.7 Notation: What are Three Common Use Case Formats?

Use cases can be written in different formats and levels of formality:

example p. 63

- **brief**—Terse one-paragraph summary, usually of the main success scenario. The prior *Process Sale* example was brief.
 - When? During early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.

example p. 63

- **casual**—Informal paragraph format. Multiple paragraphs that cover various scenarios. The prior *Handle Returns* example was casual.
 - When? As above.

2. This was a refinement and improvement to alternate definitions of actors, including those in early versions of the UML and UP [Cockburn97]. Older definitions inconsistently excluded the SuD as an actor, even when it called upon services of other systems. All entities may play multiple *roles*, including the SuD.

EXAMPLE: PROCESS SALE, FULLY DRESSED STYLE*example p. 68**more on timing of writing use cases p. 95*

- **fully dressed**—All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.
 - When? After many use cases have been identified and written in a brief format, then during the first requirements workshop a few (such as 10%) of the architecturally significant and high-value use cases are written in detail.

The following example is a fully dressed case for our NextGen case study.

6.8 Example: Process Sale, Fully Dressed Style

Fully dressed use cases show more detail and are structured; they dig deeper.

In iterative and evolutionary UP requirements analysis, 10% of the critical use cases would be written this way during the first requirements workshop. Then design and programming starts on the most architecturally significant use cases or scenarios from that 10% set.

Various format templates are available for detailed use cases. Probably the most widely used and shared format, since the early 1990s, is the template available on the Web at alistair.cockburn.us, created by Alistair Cockburn, the author of the most popular book and approach to use-case modeling. The following example illustrates this style.

First, here's the template:

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	“user-goal” or “subfunction”
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, and worth telling the reader?
Success Guarantee	What must be true on successful completion, and worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.

Main Success Scenario and Extensions are the two major sections

6 – USE CASES

Use Case Section	Comment
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Here's an example, based on the template.

Please note that this is the book's primary case study example of a detailed use case; it shows many common elements and issues.

It probably shows much more than you ever wanted to know about a POS system! But, it's for a real POS, and shows the ability of use cases to capture complex real-world requirements, and deeply branching scenarios.

Use Case UC1: Process Sale

Scope: NextGen POS application

Level: user goal

Primary Actor: Cashier

Stakeholders and Interests:

- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- Customer: Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.
- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- Manager: Wants to be able to quickly perform override operations, and easily debug Cashier problems.
- Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions: Cashier is identified and authenticated.

Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated.

Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Payment authorization approvals are recorded.

EXAMPLE: PROCESS SALE, FULLY DRESSED STYLE**Main Success Scenario (or Basic Flow):**

1. Customer arrives at POS checkout with goods and/or services to purchase.
 2. Cashier starts a new sale.
 3. Cashier enters item identifier.
 4. System records sale line item and presents item description, price, and running total.
Price calculated from a set of price rules.
- Cashier repeats steps 3-4 until indicates done.*
5. System presents total with taxes calculated.
 6. Cashier tells Customer the total, and asks for payment.
 7. Customer pays and System handles payment.
 8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
 9. System presents receipt.
 10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

- *a. At any time, Manager requests an override operation:
 1. System enters Manager-authorized mode.
 2. Manager or Cashier performs one Manager-mode operation. e.g., cash balance change, resume a suspended sale on another register, void a sale, etc.
 3. System reverts to Cashier-authorized mode.
- *b. At any time, System fails:

To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

 1. Cashier restarts System, logs in, and requests recovery of prior state.
 2. System reconstructs prior state.
 - 2a. System detects anomalies preventing recovery:
 1. System signals error to the Cashier, records the error, and enters a clean state.
 2. Cashier starts a new sale.
- 1a. Customer or Manager indicate to resume a suspended sale.
 1. Cashier performs resume operation, and enters the ID to retrieve the sale.
 2. System displays the state of the resumed sale, with subtotal.
 - 2a. Sale not found.
 1. System signals error to the Cashier.
 2. Cashier probably starts new sale and re-enters all items.
 3. Cashier continues with sale (probably entering more items or handling payment).
- 2-4a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples)
 1. Cashier verifies, and then enters tax-exempt status code.
 2. System records status (which it will use during tax calculations)
- 3a. Invalid item ID (not found in system):
 1. System signals error and rejects entry.
 2. Cashier responds to the error:
 - 2a. There is a human-readable item ID (e.g., a numeric UPC):
 1. Cashier manually enters the item ID.
 2. System displays description and price.
 - 2a. Invalid item ID: System signals error. Cashier tries alternate method.
- 2b. There is no item ID, but there is a price on the tag:
 1. Cashier asks Manager to perform an override operation.

6 – USE CASES

2. Managers performs override.
3. Cashier indicates manual price entry, enters price, and requests standard taxation for this amount (because there is no product information, the tax engine can't otherwise deduce how to tax it)
- 2c. Cashier performs Find Product Help to obtain true item ID and price.
- 2d. Otherwise, Cashier asks an employee for the true item ID or price, and does either manual ID or manual price entry (see above).
- 3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
 1. Cashier can enter item category identifier and the quantity.
- 3c. Item requires manual category and price entry (such as flowers or cards with a price on them):
 1. Cashier enters special manual category code, plus the price.
- 3-6a: Customer asks Cashier to remove (i.e., void) an item from the purchase:

This is only legal if the item value is less than the void limit for Cashiers, otherwise a Manager override is needed.

 1. Cashier enters item identifier for removal from sale.
 2. System removes item and displays updated running total.
 - 2a. Item price exceeds void limit for Cashiers:
 1. System signals error, and suggests Manager override.
 2. Cashier requests Manager override, gets it, and repeats operation.
- 3-6b. Customer tells Cashier to cancel sale:
 1. Cashier cancels sale on System.
- 3-6c. Cashier suspends the sale:
 1. System records sale so that it is available for retrieval on any POS register.
 2. System presents a “suspend receipt” that includes the line items, and a sale ID used to retrieve and resume the sale.
- 4a. The system supplied item price is not wanted (e.g., Customer complained about something and is offered a lower price):
 1. Cashier requests approval from Manager.
 2. Manager performs override operation.
 3. Cashier enters manual override price.
 4. System presents new price.
- 5a. System detects failure to communicate with external tax calculation system service:
 1. System restarts the service on the POS node, and continues.
 - 1a. System detects that the service does not restart.
 1. System signals error.
 2. Cashier may manually calculate and enter the tax, or cancel the sale.
- 5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):
 1. Cashier signals discount request.
 2. Cashier enters Customer identification.
 3. System presents discount total, based on discount rules.
- 5c. Customer says they have credit in their account, to apply to the sale:
 1. Cashier signals credit request.
 2. Cashier enters Customer identification.
 3. System applies credit up to price=0, and reduces remaining credit.
- 6a. Customer says they intended to pay by cash but don't have enough cash:
 1. Cashier asks for alternate payment method.
 - 1a. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

EXAMPLE: PROCESS SALE, FULLY DRESSED STYLE

- 7a. Paying by cash:
1. Cashier enters the cash amount tendered.
 2. System presents the balance due, and releases the cash drawer.
 3. Cashier deposits cash tendered and returns balance in cash to Customer.
 4. System records the cash payment.
- 7b. Paying by credit:
1. Customer enters their credit account information.
 2. System displays their payment for verification.
 3. Cashier confirms.
 - 3a. Cashier cancels payment step:
 1. System reverts to "item entry" mode.
 4. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
 - 4a. System detects failure to collaborate with external system:
 1. System signals error to Cashier.
 2. Cashier asks Customer for alternate payment.
 5. System receives payment approval, signals approval to Cashier, and releases cash drawer (to insert signed credit payment receipt).
 - 5a. System receives payment denial:
 1. System signals denial to Cashier.
 2. Cashier asks Customer for alternate payment.
 - 5b. Timeout waiting for response.
 1. System signals timeout to Cashier.
 2. Cashier may try again, or ask Customer for alternate payment.
 6. System records the credit payment, which includes the payment approval.
 7. System presents credit payment signature input mechanism.
 8. Cashier asks Customer for a credit payment signature. Customer enters signature.
 9. If signature on paper receipt, Cashier places receipt in cash drawer and closes it.
- 7c. Paying by check...
- 7d. Paying by debit...
- 7e. Cashier cancels payment step:
 1. System reverts to "item entry" mode.
- 7f. Customer presents coupons:
 1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.
 - 1a. Coupon entered is not for any purchased item:
 1. System signals error to Cashier.
- 9a. There are product rebates:
 1. System presents the rebate forms and rebate receipts for each item with a rebate.
- 9b. Customer requests gift receipt (no prices visible):
 1. Cashier requests gift receipt and System presents it.
- 9c. Printer out of paper:
 1. If System can detect the fault, will signal the problem.
 2. Cashier replaces paper.
 3. Cashier requests another receipt.

6 – USE CASES**Special Requirements:**

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 and 7.
- ...

Technology and Data Variations List:

- *a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.
- 3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

This use case is illustrative rather than exhaustive (although it is based on a real POS system's requirements—developed with an OO design in Java). Nevertheless, there is enough detail and complexity here to offer a realistic sense that a fully dressed use case can record many requirement details. This example will serve well as a model for many use case problems.

6.9 What do the Sections Mean?

Preface Elements

Scope

The scope bounds the system (or systems) under design. Typically, a use case describes use of one software (or hardware plus software) system; in this case it is known as a **system use case**. At a broader scope, use cases can also describe how a business is used by its customers and partners. Such an enterprise-level

WHAT DO THE SECTIONS MEAN?

process description is called a **business use case** and is a good example of the wide applicability of use cases, but they aren't covered in this introductory book.

Level

EBP p. 88

see the use case “include” relationship for more on subfunction use cases p. 494

In Cockburn's system, use cases are classified as at the user-goal level or the subfunction level, among others. A **user-goal level** use case is the common kind that describes the scenarios to fulfill the goals of a primary actor to get work done; it roughly corresponds to an **elementary business process** (EBP) in business process engineering. A **subfunction-level** use case describes substeps required to support a user goal, and is usually created to factor out duplicate substeps shared by several regular use cases (to avoid duplicating common text); an example is the subfunction use case *Pay by Credit*, which could be shared by many regular use cases.

Primary Actor

The principal actor that calls upon system services to fulfill a goal.

Stakeholders and Interests List—Important!

This list is more important and practical than may appear at first glance. It suggests and bounds what the system must do. To quote:

The [system] operates a contract between stakeholders, with the use cases detailing the behavioral parts of that contract...The use case, as the contract for behavior, captures *all and only* the behaviors related to satisfying the stakeholders' interests [Cockburn01].

This answers the question: What should be in the use case? The answer is: That which satisfies all the stakeholders' interests. In addition, by starting with the stakeholders and their interests before writing the remainder of the use case, we have a method to remind us what the more detailed responsibilities of the system should be. For example, would I have identified a responsibility for salesperson commission handling if I had not first listed the salesperson stakeholder and their interests? Hopefully eventually, but perhaps I would have missed it during the first analysis session. The stakeholder interest viewpoint provides a thorough and methodical procedure for discovering and recording all the required behaviors.

Stakeholders and Interests:

- Cashier: Wants accurate, fast entry and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- ...

6 – USE CASES

Preconditions and Success Guarantees (Postconditions)

First, don't bother with a precondition or success guarantee unless you are stating something non-obvious and noteworthy, to help the reader gain insight. Don't add useless noise to requirements documents.

Preconditions state what *must always* be true before a scenario is begun in the use case. Preconditions are *not* tested within the use case; rather, they are conditions that are assumed to be true. Typically, a precondition implies a scenario of another use case, such as logging in, that has successfully completed. Note that there are conditions that must be true, but are not worth writing, such as “the system has power.” Preconditions communicate noteworthy assumptions that the writer thinks readers should be alerted to.

Success guarantees (or postconditions) state what must be true on successful completion of the use case—either the main success scenario or some alternate path. The guarantee should meet the needs of all stakeholders.

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated.

Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Main Success Scenario and Steps (or Basic Flow)

This has also been called the “happy path” scenario, or the more prosaic “Basic Flow” or “Typical Flow.” It describes a typical success path that satisfies the interests of the stakeholders. Note that it often does *not* include any conditions or branching. Although not wrong or illegal, it is arguably more comprehensible and extendible to be very consistent and defer all conditional handling to the Extensions section.

Guideline

Defer all conditional and branching statements to the Extensions section.

The scenario records the steps, of which there are three kinds:

1. An interaction between actors.³
2. A validation (usually by the system).
3. A state change by the system (for example, recording or modifying something).

3. Note that the system under discussion itself should be considered an actor when it plays an actor role collaborating with other systems.

WHAT DO THE SECTIONS MEAN?

Step one of a use case does not always fall into this classification, but indicates the trigger event that starts the scenario.

It is a common idiom to always capitalize the actors' names for ease of identification. Observe also the idiom that is used to indicate repetition.

Main Success Scenario:

1. Customer arrives at a POS checkout with items to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. ...
- Cashier repeats steps 3-4 until indicates done.
5. ...

Extensions (or Alternate Flows)

Extensions are important and normally comprise the majority of the text. They indicate all the other scenarios or branches, both success and failure. Observe in the fully dressed example that the Extensions section was considerably longer and more complex than the Main Success Scenario section; this is common.

In thorough use case writing, the combination of the happy path and extension scenarios should satisfy "nearly" all the interests of the stakeholders. This point is qualified, because some interests may best be captured as non-functional requirements expressed in the Supplementary Specification rather than the use cases. For example, the customer's interest for a visible display of descriptions and prices is a usability requirement.

Extension scenarios are branches from the main success scenario, and so can be notated with respect to its steps 1...N. For example, at Step 3 of the main success scenario there may be an invalid item identifier, either because it was incorrectly entered or unknown to the system. An extension is labeled "3a"; it first identifies the condition and then the response. Alternate extensions at Step 3 are labeled "3b" and so forth.

Extensions:

- 3a. Invalid identifier:
 - 1. System signals error and rejects entry.
- 3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
 - 1. Cashier can enter item category identifier and the quantity.

An extension has two parts: the condition and the handling.

Guideline: When possible, write the condition as something that can be detected by the system or an actor. To contrast:

- 5a. System detects failure to communicate with external tax calculation system service:
- 5a. External tax calculation system not working:

6 – USE CASES

The former style is preferred because this is something the system can detect; the latter is an inference.

Extension handling can be summarized in one step, or include a sequence, as in this example, which also illustrates notation to indicate that a condition can arise within a range of steps:

- 3-6a: Customer asks Cashier to remove an item from the purchase:
1. Cashier enters the item identifier for removal from the sale.
 2. System displays updated running total.

At the end of extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (such as by halting the system).

Sometimes, a particular extension point is quite complex, as in the “paying by credit” extension. This can be a motivation to express the extension as a separate use case.

This extension example also demonstrates the notation to express failures within extensions.

- 7b. Paying by credit:
1. Customer enters their credit account information.
 2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
 - 2a. System detects failure to collaborate with external system:
 1. System signals error to Cashier.
 2. Cashier asks Customer for alternate payment.

If it is desirable to describe an extension condition as possible during any (or at least most) steps, the labels *a, *b, ..., can be used.

- *a. At any time, System crashes:
 In order to support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered at any step in the scenario.
1. Cashier restarts the System, logs in, and requests recovery of prior state.
 2. System reconstructs prior state.

Performing Another Use Case Scenario

Sometimes, a use case branches to perform another use case scenario. For example, the story *Find Product Help* (to show product details, such as description, price, a picture or video, and so on) is a distinct use case that is sometimes performed while within *Process Sale* (usually when the item ID can't be found). In

WHAT DO THE SECTIONS MEAN?

Cockburn notation, performing this second use case is shown with underlining, as this example shows:

-
- 3a. Invalid item ID (not found in system):
 1. System signals error and rejects entry.
 2. Cashier responds to the error:
 2a.
 2c. Cashier performs Find Product Help to obtain true item ID and price.

Assuming, as usual, that the use cases are written with a hyperlinking tool, then clicking on this underlined use case name will display its text.

Special Requirements

If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
 - Credit authorization response within 30 seconds 90% of the time.
 - Language internationalization on the text displayed.
 - Pluggable business rules to be insertable at steps 2 and 6.

Recording these with the use case is classic UP advice, and a reasonable location when *first* writing the use case. However, many practitioners find it useful to ultimately move and consolidate all non-functional requirements in the Supplementary Specification, for content management, comprehension, and readability, because these requirements usually have to be considered as a whole during architectural analysis.

Technology and Data Variations List

Often there are technical variations in *how* something must be done, but not what, and it is noteworthy to record this in the use case. A common example is a technical constraint imposed by a stakeholder regarding input or output technologies. For example, a stakeholder might say, "The POS system must support credit account input using a card reader and the keyboard." Note that these are examples of early design decisions or constraints; in general, it is skillful to avoid premature design decisions, but sometimes they are obvious or unavoidable, especially concerning input/output technologies.

It is also necessary to understand variations in data schemes, such as using UPCs or EANs for item identifiers, encoded in bar code symbology.

6 – USE CASES**Congratulations: Use Cases are Written and Wrong (!)**

The NextGen POS team is writing a few use cases in multiple short requirements workshops, in parallel with a series of short timeboxed development iterations that involve production-quality programming and testing. The team is incrementally adding to the use case set, and refining and adapting based on feedback from early programming, tests, and demos. Subject matter experts, cashiers, and developers actively participate in requirements analysis.

That's a good evolutionary analysis process—rather than the waterfall—but a dose of “requirements realism” is still needed. Written specifications and other models give the *illusion* of correctness, but models lie (unintentionally). Only code and tests reveals the truth of what's really wanted and works.

The use cases, UML diagrams, and so forth won't be perfect—guaranteed. They will lack critical information and contain wrong statements. The solution is not the waterfall attitude of trying to record specifications near-perfect and complete at the start—although of course we do the best we can in the time available, and should learn and apply great requirements practices. But it will never be enough.

This isn't a call to rush to coding without any analysis or modeling. There is a middle way, between the waterfall and ad hoc programming: iterative and evolutionary development. In this approach the use cases and other models are incrementally refined, verified, and clarified through early programming and testing.

You know you're on the wrong path if the team tries to write in detail all or most of the use cases before beginning the first development iteration—or the opposite.

This list is the place to record such variations. It is also useful to record variations in the data that may be captured at a particular step.

Technology and Data Variations List:

- 3a. Item identifier entered by laser scanner or keyboard.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

6.10 Notation: Are There Other Formats? A Two-Column Variation

Some prefer the two-column or conversational format, which emphasizes the interaction between the actors and the system. It was first proposed by Rebecca Wirfs-Brock in [Wirfs-Brock93], and is also promoted by Constantine and Lockwood to aid usability analysis and engineering [CL99]. Here is the same content using the two-column format:

NOTATION: ARE THERE OTHER FORMATS? A TWO-COLUMN VARIATION**Use Case UC1: Process Sale****Primary Actor:** ...

... as before ...

Main Success Scenario:

Actor Action (or Intention)

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.

Cashier repeats steps 3-4 until indicates done.

6. Cashier tells Customer the total, and asks for payment.
7. Customer pays.

...

System Responsibility

4. Records each sale line item and presents item description and running total.
5. Presents total with taxes calculated.
8. Handles payment.
9. Logs the completed sale and sends information to the external accounting (for all accounting and commissions) and inventory systems (to update inventory). System presents receipt.

...

The Best Format?

There isn't one best format; some prefer the one-column style, some the two-column. Sections may be added and removed; heading names may change. None of this is particularly important; the key thing is to write the details of the main success scenario and its extensions, in some form. [Cockburn01] summarizes many usable formats.

Personal Practice

This is my practice, not a recommendation. For some years, I used the two-column format because of its clear visual separation in the conversation. However, I have reverted to a one-column style as it is more compact and easier to format, and the slight value of the visually separated conversation does not for me outweigh these benefits. I find it still simple to visually identify the different parties in the conversation (Customer, System, ...) if each party and the System responses are usually allocated to their own steps.

6 – USE CASES

6.11 Guideline: Write in an Essential UI-Free Style

New and Improved! The Case for Fingerprinting

During a requirements workshop, the cashier may say one of his goals is to “log in.” The cashier was probably thinking of a GUI, dialog box, user ID, and password. This is a mechanism to achieve a goal, rather than the goal itself. By investigating up the goal hierarchy (“What is the goal of that goal?”), the system analyst arrives at a mechanism-independent goal: “identify myself and get authenticated,” or an even higher goal: “prevent theft ...”.

This *root-goal* discovery process can open up the vision to new and improved solutions. For example, keyboards and mice with biometric readers, usually for a fingerprint, are now common and inexpensive. If the goal is “identification and authentication” why not make it easy and fast using a biometric reader on the keyboard? But properly answering that question involves some usability analysis work as well. Are their fingers covered in grease? Do they have fingers?

Essential Style Writing

This idea has been summarized in various use case guidelines as “keep the user interface out; focus on intent” [Cockburn01]. Its motivation and notation has been more fully explored by Larry Constantine in the context of creating better user interfaces (UIs) and doing usability engineering [Constantine94, CL99]. Constantine calls the writing style **essential** when it avoids UI details and focuses on the real user intent.⁴

In an essential writing style, the narrative is expressed at the level of the user’s *intentions* and system’s *responsibilities* rather than their concrete actions. They remain free of technology and mechanism details, especially those related to the UI.

Guideline

Write use cases in an essential style; keep the user interface out and focus on actor intent.

All of the previous example use cases in this chapter, such as *Process Sale*, were written aiming towards an essential style.

4. The term comes from “essential models” in *Essential Systems Analysis* [MP84].

GUIDELINE: WRITE TERSE USE CASES*Contrasting Examples***Essential Style**

Assume that the *Manage Users* use case requires identification and authentication:

-
- ...
 - 1. Administrator identifies self.
 - 2. System authenticates identity.
 - 3. ...

The design solution to these intentions and responsibilities is wide open: biometric readers, graphical user interfaces (GUIs), and so forth.

Concrete Style—Avoid During Early Requirements Work

In contrast, there is a **concrete use case** style. In this style, user interface decisions are embedded in the use case text. The text may even show window screen shots, discuss window navigation, GUI widget manipulation and so forth. For example:

-
- ...
 - 1. Adminstrator enters ID and password in dialog box (see Picture 3).
 - 2. System authenticates Administrator.
 - 3. System displays the “edit users” window (see Picture 4).
 - 4. ...

These concrete use cases may be useful as an aid to concrete or detailed GUI design work during a later step, but they are not suitable during the early requirements analysis work. During early requirements work, “keep the user interface out—focus on intent.”

6.12 Guideline: Write Terse Use Cases

Do you like to read lots of requirements? I didn’t think so. So, write terse use cases. Delete “noise” words. Even small changes add up, such as “System authenticates...” rather than “*The System authenticates...*”

6.13 Guideline: Write Black-Box Use Cases

Black-box use cases are the most common and recommended kind; they do not describe the internal workings of the system, its components, or design. Rather, the system is described as having *responsibilities*, which is a common unifying

6 – USE CASES

metaphorical theme in object-oriented thinking—software elements have responsibilities and collaborate with other elements that have responsibilities.

By defining system responsibilities with black-box use cases, one can specify *what* the system must do (the behavior or functional requirements) without deciding *how* it will do it (the design). Indeed, the definition of “analysis” versus “design” is sometimes summarized as “what” versus “how.” This is an important theme in good software development: During requirements analysis avoid making “how” decisions, and specify the external behavior for the system, as a black box. Later, during design, create a solution that meets the specification.

Black-box style	Not
The system records the sale.	The system writes the sale to a database. ...or (even worse): The system generates a SQL INSERT statement for the sale...

6.14 Guideline: Take an Actor and Actor-Goal Perspective

Here's the RUP use case definition, from the use case founder Ivar Jacobson:

A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.

The phrase “*an observable result of value to a particular actor*” is a subtle but important concept that Jacobson considers critical, because it stresses two attitudes during requirements analysis:

- Write requirements focusing on the users or actors of a system, asking about their goals and typical situations.
- Focus on understanding what the actor considers a valuable result.

function lists p. 92

Perhaps it seems obvious to stress providing observable user value and focusing on users' typical goals, but the software industry is littered with failed projects that did not deliver what people really needed. The old feature and function list approach to capturing requirements can contribute to that negative outcome because it did not encourage asking who is using the product, and what provides value.

6.15 Guideline: How to Find Use Cases

Use cases are defined to satisfy the goals of the primary actors. Hence, the basic

GUIDELINE: HOW TO FIND USE CASES

procedure is:

1. Choose the system boundary. Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
2. Identify the primary actors—those that have goals fulfilled through using services of the system.
3. Identify the goals for each primary actor.
4. Define use cases that satisfy user goals; name them according to their goal. Usually, user-goal level use cases will be one-to-one with user goals, but there is at least one exception, as will be examined.

Of course, in iterative and evolutionary development, not all goals or use cases will be fully or correctly identified near the start. It's an evolving discovery.

Step 1: Choose the System Boundary

For this case study, the POS system itself is the system under design; everything outside of it is outside the system boundary, including the cashier, payment authorization service, and so on.

If the definition of the boundary of the system under design is not clear, it can be clarified by further definition of what is outside—the external primary and supporting actors. Once the external actors are identified, the boundary becomes clearer. For example, is the complete responsibility for payment authorization within the system boundary? No, there is an external payment authorization service actor.

Steps 2 and 3: Find Primary Actors and Goals

It is artificial to strictly linearize the identification of primary actors before user goals; in a requirements workshop, people brainstorm and generate a mixture of both. Sometimes, goals reveal the actors, or vice versa.

Guideline: Brainstorm the primary actors first, as this sets up the framework for further investigation.

Are There Questions to Help Find Actors and Goals?

In addition to obvious primary actors and goals, the following questions help identify others that may be missed:

- | | |
|--|---|
| Who starts and stops the system? | Who does system administration? |
| Who does user and security management? | Is "time" an actor because the system does something in response to a time event? |

6 – USE CASES

- | | |
|---|--|
| <p>Is there a monitoring process that restarts the system if it fails?</p> <p>How are software updates handled? Push or pull update?</p> <p>In addition to <i>human</i> primary actors, are there any external software or robotic systems that call upon services of the system?</p> | <p>Who evaluates system activity or performance?</p> <p>Who evaluates logs? Are they remotely retrieved?</p> <p>Who gets notified when there are errors or failures?</p> |
|---|--|

How to Organize the Actors and Goals?

There are at least two approaches:

use case diagrams
p. 89

1. As you discover the results, draw them in a use case diagram, naming the goals as use cases.
2. Write an actor-goal list first, review and refine it, and then draw the use case diagram.

If you create an actor-goal list, then in terms of UP artifacts it may be a section in the Vision artifact.

For example:

Actor	Goal	Actor	Goal
Cashier	process sales process rentals handle returns cash in cash out ...	System Administrator	add users modify users delete users manage security manage system tables ...
Manager	start up shut down ...	Sales Activity System	analyze sales and performance data
...

The Sales Activity System is a remote application that will frequently request sales data from each POS node in the network.

Why Ask About Actor Goals Rather Than Use Cases?

Actors have goals and use applications to help satisfy them. The viewpoint of use case modeling is to find these actors and their goals, and create solutions that produce a result of value. This is slight shift in emphasis for the use case

GUIDELINE: HOW TO FIND USE CASES

modeler. Rather than asking “What are the tasks?”, one starts by asking: “Who uses the system and what are their goals?” In fact, the name of a use case for a user goal should reflect its name, to emphasize this viewpoint—Goal: capture or process a sale; use case: *Process Sale*.

Thus, here is a key idea regarding investigating requirements and use cases:

Imagine we are together in a requirements workshop. We could ask either:

- “What do you do?” (roughly a task-oriented question) or,
- “What are your goals whose results have measurable value?”

Prefer the second question.

Answers to the first question are more likely to reflect current solutions and procedures, and the complications associated with them.

Answers to the second question, especially combined with an investigation to move higher up the goal hierarchy (“what is the root goal?”) open up the vision for new and improved solutions, focus on adding business value, and get to the heart of what the stakeholders want from the system.

Is the Cashier or Customer the Primary Actor?

Why is the cashier, and not the customer, a primary actor in the use case *Process Sale*?

The answer depends on the system boundary of the system under design, and who we are primarily designing the system for, as illustrated in Figure 6.2. If the enterprise or checkout service is viewed as an aggregate system, the customer *is* a primary actor, with the goal of getting goods or services and leaving. However, from the viewpoint of just the POS system (which is the choice of system boundary for this case study), the system services the goal of a trained cashier (and the store) to process the customer’s sale. This assumes a traditional checkout environment with a cashier, although there are an increasing number of self-checkout POS systems in operation for direct use by customers.

6 – USE CASES

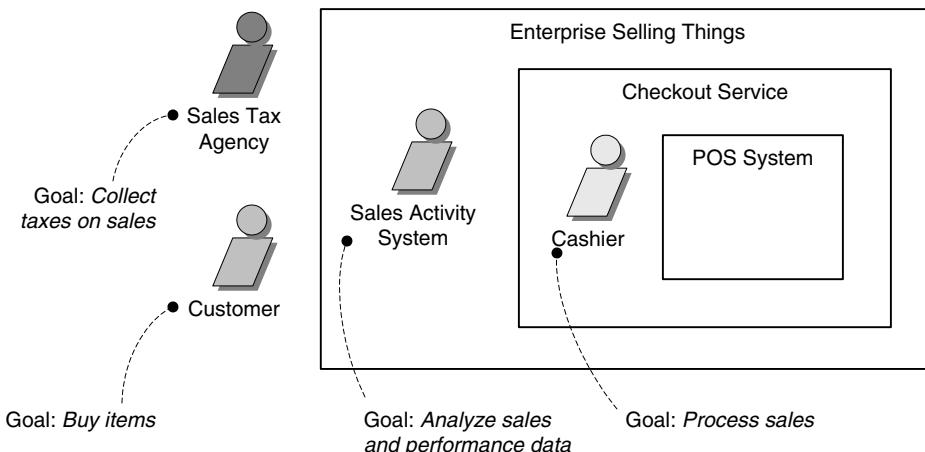


Figure 6.2 Primary actors and goals at different system boundaries.

The customer *is* an actor, but in the context of the NextGen POS, not a primary actor; rather, the cashier is the primary actor because the system is being designed to primarily serve the trained cashier’s “power user” goals (to quickly process a sale, look up prices, etc.). The system does not have a UI and functionality that could equally be used by the customer or cashier. Rather, it is optimized to meet the needs and training of a cashier. A customer in front of the POS terminal wouldn’t know how to use it effectively. In other words, it was designed for the cashier, not the customer, and so the cashier is not just a proxy for the customer.

On the other hand, consider a ticket-buying website that is identical for a customer to use directly or a phone agent to use, when a customer calls in. In this case, the agent is simply a proxy for the customer—the system is not designed to especially meet the unique goals of the agent. Then, showing the customer rather than the phone agent as the primary actor is correct.

Other Ways to Find Actors and Goals? Event Analysis

Another approach to aid in finding actors, goals, and use cases is to identify external events. What are they, where from, and why? Often, a group of events belong to the same use case. For example:

External Event	From Actor	Goal/Use Case
enter sale line item	Cashier	process a sale

GUIDELINE: WHAT TESTS CAN HELP FIND USEFUL USE CASES?

External Event	From Actor	Goal/Use Case
enter payment	Cashier or Customer	process a sale
...		

Step 4: Define Use Cases

In general, define one use case for each user goal. Name the use case similar to the user goal—for example, Goal: process a sale; Use Case: *Process Sale*.

Start the name of use cases with a verb.

A common exception to one use case per goal is to collapse CRUD (create, retrieve, update, delete) separate goals into one CRUD use case, idiomatically called *Manage <X>*. For example, the goals “edit user,” “delete user,” and so forth are all satisfied by the *Manage Users* use case.

6.16 Guideline: What Tests Can Help Find Useful Use Cases?

Which of these is a valid use case?

- Negotiate a Supplier Contract
- Handle Returns
- Log In
- Move Piece on Game Board

An argument can be made that all of these are use cases *at different levels*, depending on the system boundary, actors, and goals.

But rather than asking in general, “What is a valid use case?”, a more practical question is: “What is a useful level to express use cases for application requirements analysis?” There are several rules of thumb, including:

- The Boss Test
- The EBP Test
- The Size Test

The Boss Test

Your boss asks, “What have you been doing all day?” You reply: “Logging in!” Is your boss happy?

6 – USE CASES

If not, the use case fails the Boss Test, which implies it is not strongly related to achieving results of measurable value. It may be a use case at some low goal level, but not the desirable level of focus for requirements analysis.

That doesn't mean to always ignore boss-test-failing use cases. User authentication may fail the boss test, but may be important and difficult.

The EBP Test

An **Elementary Business Process (EBP)** is a term from the business process engineering field,⁵ defined as:

A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state, e.g., Approve Credit or Price Order [original source lost].

Focus on use cases that reflect EBPs.

The EBP Test is similar to the Boss Test, especially in terms of the measurable business value qualification.

The definition can be taken too literally: Does a use case fail as an EBP if two people are required, or if a person has to walk around? Probably not, but the feel of the definition is about right. It's not a single small step like "delete a line item" or "print the document." Rather, the main success scenario is probably five or ten steps. It doesn't take days and multiple sessions, like "negotiate a supplier contract"; it is a task done during a single session. It is probably between a few minutes and an hour in length. As with the UP's definition, it emphasizes adding observable or measurable business value, and it comes to a resolution in which the system and data are in a stable and consistent state.

The Size Test

A use case is very seldom a single action or step; rather, a use case typically contains many steps, and in the fully dressed format will often require 3–10 pages of text. A common mistake in use case modeling is to define just a single step within a series of related steps as a use case by itself, such as defining a use case called *Enter an Item ID*. You can see a hint of the error by its small size—the use case name will wrongly suggest just one step within a larger series of steps, and if you imagine the length of its fully dressed text, it would be extremely short.

5. EBP is similar to the term **user task** in usability engineering, although the meaning is less strict in that domain.

APPLYING UML: USE CASE DIAGRAMS

Example: Applying the Tests

- Negotiate a Supplier Contract
 - Much broader and longer than an EBP. Could be modeled as a *business* use case, rather than a system use case.
- Handle Returns
 - OK with the boss. Seems like an EBP. Size is good.
- Log In
 - Boss not happy if this is all you do all day!
- Move Piece on Game Board
 - Single step—fails the size test.

Reasonable Violations of the Tests

Although the majority of use cases identified and analyzed for an application should satisfy the tests, exceptions are common.

see the use case “include” relationship for more on linking subfunction use cases p. 494

It is sometimes useful to write separate subfunction-level use cases representing subtasks or steps within a regular EBP-level use case. For example, a subtask or extension such as “paying by credit” may be repeated in several base use cases. If so, it is desirable to separate this into its own use case, even though it does not really satisfy the EBP and size tests, and link it to several base use cases, to avoid duplication of the text.

Authenticate User may not pass the Boss test, but be complex enough to warrant careful analysis, such as for a “single sign-on” feature.

6.17 Applying UML: Use Case Diagrams

The UML provides use case diagram notation to illustrate the names of use cases and actors, and the relationships between them (see Figure 6.3).⁶

Use case diagrams and use case relationships are secondary in use case work.
Use cases are text documents. Doing use case work means to write text.

A common sign of a novice (or academic) use case modeler is a preoccupation with use case diagrams and use case relationships, rather than writing text.

6. “Cash In” is the act of a cashier arriving with a drawer insert with cash, logging in, and recording the cash amount in the drawer insert.

6 – USE CASES

World-class use case experts such as Fowler and Cockburn, among others, downplay use case diagrams and use case relationships, and instead focus on writing. With that as a caveat, a simple use case diagram provides a succinct visual context diagram for the system, illustrating the external actors and how they use the system.

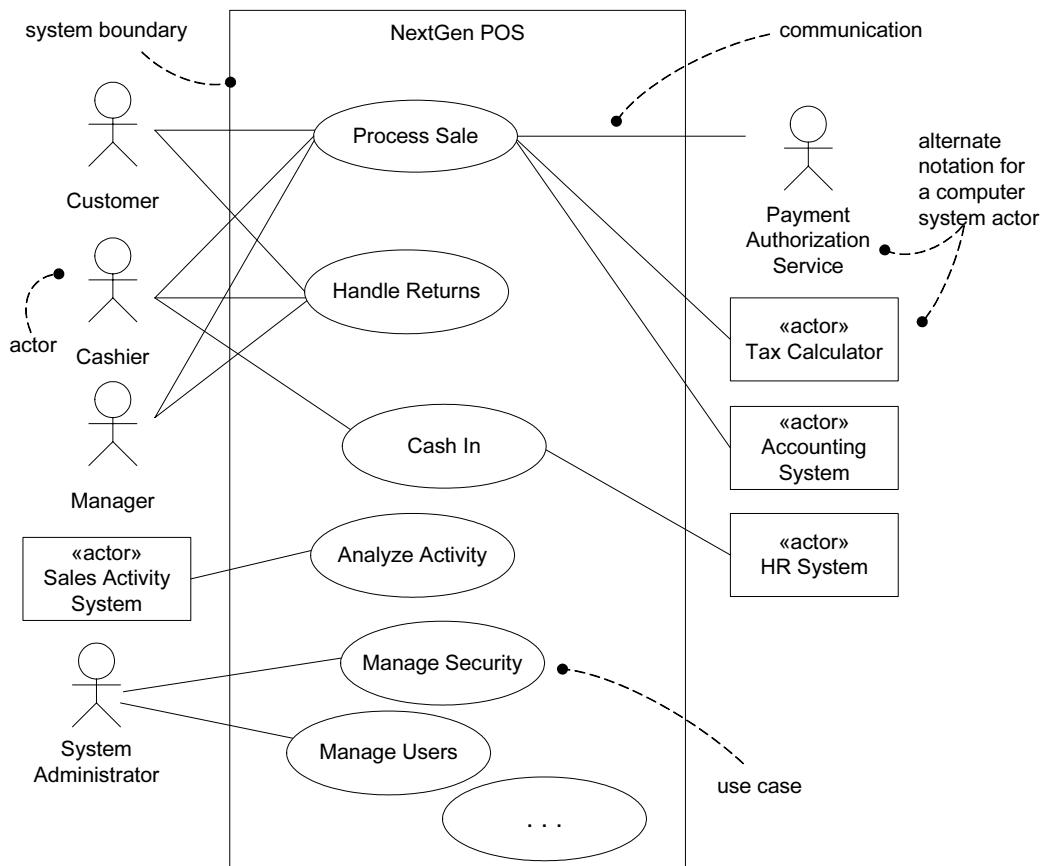


Figure 6.3 Partial use case context diagram.

Guideline

Draw a simple use case diagram in conjunction with an actor-goal list.

A use case diagram is an excellent picture of the system context; it makes a good **context diagram**, that is, showing the boundary of a system, what lies outside of it, and how it gets used. It serves as a communication tool that summarizes the behavior of a system and its actors. A sample *partial* use case context diagram for the NextGen system is shown in Figure 6.3.

APPLYING UML: USE CASE DIAGRAMS

Guideline: Diagramming

Figure 6.4 offers diagram advice. Notice the actor box with the symbol «actor». This style is used for UML **keywords** and **stereotypes**, and includes guillemet symbols—special *single*-character brackets («actor», not <>actor>>) most widely known by their use in French typography to indicate a quote.

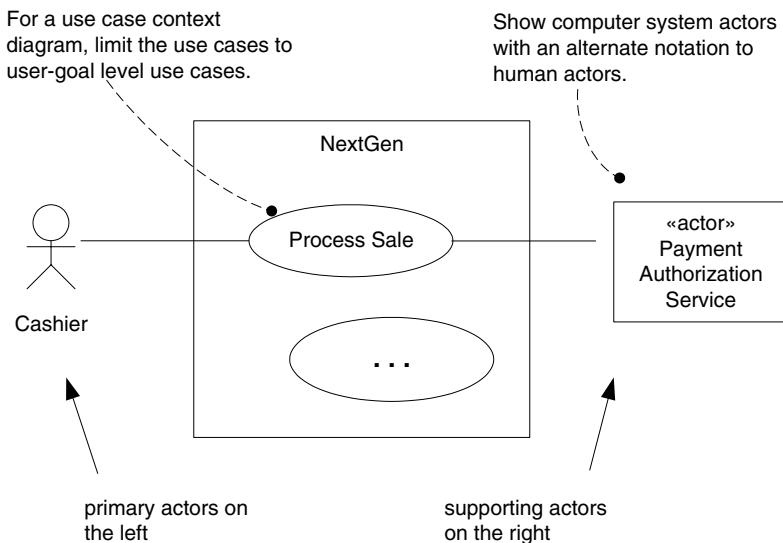


Figure 6.4 Notation suggestions.

To clarify, some prefer to highlight external computer system actors with an alternate notation, as illustrated in Figure 6.5.

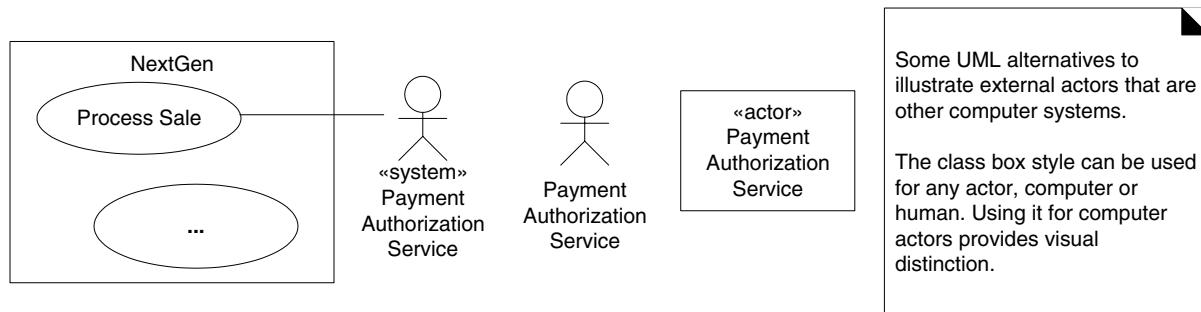


Figure 6.5 Alternate actor notation.

Guideline: Downplay Diagramming, Keep it Short and Simple

To reiterate, the important use case work is to write text, not diagram or focus on use case relationships. If an organization is spending many hours (or worse,

6 – USE CASES

days) working on a use case diagram and discussing use case relationships, rather than focusing on writing text, effort has been misplaced.

6.18 Applying UML: Activity Diagrams

UML activity diagrams p. 477

The UML includes a diagram useful to visualize workflows and business processes: activity diagrams. Because use cases involve process and workflow analysis, these can be a useful alternative or adjunct to writing the use case text, especially for *business* use cases that describe complex workflows involving many parties and concurrent actions.

6.19 Motivation: Other Benefits of Use Cases? Requirements in Context

motivation p. 64

A motivation for use cases is focusing on who the key actors are, their goals, and common tasks. Plus, in essence, use cases are a simple, widely-understood form (a story or scenario form).

Another motivation is to replace detailed, low-level function lists (which were common in 1970s traditional requirements methods) with use cases. These lists tended to look as follows:

ID	Feature
FEAT1.9	The system shall accept entry of item identifiers.
...	...
FEAT2.4	The system shall log credit payments to the accounts receivable system.

As implied by the title of the book *Uses Cases: Requirements in Context* [GK00], use cases organize a set of requirements *in the context* of the typical scenarios of using a system. That's a good thing—it improves cohesion and comprehension to consider and group requirements by the common thread of user-oriented scenarios (i.e., use cases). In a recent air traffic control system project: the requirements were originally written in the old-fashioned function list format, filling volumes of incomprehensible, unrelated specifications. A new leadership team analyzed and reorganized the massive requirements primarily by use cases. This provided a unifying and understandable way to pull the requirements together—into stories of requirements in context of use.

EXAMPLE: MONOPOLY GAME

Supplementary Specification p. 104 To reiterate, however, use cases are not the only necessary requirements artifact. Non-functional requirements, report layouts, domain rules, and other hard-to-place elements are better captured in the UP Supplementary Specification.

High-Level System Feature Lists Are Acceptable

Vision p. 109 Although detailed function lists are undesirable, a terse, high-level feature list, called *system features*, added to a Vision document can usefully summarize system functionality. In contrast to 50 pages of low-level features, a system features list includes only a few dozen items. It provides a succinct summary of functionality, independent of the use case view. For example:

Summary of System Features

- sales capture
- payment authorization (credit, debit, check)
- system administration for users, security, code and constants tables, and so on
- ...

When Are Detailed Feature Lists Appropriate Rather than Use Cases?

Sometimes use cases do not really fit; some applications cry out for a feature-driven viewpoint. For example, application servers, database products, and other middleware or back-end systems need to be primarily considered and evolved in terms of *features* (“We need Web Services support in the next release”). Use cases are not a natural fit for these applications or the way they need to evolve in terms of market forces.

6.20 Example: Monopoly Game

The only significant use case in the Monopoly software system is *Play Monopoly Game*—even if it doesn’t pass the Boss Test! Since the game is run as a computer simulation simply watched by one person, we might say that person is an observer, not a player.

This case study will show that use cases aren’t always best for behavioral requirements. Trying to capture all the game rules in the use case format is awkward and unnatural. Where do the game rules belong? First, more generally, they are **domain rules** (sometimes called business rules). In the UP, domain rules can be part of the Supplementary Specification (SS). In the SS “domain rules” section there would probably be a reference to either the official paper booklet of rules, or to a website describing them. In addition, there may be a pointer to these rules from the use case text, as shown below.

Supplementary Specification p. 104

6 – USE CASES

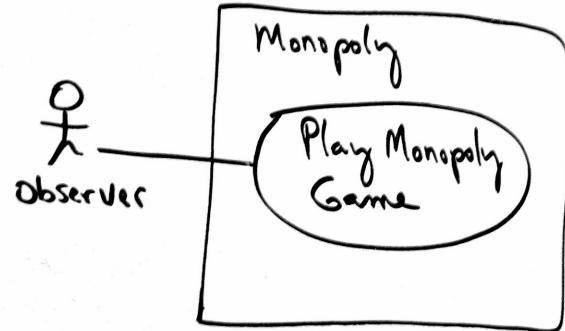


Figure 6.6 Use case diagram (“context diagram”) for Monopoly system.

The text for this use case is very different than the NextGen POS problem, as it is a simple simulation, and the many possible (simulated) player actions are captured in the domain rules, rather than the Extensions section.

Use Case UC1: Play Monopoly Game

Scope: Monopoly application

Level: user goal

Primary Actor: Observer

Stakeholders and Interests:

– Observer: Wants to easily observe the output of the game simulation.

Main Success Scenario:

1. Observer requests new game initialization, enters number of players.
2. Observer starts play.

3. System displays game trace for next player move (see domain rules, and “game trace” in glossary for trace details).

Repeat step 3 until a winner or Observer cancels.

Extensions:

- *a. At any time, System fails:

(To support recovery, System logs after each completed move)

1. Observer restarts System.
2. System detects prior failure, reconstructs state, and prompts to continue.
3. Observer chooses to continue (from last completed player turn).

Special Requirements:

- Provide both graphical and text trace modes.

PROCESS: HOW TO WORK WITH USE CASES IN ITERATIVE METHODS?

6.21 Process: How to Work With Use Cases in Iterative Methods?

Use cases are central to the UP and many other iterative methods. The UP encourages **use-case driven development**. This implies:

- Functional requirements are primarily recorded in use cases (the Use-Case Model); other requirements techniques (such as functions lists) are secondary, if used at all.
- Use cases are an important part of iterative planning. The work of an iteration is—in part—defined by choosing some use case scenarios, or entire use cases. And use cases are a key input to estimation.
- **Use-case realizations** drive the design. That is, the team designs collaborating objects and subsystems in order to perform or realize the use cases.
- Use cases often influence the organization of user manuals.
- Functional or system testing corresponds to the scenarios of use cases.
- UI “wizards” or shortcuts may be created for the most common scenarios of important use cases to ease common tasks.

How to Evolve Use Cases and Other Specifications Across the Iterations?

This section reiterates a key idea in evolutionary iterative development: The timing and level of effort of specifications across the iterations. Table 6.1 presents a sample (not a recipe) that communicates the UP strategy of how requirements are developed.

Note that a technical team starts building the production core of the system when only perhaps 10% of the requirements are detailed, and in fact, the team deliberately delays in continuing with deep requirements work until near the end of the first elaboration iteration.

This is a key difference between iterative development and a waterfall process: Production-quality development of the core of a system starts quickly, long before all the requirements are known.

Observe that near the end of the first iteration of elaboration, there is a second requirements workshop, during which perhaps 30% of the use cases are written in detail. This staggered requirements analysis benefits from the feedback of having built a little of the core software. The feedback includes user evaluation, testing, and improved “knowing what we don’t know.” The act of building software rapidly surfaces assumptions and questions that need clarification.

In the UP, use case writing is encouraged in a requirements workshop. Figure 6.7 offers suggestions on the time and space for doing this work.

6 – USE CASES

Discipline	Artifact	Comments and Level of Requirements Effort				
		Incep 1 week	Elab 1 4 weeks	Elab 2 4 weeks	Elab 3 3 weeks	Elab 4 3 weeks
Requirements	Use-Case Model	2-day requirements workshop. Most use cases identified by name, and summarized in a short paragraph. Pick 10% from the high-level list to analyze and write in detail. This 10% will be the most architecturally important, risky, and high-business value.	Near the end of this iteration, host a 2-day requirements workshop. Obtain insight and feedback from the implementation work, then complete 30% of the use cases in detail.	Near the end of this iteration, host a 2-day requirements workshop. Obtain insight and feedback from the implementation work, then complete 50% of the use cases in detail.	Repeat, complete 70% of all use cases in detail.	Repeat with the goal of 80–90% of the use cases clarified and written in detail. Only a small portion of these have been built in elaboration; the remainder are done in construction.
Design	Design Model	none	Design for a small set of high-risk architecturally significant requirements.	repeat	repeat	Repeat. The high risk and architecturally significant aspects should now be stabilized.
Implementation	Implementation Model (code, etc.)	none	Implement these.	Repeat. 5% of the final system is built.	Repeat. 10% of the final system is built.	Repeat. 15% of the final system is built.
Project Management	SW Development Plan	Very vague estimate of total effort.	Estimate starts to take shape.	a little better...	a little better...	Overall project duration, major milestones, effort, and cost estimates can now be rationally committed to.

Table 6.1 Sample requirements effort across the early iterations; this is not a recipe.

When Should Various UP Artifact (Including Use Cases) be Created?

Table 6.2 illustrates some UP artifacts, and an example of their start and refinement schedule. The Use-Case Model is started in inception, with perhaps only 10% of the architecturally significant use cases written in any detail. The majority are incrementally written over the iterations of the elaboration phase, so that by the end of elaboration, a large body of detailed use cases and other requirements (in the Supplementary Specification) are written, providing a realistic basis for estimation through to the end of the project.

PROCESS: HOW TO WORK WITH USE CASES IN ITERATIVE METHODS?

Discipline	Artifact Iteration →	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		

Table 6.2 Sample UP artifacts and timing. s - start; r - refine

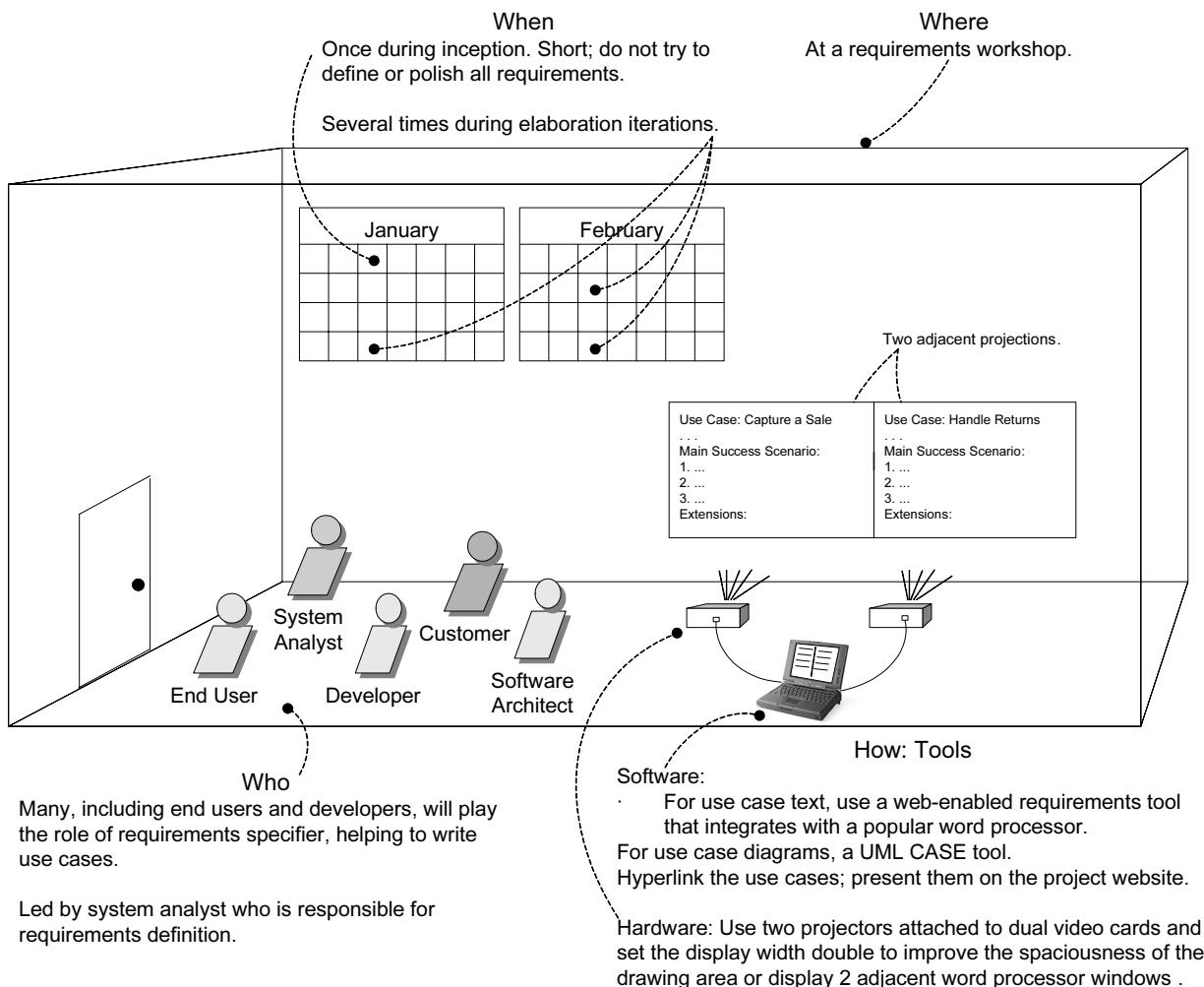


Figure 6.7 Process and setting context for writing use cases.

6 – USE CASES

How to Write Use Cases in Inception?

The following discussion expands on the information in Table 6.1.

Not all use cases are written in their fully dressed format during the inception phase. Rather, suppose there is a two-day requirements workshop during the early NextGen investigation. The earlier part of the day is spent identifying goals and stakeholders, and speculating what is in and out of scope of the project. An actor-goal-use case table is written and displayed with the computer projector. A use case context diagram is started. After a few hours, perhaps 20 use cases are identified by name, including *Process Sale*, *Handle Returns*, and so on. Most of the interesting, complex, or risky use cases are written in brief format, each averaging around two minutes to write. The team starts to form a high-level picture of the system's functionality.

After this, 10% to 20% of the use cases that represent core complex functions, require building the core architecture, or that are especially risky in some dimension are rewritten in a fully dressed format; the team investigates a little deeper to better comprehend the magnitude, complexities, and hidden demons of the project through deep investigation of a small sample of influential use cases. Perhaps this means two use cases: *Process Sale* and *Handle Returns*.

How to Write Use Cases in Elaboration?

The following discussion expands on the information in Table 6.1.

This is a phase of multiple timeboxed iterations (for example, four iterations) in which risky, high-value, or architecturally significant parts of the system are incrementally built, and the “majority” of requirements identified and clarified. The feedback from the concrete steps of programming influences and informs the team’s understanding of the requirements, which are iteratively and adaptively refined. Perhaps there is a two-day requirements workshop in each iteration—four workshops. However, not all use cases are investigated in each workshop. They are prioritized; early workshops focus on a subset of the most important use cases.

Each subsequent short workshop is a time to adapt and refine the vision of the core requirements, which will be unstable in early iterations, and stabilizing in later ones. Thus, there is an iterative interplay between requirements discovery, and building parts of the software.

During each requirements workshop, the user goals and use case list are refined. More of the use cases are written, and rewritten, in their fully dressed format. By the end of elaboration, “80–90%” of the use cases are written in detail. For the POS system with 20 user-goal level use cases, 15 or more of the most complex and risky should be investigated, written, and rewritten in a fully dressed format.

Note that elaboration involves programming parts of the system. At the end of this step, the NextGen team should not only have a better definition of the use cases, but some quality executable software.

HISTORY

How to Write Use Cases in Construction?

The construction phase is composed of timeboxed iterations (for example, 20 iterations of two weeks each) that focus on completing the system, once the risky and core unstable issues have settled down in elaboration. There may still be some minor use case writing and perhaps requirements workshops, but much less so than in elaboration.

Case Study: Use Cases in the NextGen Inception Phase

As described in the previous sections, not all use cases are written in their fully dressed form during inception. The Use-Case Model at this phase of the case study could be detailed as follows:

Fully Dressed	Casual	Brief
Process Sale Handle Returns	Process Rental Analyze Sales Activity Manage Security ...	Cash In Cash Out Manage Users Start Up Shut Down Manage System Tables ...

6.22 History

The idea of use cases to describe functional requirements was introduced in 1986 by Ivar Jacobson [Jacobson92], a main contributor to the UML and UP. Jacobson's use case idea was seminal and widely appreciated. Although many have made contributions to the subject, arguably the most influential and coherent next step in defining what use cases are and how to write them came from Alistair Cockburn (who was trained by Jacobson), based on his earlier work and writings stemming from 1992 onwards [e.g., Cockburn01].

6.23 Recommended Resources

The most popular use-case guide, translated into several languages, is *Writing Effective Use Cases* [Cockburn01].⁷ This has emerged with good reason as the most widely read and followed use-case book and is therefore recommended as a primary reference. This introductory chapter is consequently based on and consistent with its content.

7. Note that Cockburn rhymes with *slow burn*.

6 – USE CASES

Patterns for Effective Use Cases by Adolph and Bramble in some ways picks up where *Writing* leaves off, covering many useful tips—in pattern format—related to the process of creating excellent use cases (team organization, methodology, editing), and how to better structure and write them (patterns for judging and improving their content and organization).

Use cases are usually best written with a partner during a requirements workshop. An excellent guide to the art of running a workshop is *Requirements by Collaboration: Workshops for Defining Needs* by Ellen Gottesdiener.

Use Case Modeling by Bittner and Spence is another quality resource by two experienced modelers who also understand iterative and evolutionary development and the RUP, and present use case analysis in that context.

“Structuring Use Cases with Goals” [Cockburn97] is the most widely cited paper on use cases, available online at alistair.cockburn.us.

Use Cases: Requirements in Context by Kulak and Guiney is also worthwhile. It emphasizes the important viewpoint—as the title states—that use cases are not just another requirements artifact, but are the central vehicle that drives requirements work.

James K. Peckol

Embedded Systems Design, A Contemporary Design Tool

Wiley, ISBN 978-0-471-72180-2.

Chapter 10, Hardware Test and Debug, pp 401 - 407

The *Data Assembler* must operate according to the following algorithm:

The first 4 bits appear on the bus accompanied by a *DV—Data Valid* signal.

The *DV* transitions from a logical 0 to a logical 1 to signify that the data on the bus is valid.

The 4 bits of data are loaded, in parallel, into the *most significant* 4 bits of an 8-bit shift register.

The *DV* transitions from a logical 1 to a logical 0.

An output signal, *Shift Left*, is generated to enable the shift register to be clocked four times.

The second 4 bits appear on the bus accompanied by a *DV—Data Valid* signal.

The *DV* transitions from a logical 0 to a logical 1 to signify that the data on the bus is valid.

The 4 bits of data are stored into the *most significant* 4 bits of the 8-bit shift register, thereby building up the complete 8-bit word.

The *DV* transitions from a logical 1 to a logical 0.

A *Load Complete* output signal is generated to signify that the data has been captured.

The system needs a clock source that has a 0.5- μ s period as part of the system time base. An 18-MHz clock source is the only one available in the system.

9.11 Choose a system of your own.

Chapter 10

Hardware Test and Debug

THINGS TO LOOK FOR ...

- The vocabulary of testing.
- The reasons for debugging, troubleshooting, and testing.
- The need for planning, specifications, test procedures, and test cases.
- Steps and heuristics for the debugging process.
- Identification and isolation of common faults in combinational and sequential circuitry.
- Tests for the designer and for the customer.

10.0 INTRODUCTION

Formulating a testing, debugging, or troubleshooting strategy should occur early in the design and development process. Ideally, it should be concurrent with the hardware and software development.

Most often, when we come up with an idea for any new design, we begin with a plan. We think about what the system will do; we think about its features, its capabilities, and its functionality. Debugging, troubleshooting, and testing are no different. We must have a plan. Without a plan, we can't be sure what we are looking for, or what it might look like if we find it, or when we are finished.

We will begin the study of test by introducing some of the relevant vocabulary. Understanding the terminology facilitates understanding the requirements of a test or test strategy, as well as the capabilities and limitations of the equipment utilized to execute the strategy. Earlier discussions of the effects of microprocessor word size on real numbers and subsequent computations now provide a basis for understanding, formulating, and interpreting measurements during the test process. We will then present a high-level model for a testing strategy; examine and motivate the need for planning, specifications, test procedures, and test cases; and examine several views, ranging from black to white box models, for approaching test. Finally, we will move into testing during the different stages of the product life cycle.

10.1 SOME VOCABULARY

As a prelude to the study of debugging and testing, it is important to understand some of the vocabulary of the area. Making measurements or generating signals is rather straightforward. Doing so properly is more of a challenge. In part, the vocabulary will illustrate where the challenges lie. As we do so (philosophy aside), it is important to recognize that physical entities exist, they have attributes, and, based on fundamental physics, those attributes have

values independent of any ability to measure them or to replicate them. Herein lie most of the challenges.

<i>True Value</i>	The actual or inherent value of a physical quantity.
<i>UUT / DUT</i>	Unit or Device Under Test.
<i>Accuracy</i>	The measure of an instrument's capability to approach a true or absolute value.
<i>Resolution</i>	Measure of ability to discern the value of a measurement. Expression of the value measurement to 1, 2, or 3 decimal places provides three levels of resolution of the true value of the measured value.
<i>Variance</i>	Has no unit of measure. Variance provides an indication of the relative degree of repeatability of a set of measurements; that is, how closely the values of series of repeated measurements agree with each other.
<i>Mean</i>	Measure of the central value of a set of measurements and is given by the following equation. m_i is the value of an individual measurement.
	$\text{mean} = \frac{1}{N} \sum_{i=0}^{N-1} m_i$
<i>Root Mean Square</i>	The square root of the average of the squares of a set of values and given by the following equation. m_i is the value of an individual measurement.
	$\text{rms} = \sqrt{\frac{\sum_{i=0}^{N-1} (y_i)^2}{N}}$
<i>Bias</i>	Measure of how closely the mean value in a series of repeated measurements approaches the true value.
<i>Residual</i>	Measured value minus the mean.
<i>Golden Unit</i>	Unit whose behavior is completely known used as a standard.
<i>Statistical Tolerance Interval</i>	Estimate of the amount of measurement variability due to the test system, excluding the UUT variability. Test limits must be outside the STI limits.
<i>Test Limits</i>	Upper and Lower physical limits of the measurement.

With some of the basic vocabulary in hand, we begin by formulating a high-level strategy.

10.2 PUTTING TOGETHER A STRATEGY

debugging, testing, troubleshooting

Although the words *debugging*, *testing*, and *troubleshooting* all have the same general objectives, they represent three different tasks; they are undertaken at different times during the product's lifetime and with different underlying assumptions. Debugging is done during the early phases; testing is done before delivery; and troubleshooting afterwards. Debugging does not assume that the design has ever worked. *Debugging* is a process utilized to identify the cause of problems that occur in the design and implementation of a system as it is incrementally made to work. *Testing* begins with the assumption that the design is correct. It is a process charged with identifying any faults that may have been introduced during the manufacture of the system and ensuring that a properly working product is delivered to the cus-

troubleshooting

customer. *Troubleshooting* begins with the premise that the design is correct and that the product worked at one time. The troubleshooting process seeks to identify which hardware component(s) have failed. The focus is on the hardware because, as was stated earlier in the discussion of safety and reliability, software components do not wear out and fail during use.

During the early stages of product development, any debugging plan is likely to be less formal. Most of it will probably be in the designer's head. Normally, the person who designed the circuit or the software *should* have a good idea of what to look for in terms of both Unit Under Test (UUT) functionality and the potential cause(s) of any anomalous or unintended behavior. Such is not always the case, however, so outlining a well-reasoned and orderly strategy is an excellent first step to help to focus the process.

As the design evolves and progresses through the development cycle, the need for a more formal approach becomes essential. In production, formal test procedures, based on a formal test plan, are required. In the field, troubleshooting procedures for the customer or the field service personnel are an essential part of a product's deliverables.

**test plan, test specification,
test procedure, test cases**

In the ensuing discussion, the terms *test plan*, *test specification*, *test procedure*, and *test cases* are used in a generic sense. They apply equally to debug, test, and troubleshooting; only the initial presumptions and focus will differ for each. Although the main focus of the chapter is on the hardware side, the vocabulary, strategy, and philosophy apply equally well to the software side.

10.3 FORMULATING A PLAN

stress testing

Testing in industry is not taken lightly. In companies that know what they are doing, it is a very serious task. For example, let's consider the approach of a very large manufacturer of electronic test and measurement equipment. For many large companies, there is a full, highly qualified test group. For smaller companies, the approach may be to take several senior engineers off design projects and have them thoroughly test a (new) product with the intention of finding problems before the design is approved for release to production.

In some cases, the engineers have no knowledge of the specifics of the software or hardware inside of the box; they simply try to break it. Such testing is also called *stress testing* because the idea is to try to stress the software or hardware to break it or to find potential problems before they surface in a delivered product.

With a high-level overview, it is possible to begin to see how to put the high-level concepts to use. It is important to remember that testing is an integral part of all phases of product development, including design. Testing is done for four main reasons, as shown in Figure 10.0.

- To verify that any of the following performs as intended:
 - Code module or hardware prototypes
 - Subsystem or collection of subsystems
 - System interfaces
- To ensure that the system meets specification
- To ensure that any changes to the system work
 - Do not alter other intended functionality.
- To ensure that the system functions properly after being built

Figure 10.0 Principal Reasons to Test

*System Test Plan
Test Specification*

Each of these reasons has a different objective and scope and tests different aspects of the design. All such requirements should be found in the *System Test Plan* and subsequent *Test Specification*. Remember: like the System Requirements Document that was discussed earlier in the context of original design, the Test Plan identifies what tests need to be carried out. It describes in general terms the following information:

- What is to be tested?
- The testing order within each type of test
- Assumptions made
- Algorithms that may be used

*Test Plan, what
Requirements Specification
how*

Testing formality increases as the system moves toward the latter stages of development. Testing to ensure functionality can be reasonably informal; we still should have some form of plan. As the system begins to come together, formality must increase. Today's systems are becoming too complex. It is becoming too easy to miss critical, yet subtle, points.

A *Test Plan* begins the process. It describes in general terms *what* must be tested. Such a plan is based on the initial requirements captured in the *Requirements Specification*. It may specify *how* the testing will be carried out, the testing order within each general category of test (input, output, processing), as well as any assumptions that are made.

EXAMPLE 10.0

Consider a simple AND gate as a Unit or (device) under test UUT/DUT (see Figure 10.1).

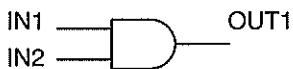


Figure 10.1 Unit Under Test

A test plan might be similar to the following,

First verify the following static behavior of the device

- a. Ensure that the circuit functions as a logical AND gate according to its specified truth table.
- b. Verify the following signal values: $V_{OH\min}$, $V_{OL\max}$, $V_{IH\min}$, $V_{IL\max}$.

Then verify its dynamic behavior.

- a. Confirm the following parameters: τ_{PDHL} , τ_{PDLH} , τ_{rise} , τ_{fall} ,

When putting an informal or formal test plan together, remember that the goal is to capture the essence of what must be tested. Keep things simple, precise, and to the point. The plan is not intended to be the next great novel.

The test plan establishes a strategy. During debugging, an informal test plan guides the process. During this phase, the behavior of the design is examined from a high level. Does a 5-volt signal appear on this pin? Does the counter work? Is the proper sequence of control signals being generated? Once again, informality, yet precision, is appropriate. As the design matures, testing must be based on concrete values and tolerances.

10.4 FORMALIZING THE PLAN—WRITING A SPECIFICATION

Test Specification Requirements Specification, Design Specification what how

The *Test Specification* evolves from and formalizes the test plan in a manner analogous to the relationship between the *Requirements Specification* and *Design Specification* studied earlier. The test specification includes a description of and specification for each test. As with the test plan, its focus remains on *what* is being tested. Analogous to the design specification studied earlier, the test specification also begins to establish *how* the tests are to be carried out and what the appropriate test stimuli and test limits should be. The test specification assigns specific values, limits, and tolerances to all of the parameters to be tested based on what was stated in the design specification. These values ultimately lead to constraints, requirements, and specifications for the test equipment to be utilized in creating and conducting the tests.

The test specification that is used during the design phase will often serve as a base on which to build the production test strategy. The full complement of tests utilized during the design phase to ensure compliance with the system specifications is generally not needed once the product is in production. Upon release to production, the design is assumed to be correct. A specific subset to confirm continued compliance is definitely appropriate.

The test specification for the UUT will now take on a more formal appearance as we see in the next example.

EXAMPLE 10.1

Verify the following static behavior of the device.

- Ensure that the circuit functions as a logical AND gate according to its specified truth table.

IN1	IN2	OUT1
0	0	0
0	1	0
1	0	0
1	1	1

- Verify the following signal values and limits:

$$\begin{aligned}
 V_{OH\min} &= 2.4 \pm 0.003 \text{ V}_DC \\
 V_{OL\max} &= 0.4 \pm 0.001 \text{ V}_DC \\
 V_{IH\min} &= 2.0 \pm 0.003 \text{ V}_DC \\
 V_{IL\max} &= 0.8 \pm 0.001 \text{ V}_DC
 \end{aligned}$$

Next, we must verify its dynamic behavior.

Confirm the following parameters:

$$\begin{aligned}
 \tau_{PDHL} &= 15.0 \pm 0.05 \text{ ns} \\
 \tau_{PDLH} &= 15.0 \pm 0.05 \text{ ns} \\
 \tau_{rise} &= 5.0 \pm 0.001 \text{ ns} \\
 \tau_{fall} &= 5.0 \pm 0.001 \text{ ns}
 \end{aligned}$$

The test specification provided a quantified description of what must be tested. Now, those tests must be carried out. During the debugging phase, the procedures and approaches may be based primarily on heuristic experience. In production, such an approach gives way to more formal methods.

10.5 EXECUTING THE PLAN—THE TEST PROCEDURE AND TEST CASES

Test Procedure, Test Cases,

how

test suite

The *Test Procedure* and *Test Cases* specify *how* the test plan and specification are to be implemented and must provide the detailed lists of the necessary equipment and the steps for each test. These documents first decompose the plan into a series of blocks in much the same way we first decomposed the overall design into functional modules. Each block has a specific behavior, parameter, or set of related parameters in the system that it is testing. It gives the order of the test steps, values, and ranges of stimuli to be applied to the UUT during each test step. It specifies the values and ranges of the resulting measurements for each step. A series of (related) test cases is called a *test suite*.

Test case design is essential for testing at any level. The content of test cases will of course vary with the specific nature and intent of each individual test. During the early stages of test, one must test the design for behavior with following three kinds of values:

- Expected values
- Unexpected values
- The boundaries of expected values, inside, outside, and at the boundary

Recall the earlier comments in the chapter on safety and reliability.

The test values may be randomly generated test vectors or statistically based patterns. Such an approach is reasonable for combinational logic; however, it falls down on sequential types of relationships. See the earlier discussions of testing sequential circuits.

test coverage

To describe the efficacy of a test, the phrase *test coverage* is used. Test coverage provides the percentage of hardware, software, or system tested in a specific test or series of tests. When putting the test cases together, one must ensure that every path through the system is traversed at least once with signals of both polarities for the hardware cases and with variables of nominal and extreme values for the software cases.

During test, the emphasis is primarily on the system's behavior as manifest through various hardware signals. The purpose of the underlying firmware is to produce the intended or specified hardware behavior. Access to such hardware signals is gained through test points and/or test connectors. Access to software-driven results comes indirectly through those same signals. These are a signal or sets of signals, internal to the UUT, that can be observed directly via a probe point or connector that is incorporated into the circuit or system during design. When direct access to signals (inside of a complex component) is not available, boundary scan or similar techniques are used.

The test suite must evolve with testing. As faults are found and fixed, further tests are often suggested that may find similar faults that were not in the original test design.

The details of the test procedures strongly depend on the test system being used. Often such systems are a combination of commercial instruments such as power supplies, function generators, or digital word generators and proprietary circuits designed specifically for the test system. During debugging, one may use more sophisticated analysis equipment such as data generators and logic analyzers or oscilloscopes to help to verify the design. In production, it is assumed that the design is good; testing serves to identify defects introduced during manufacturing.

10.6 APPLYING THE STRATEGY—EGOLESS DESIGN

Let's now follow the test process from the initial stages of testing prior to release to manufacture. The process commences with testing for ourselves and then moves to testing for our customer.

egoless design

A key element of debug and testing during the early phases of the development life cycle is *egoless design*. What does design have to do with test? At this stage, testing begins with the initial specification as the basis for evaluating the preliminary designs. As we discussed in Chapter 8, such evaluation occurs through design reviews, code walkthroughs, and code inspections. One cannot let the belief that the best widget ever witnessed by humankind has just been designed to get in the way of an unbiased assessment of that design to determine if it is really the case. It is essential that ego not be part of the review process.

Design reviews, code walkthroughs, and code inspections must be done by someone else. As is often the case when proofreading one's own writing, schematics, or code, our brain ensures that they appear exactly the way we want them to rather than as it is actually written or designed.

10.7 APPLYING THE STRATEGY—DESIGN REVIEWS

During the early stages of product design, we are really testing for ourselves. That testing does not start when the first few pieces are on a lab bench or a couple of high-level algorithms have been coded up. In reality, it must begin much earlier during the design process.

A good first step in this direction is to hold design reviews as the design of the system progresses. Initially, such a review can ensure that everyone understands the high-level specifications and functionality of the system. Thereafter, a design review preceding the architectural phase can confirm detailed functionality. As the design progresses, at least one review prior to moving to prototype can ensure that the mapping from function to processors, FPGAs, or ASICs is sound. The formality of such reviews can vary with need. They can range from a simple exchange of drawings and code among the team members to detailed reviews with reviewers who are not directly involved with the project.

No matter how one chooses to proceed, it is important that the review be conducted in a constructive manner. All participants should recognize that a good review helps to ensure a more robust product at the end of the day.

10.8 APPLYING THE STRATEGY—MODULE DEBUG AND TEST

smoke test debugging

As the design moves along the development cycle, the first prototypes of the individual modules are built and ready for a *smoke test* as the jargon goes. The phase now entered is called *debugging*. During this phase, the goal is to identify all of the different kinds of errors and faults that might have occurred in the development of a new or modified circuit, software module, or system.

Caution: When running a smoke test. There is strong scientific evidence that most electronic circuits have an embedded smoke demon whose presence is essential to keeping the circuit working. Empirical evidence seems to support the theory since we can show that once the smoke demon is released from the circuit, it no longer works.

On both sides, we have design errors and oversights that have escaped earlier analysis, modeling, and reviews. On the hardware side we have implementation issues such as wiring errors (which often lead to stuck-at types of faults) and incorrect or incorrectly installed parts that have occurred during the prototype build.

To effectively debug the design, it is essential to know what behavior is being tested, how the necessary and appropriate stimuli are going to be produced, and how the results are going to be analyzed.

Stephen Biering-Sørensen, Finn Overgaard Hansen, Susanne Klim, Preben Thalund Madsen

Struktureret Program-Udvikling (SPU)

Teknisk Forlag, ISBN 87-571-1046-8.

Vejledning i softwaretest, pp 171 - 207

Vejledning i softwaretest

Indhold

- 1. Indledning** 172
 - 1.1 Hvorfor teste? 172
 - 1.2 Hvad er systematisk test? 173
- 2. Generelt om test** 174
 - 2.1 Testbehov 174
 - 2.2 Testplanlægning 175
 - 2.3 Testteknikker 177
 - 2.4 Testbemanding 180
- 3. SPU-tests** 181
 - 3.1 Modultests 181
 - 3.2 Integrationstests 182
 - 3.3 Accepttest 187
- 4. Testforberedelse og -udførelse** 190
 - 4.1 Specifier testemner 190
 - 4.2 Design testen 192
 - 4.3 Implementer testcases og testomgivelser 193
 - 4.4 Kør testen 193
 - 4.5 Evaluér testforløbet 194
- 5. Udvælgelse af effektive testcases** 195
 - 5.1 Design af testcases til blackboxtests 195
 - 5.2 Design af testcases til whiteboxtests 198
 - 5.3 Strategi for udvælgelse af testdata 199
- 6. Testdokumentation** 200
 - 6.1 Referencedokumenter 201
 - 6.2 Indholdsfortegnelse for en testspecifikation 201
 - 6.3 Indholdsfortegnelse for en testrapport 204
 - 6.4 Testdokumentationens udvikling gennem faserne 206
- 7. Debugging** 208
- 8. Testværktøjer** 209
- 9. Vejledningens hovedpunkter** 210
- Litteraturliste** 211

1. Indledning

Formålet med denne vejledning er at sætte softwareudviklere i stand til at planlægge og udføre systematisk softwaretest. Ifølge denne vejledning ligger den første testplanlægning ved projektets begyndelse, således at ideerne fra vejledningen bør tages op allerede ved projektetableringen. Vejledningen lægger vægt på at testarbejdet opdeles. Svarende til SPU-modellen er testarbejdet overordnet opdelt i modultest, integrationstest og accepttest. Hver af disse tests er igen detaljeret opdelt i testaktiviteterne forberedelse, kørsel og evaluering. Testaktiviteter og dokumentation er beskrevet generelt, dvs. at uanset hvilken type test, man skal udvikle og udføre, er arbejdsgangen og dokumentationsstrukturen den samme.

Vejledningen gennemgår i kapitel 2 nogle generelle aspekter omkring test, nemlig testbehov, testbemanding, testplanlægning og forskellige testteknikker. I kapitel 3 beskrives de tests, som SPU-modellen foreskriver, nemlig modultest, modulintegration, procesintegration og accepttest i større detaljer, idet de forskellige tests har forskellige formål.

I kapitel 4 gennemgås i tidsmæssig rækkefølge de enkelte testaktiviteter. Som en meget vigtig testaktivitet gennemgås i kapitel 5 separat principperne for at udvælge testcases.

I kapitel 6 gennemgås herefter den testdokumentation, som vejledningen anbefaler. Et testdokument kan ikke skrives på én gang, men bygges op af forskellige afsnit, som skrives efterhånden svarende til testaktiviteterne gennemgået i kapitel 4.

Kapitel 7 summerer ganske kort nogle enkle principper for fejlfinding og fejlretning, og kapitel 8 giver eksempler på testværktøjer.

1.1 Hvorfor teste?

Følgende facts er vigtige for at forstå softwaretest:

- Mennesker laver fejl. Derfor skal tests udvikles med det formål at jagte fejl. Jo flere fejl man finder jo bedre.
- Det tager tid at finde fejl, og test skal inkluderes i projektplanlægningen med realistiske tider. På større softwareprojekter har ca. 50% af udviklingstiden været brugt til test. Systematisk testplanlægning er med til at nedsætte dette tal.
- Fejl skal findes så tidligt som muligt. Det er dårlig økonomi at udskyde alle testaktiviteter til de sidste faser.
- Udviklere er blinde over for egne fejl. Det er derfor hensigtsmæssigt at bemande et projekt, således at ingen skal teste sit eget program alene.

- Selv gode tests kan ikke redde et dårligt program. Programkvaliteter skal indbygges løbende.

1.2 Hvad er systematisk test?

Systematisk test kan beskrives ved hjælp af følgende punkter:

- Gør dig klart, hvad du skal teste.
- Gør dig klart, hvorfor du tester. Hvor vigtigt er det, at det færdige produkt ikke fejler?
- Opdel testarbejdet i mindre, overskuelige aktiviteter.
- Udvælg omhyggeligt testdata til hver enkelt test, og forudsig resultaterne.
- Vær ikke alene om både at planlægge og køre testen.
- Læg vægt på, at tests skal kunne gentages.
- Undgå at sidde foran skærmen for "...lige at prøve noget!" Vær opmærksom på forskellen mellem debugging og test. Test er nøje planlagte programkørsler, som skal finde fejl. Når der er fundet en fejl, anvendes debugging til at finde ud af, hvor fejlen er.
- Kør en passende mængde af tests. Tag resultaterne med ind til skrivebordet. Kontroller dem ved at sammenligne med det forudsigte resultat. Planlæg næste trin.
- Sæt arbejdet i system ved at anvende fortrykte formularer, ringbind med fanebladsinddeling, og eventuelt en database til at gemme testdata, resultater, o.lign.
- Lad testresultaterne have en form, som er direkte brugbar i en testrapport.

2. Generelt om test

I dette kapitel beskrives nogle grundlæggende ting om softwaretest. Det er:

- Testbehov. Hvilke grunde kan projektgruppen have til at ville investere resurser i testarbejde?
- Testplanlægning. Hvordan er opdelingen af testarbejdet i SPU-sammenhæng?
- Testteknikker. Hvordan tester man i praksis?
- Testbemanding. Hvem skal teste?

2.1 Testbehov

Det tager tid og energi at foretage en systematisk og grundig test. Derfor er det vigtigt at gøre sig klart hvilke behov, der er for test. Måske er det tilstrækkeligt at debugge sig frem til et program, der ser ud til at virke. Måske er det ikke nødvendigt at lave separate modultests på alle moduler eller at lave formelle, nedskrevne integrationstests.

Hvor godt et program skal testes, afhænger af mange faktorer. De følgende eksempler kan give et fingerpeg om, hvad der skal overvejes.

Hvordan gik det ved afleveringen af det sidste system?

'Vi havde lovet at aflevere den 15. september. Det gjorde vi, så vi måtte rejse over og rette alle de fejl, der blev fundet bagefter.'

'Det betød ikke noget, om det varede en måned mere eller mindre.'

'Kunden ville ikke betale, før systemet virkede, som han mente, der var aftalt.'

Er en fejl kritisk i det endelige system?

'Ja, du godeste, der er jo menneskeliv på spil!'

'Ja, produktionsstop koster 50.000 kr. pr. time.'

'Ja, tænk på firmaets gode rygte og markedsandel.'

'Nej, den retter vi bare.'

Vil en eventuel fejl være svær (dyr!!) at rette?

'Ja, vores system er installeret på Nordpolen.'

'Ja, vi sælger 50.000 apparater om året, og vi kan ikke rette, når først apparatet er på markedet.'

'Nej, udstyret står jo i vores laboratorium.'

Skal der bygges/testes videre på programmet?

'Ja, vi forventer at sælge et stort antal af dette program i mange variationer.'

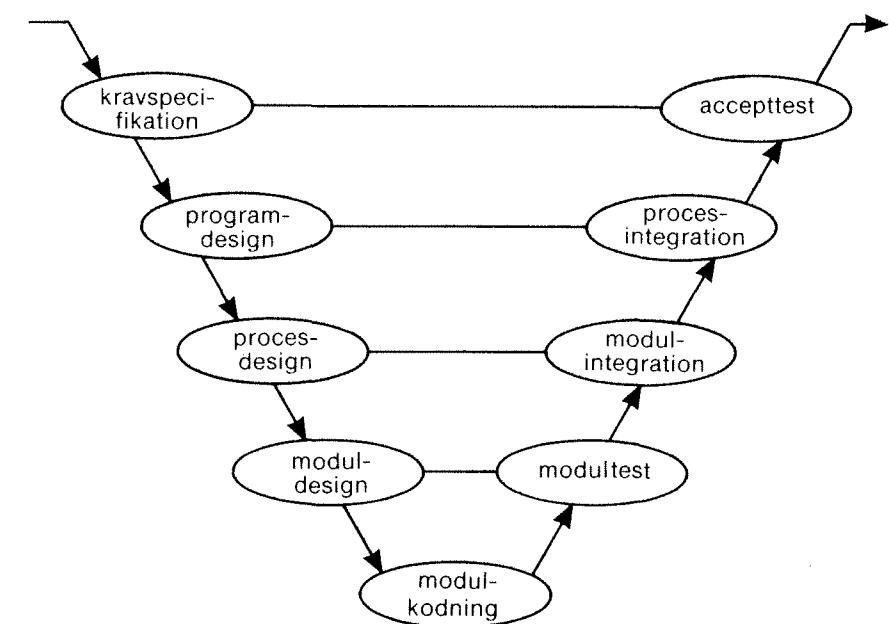
'Ja, og vi ønsker ikke, at Søren skal hænge på vedligeholdelse i al fremtid.'

'Nej, det er kun et demo-program.'

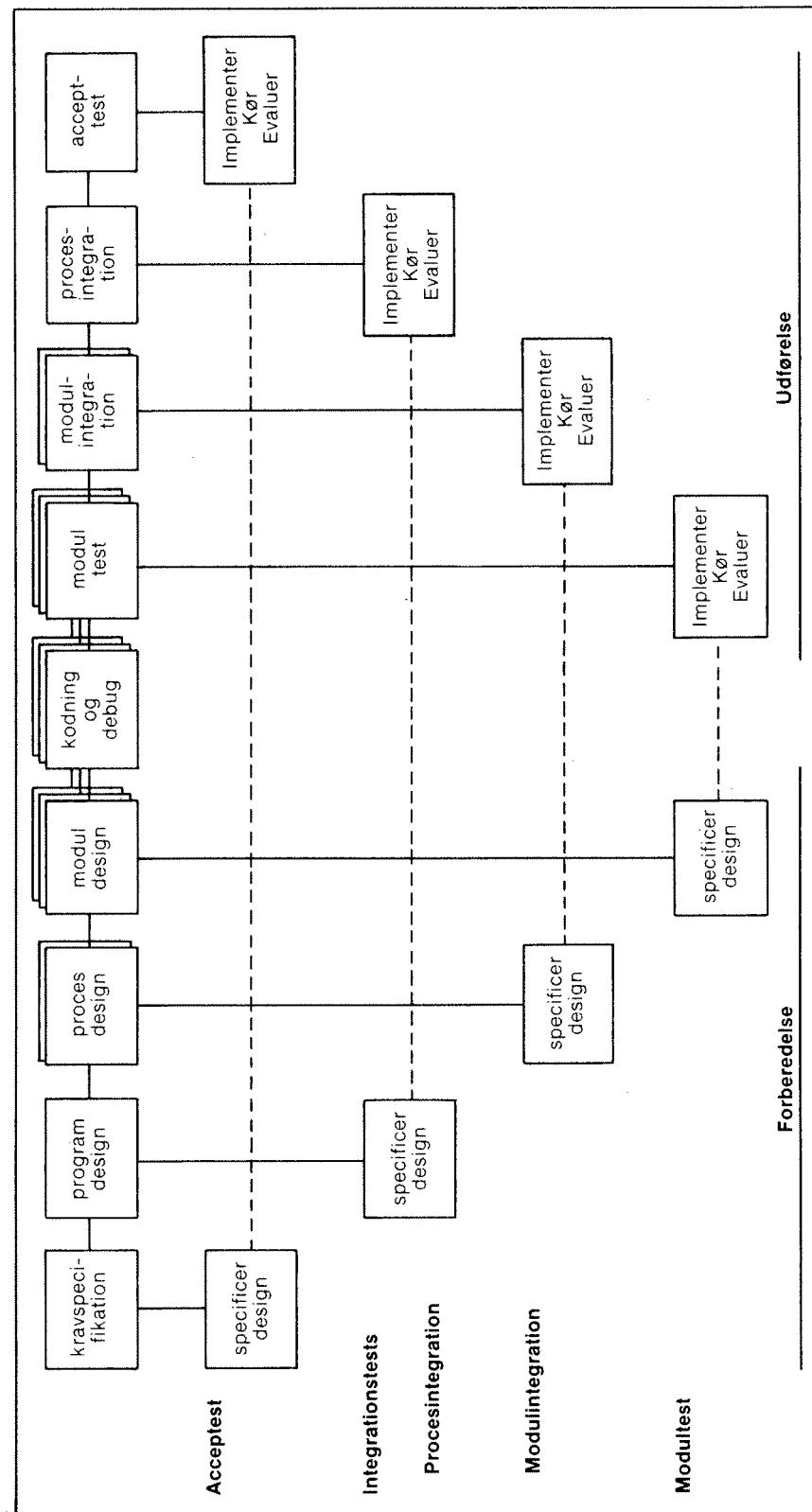
2.2 Testplanlægning

I et projektforløb kan testarbejdet opdeles i modultest, modulintegration, procesintegration og accepttest.

Man skal ikke vente til testfaserne med at tænke på test. Forberedelsen bør i vid udstrækning finde sted tidligt i projektet. Accepttesten kan forberedes, så snart den færdige kravspecifikation foreligger, integrationstesten så snart program- og procesdesignet er færdigt, og modultesten så snart specifikationen for det enkelte modul er færdigt.



Figur 1. SPU's 'V'-model for testaktiviteter.



Figur 2. Testaktiviteter i projekts faser.

'V'-modellen på figur 1 viser, hvorledes testfaserne er knyttet sammen med de egentlige udviklingsfaser:

- accepttesten er knyttet sammen med kravspecifikationen og tester således det færdige produkts funktioner og egenskaber,
- procesintegrationen er knyttet sammen med programdesign og tester således samspillet mellem processerne efterhånden som de integreres,
- modulintegrationen er knyttet sammen med designet af den enkelte proces og tester således samspillet mellem modulerne efterhånden som de integreres,
- modultesten er knyttet sammen med beskrivelsen af det enkelte moduls moduldesign og tester således funktioner og egenskaber ved det enkelte modul.

Figur 2 viser testaktiviteterne for hver test opdelt i forberedelse (SPECIFICER testemner og DESIGN testen) og udførelse (IMPLEMENTER testen, KØR testen og EVALUER testen), således som de passer ind i projektets faser. Testaktiviteterne er beskrevet i kapitel 4.

Der er mange fordele ved at planlægge testaktiviteterne på den måde. Man opnår, at planlægningen udføres før den ofte hektiske testfase, hvor man måske uden planlægning ville fristes til at skyde genveje. Dernæst findes der gennem testplanlægning en masse mangler, fejl og uklarheder i den grundlæggende dokumentation, og man får mulighed for at påvirke udviklingen af programmet, så det bliver lettere at teste. Til projektstyringen opnår man en detaljeret oversigt over kommende arbejdsopgaver, hvilket muliggør en bedre planlægning af det resterende projekt.

2.3 Testteknikker

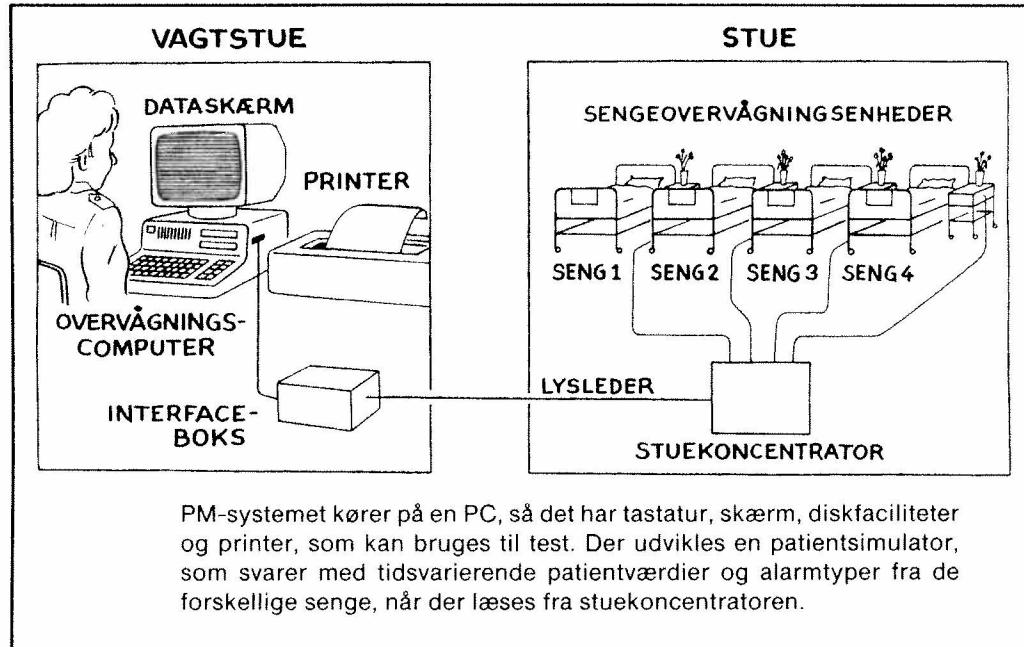
En test foregår ved, at man påvirker den software, der skal testes, med vel-defineret input og observerer om output er i overensstemmelse med det forventede. Det er testomgivelserne, der bestemmer, hvordan testinput påtrykkes, og hvordan testresultaterne præsenteres.

Ved planlægning af en test skal man tænke på, at den skal kunne gentages også af andre end den, der oprindeligt kørte den. Det betyder, at såvel de anvendte testcases, dvs. input plus forventet output, som testomgivelser skal være vel-dokumenterede, og at alle testresultater skal logges.

Testomgivelser

Til testbrug er det hensigtsmæssigt at have tastatur, skærm, prin-

ter og disk-faciliteter. Det er imidlertid vidt forskelligt, hvilke testfaciliteter et produkt har til rådighed. Et PC-produkt har mange testfaciliteter, som gør systematisk test enkel, mens en mikroprocessorbaseret styring kun har få naturlige testfaciliteter, således at man nødvendigvis må udvikle specielle testomgivelser for at kunne se, hvorledes produktet opfører sig.



Eksisterende testfaciliteter i PM.

Til forskellige testtyper har man forskellige ønsker til sine testomgivelser. I de tidlige testfaser er det vigtigt at teste i isolerede, kontrollerede omgivelser. Ved accepttesten er det ønskeligt, at testomgivelserne ligner den virkelige verden så meget som muligt.

Der er i høj grad tale om at afbalancere omkostningerne ved at udvikle testomgivelser og den ensartethed, effektivitet og sikkerhed man får til gengæld. Ved udvikling af f.eks. satellitstyr bruges mange resurser på udvikling af udstyr, som simulerer de omgivelser, som programmet skal fungere i. Det bør overvejes, om ikke der er økonomi i at udvikle et omgivelsessystem, som ville sikre en ensartet arbejdsgang for alle projektdeltagerne, og som kan give en påvirkning af produktet, som svarer til den virkelige verdens.

Interaktive testkørsler

En testkørsel kan designes, så den afvikles interaktivt, ved at testeren manuelt påtrykker testinput og løbende observerer testoutput. Testeren logger resultaterne enten med papir og blyant eller ved at

udskrive skærmbilleder, hvis det er muligt. Interaktive testkørsler kræver at testeren arbejder disciplineret efter en testspecifikation, og det kan være en tidskrævende og beværlig arbejdsform. Ofte er det imidlertid den enkleste og billigste test at udvikle, især hvis der er tale om et slut-produkt uden tastatur, skærm og disk-faciliteter, som f.eks. en vaskemaskinestyring.

Automatiske testkørsler

Testkørsler kan imidlertid også designes, således at en test afvikles helt eller delvist automatisk, således at alt hvad testeren skal huske er, at give en simpel kommando: KØR TEST27. Det kræver omfattende testprogrammer, som selv henter testinput fra en testcase-fil, og som opsamler testresultaterne på en fil, hvorfra de senere kan udskrives med testnummer, dato, osv. Den helt fuld-automatiske test afslutter testkørslen med en automatisk sammenligning mellem de forventede og de opnåede resultater.

Det kræver mere udvikling at satse på automatiske testkørsler, men det gør selve testen effektiv og vel-defineret. Især ved efterfølgende tests efter ændringer i vedligeholdelsesperioden er det en meget sikker og effektiv teknik.

Test-instrumentering

Man kan ikke altid nøjes med det egentlige output fra programmet. Der kan ske ting inde i programmet, som skal synliggøres. Til dette anvendes testinstrumentering eller debuggere. Ved testinstrumentering forstås indsættelse af specielle kodelinjer i kildeteksten i testøjemed. Der kan f.eks. instrumenteres med det formål at udskrive vigtige parametre på centrale punkter i programmet eller at holde statistik på antal gange, en monitor er blevet kaldt.

Testinstrumentering indsættes permanent, således at der ikke manuelt skal ændres frem og tilbage i kildeteksterne. De testinstruktioner, der skal udføres, kan være betingede på kørselstidspunktet vha. IF-sætninger:

```
IF TEST1 THEN <udskriv en enkelt parameter>;
IF TEST3 THEN <udskriv en hel tabel>;
```

Testinstruktionerne kan også være betingede på oversættelses-tidspunktet, idet de fleste oversættere har kommandoer, som muliggør, at man ved oversættelsen i en slutversion kan udelade testinstrumenteringen. Det kan være nødvendigt af tids- og pladsmæs-sige årsager. Det er imidlertid vigtigt at overveje hvilke faciliteter, man kan have glæde af under fejfinding og udførelse af ændringer i vedligeholdelsesfasen.

Debuggere

Der findes mange symbolske debuggere, som er brugbare som testværktøj, f.eks. til modultest. Forudsætningen er, at en debug-kørsel kan programmeres, så den kan gentages, at mellem-resultater kan logges på en fil, og at debugkørslen ikke kræver interaktiv operatørhåndtering ved breakpoints, etc. Modultest kan foretages ved, at man anvender en generel initialisering af modulet, f.eks. ved at parametre tildeles testværdier svarende til et testdriverprograms funktion. Ved hjælp af log-faciliteten kan man så ved breakpoints udlæse parametre til log-filen sammen med de ønskede kommentarer.

2.4 Testbemanding

Det er spændende og kreativt arbejde at udvikle en god test, men man må se i øjnene, at det er mindre spændende at udføre mange siders testanvisninger. Det er vigtigt, at man sørger for, at også de kedelige aktiviteter bliver gennemført. Dette kan f.eks. gøres ved aktivt at fordele testaktiviteterne i et projekt til flere projektdeltagere, f.eks.

- én person er ansvarlig for accepttest
- én person er ansvarlig for integrationstest

Det skal ikke være den samme person, som udvikler og tester et modul. Modultestarbejdet uddelegeres efter bytteprincippet: "Tester du mit modul, så tester jeg dit". I projektplanen er der således to personer på et modul:

- én person er ansvarlig for moduldesign og kodning
- én person er ansvarlig for modultesten.

Det er testerens hovedopgave at planlægge gode tests og finde mange fejl. Det er mest optimalt at sende fundne fejl, mangler og uklarheder tilbage til udvikleren, som er den, der hurtigst og bedst kan besvare spørgsmål og rette fejl.

3. SPU-tests

Denne vejledning omhandler fire typer test, nemlig:

- Modultest
- Modulintegration
- Procesintegration
- Accepttest

Disse beskrives i de følgende afsnit. Dog beskrives modul- og procesintegration delvis under ét.

3.1 Modultests

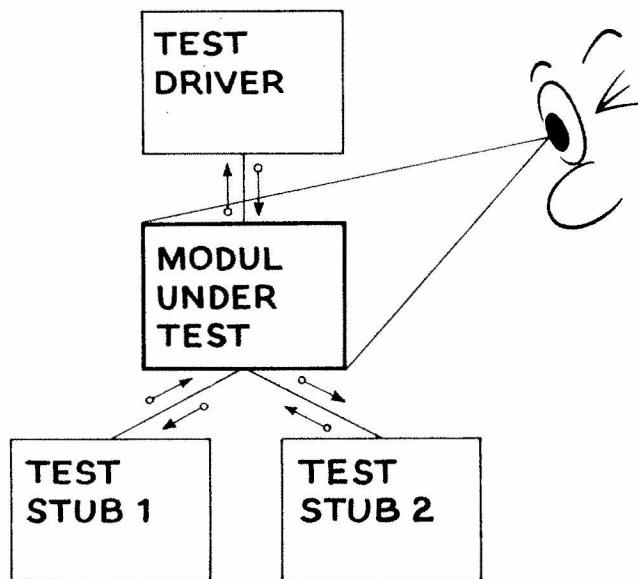
Formålet med modultest er at teste hvert enkelt modul for således at sikre, at modulet fungerer som beskrevet i modulspecifikationen. En modultest skal også afprøve grænsefladen og den interne, logiske struktur af programmet, således at alle kalde- og returnparametre anvendes, at alle instruktioner bliver udført mindst én gang, og at alle grene, f.eks. i en IF-sætning, bliver udført.

Modultest finder typisk sted i en værtsmaskine i kontrollerede omgivelser. Afhængigt af modulets placering i et modulhierarki, skal der bruges testdrivere og stubbe. Testdriveren aktiverer modulet og modtager det svar, som modulet eventuelt afleverer. Modulet kan f.eks. være en procedure. Testdriveren kalder proceduren med passende parametre så ofte, som testen angiver. Hvis den procedure, der skal testes, kalder andre procedurer uden for modulet, skal der skrives teststubbe, som kan fungere, som om procedurerne på det lavere niveau var til stede, f.eks. ved at returnere parametre. Når proceduren, der skal testes, er færdig med sin opgave, returnerer den til testdriveren.

Ofte kan store dele af testdrivere og teststubbe genbruges fra modultest til modultest med få ændringer. Det vil spare tid for den enkelte udvikler, men kræver at projektgruppen i fællesskab har fastlagt et testkoncept.

Det kan somme tider lade sig gøre at anvende et debuggerprogram som både driver og stub. Hvis der findes faciliteter til at gemme testinput- og testoutputdata, kan testen gøres delvis automatisk.

Nogle moduler har grænseflade ud til ydre enheder, f.eks. et printermodul eller i PM-eksemplet Stuekoncentratormodulet. I det tilfælde er der to argumenter, som man må veje imod hinanden: Er det ønskeligt at kunne afprøve programmet helt uden den ydre enhed, eller er det mest hensigtsmæssigt at teste med den sande hardware?



Figur 3. Modultestomgivelser.

Det er et stort arbejde at modulteste alle moduler, men det kan ofte betale sig, idet moduler, som er grundigt modultestede, meget sjældent volder kvaler i det senere udviklingsforløb. Man kan imidlertid vælge ikke at modulteste moduler, som skønnes at være enkle og ukritiske. Det ændrer ikke ved det krav, at alle moduler skal gennemgå review eller kodegranskning før de integreres.

3.2 Integrationstests

Der er principielt to forskellige aktiviteter forbundet med integration, nemlig dels at sætte enheder sammen dels at teste de nye enheder.

Hovedformålet med de to SPU-integrationsfaser, hvor hhv. moduler integreres til processer og processer til det færdige program er at afprøve samspillet mellem først moduler dernæst processer i det stadigt voksende program.

I praksis har integrationstests ofte også et andet formål, nemlig specifikt at afprøve den sidst integrerede enhed, idet man så anvender de allerede testede enheder som hjælp i testen. Derved sparer man f.eks., at udvikle specielle testdrivere og stubbe til alle moduler. Det er imidlertid en grundregel, at to enheder ikke må kædes sammen, før mindst én af dem er afprøvet.

Det er en vigtig del af integrationsarbejdet, at planlægge, hvordan

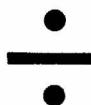
integrationen kan foregå på den mest hensigtsmæssige måde. Principielt er der to forskellige måder at integrere på:

Trinvis integration:



Enheder tilføjes én efter én med en integrationstest mellem hver udvidelse. Dette gør det muligt at fokusere på grænsefladen til den nye enhed og eventuelt på den nye enhed. Det er sandsynligt, at fejl begrænser sig til det nye delsystems funktioner. Desuden er det muligt at anvende allerede testede funktioner til at understøtte testen. Dette er den anbefaelsesværdige metode.

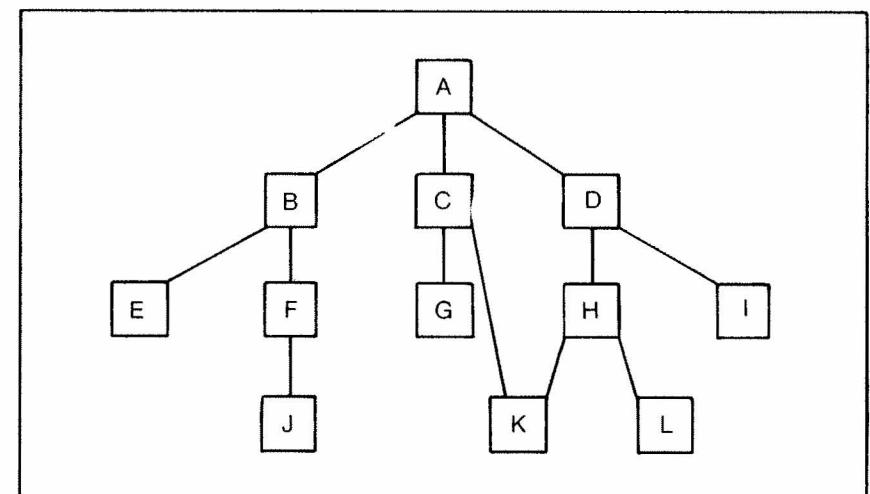
Samlet integration:



Efter individuel test i isolerede omgivelser samles alle enheder på én gang til det færdige system. Denne metode kaldes også 'BIG BANG'-test. Metoden kan ikke anbefales, fordi problemer, der opdages, er vanskelige at isolere til et bestemt modul.

Modulintegration

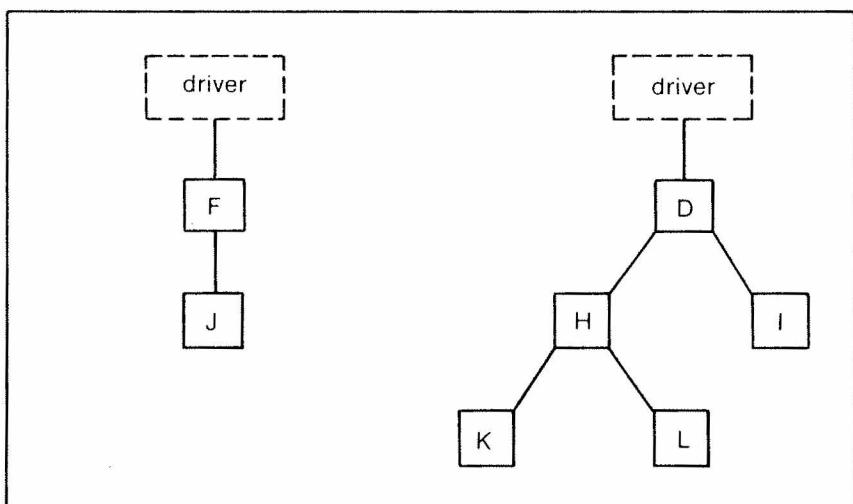
Grundlaget for modulintegrationen er moduloversigten i programdokumentationen. Modulerne vil normalt indgå i et modulhierarki, og svarende hertil kan integrationen foretages forskelligt, nemlig bottom-up eller top-down. Figur 4 viser et eksempel på et modulhierarki.



Figur 4. Et eksempel på et modulhierarki, som skal trinvis integreres.

Bottom-up-integration går ud på at teste og integrere modulerne fra bunden til toppen af hierarkiet. Der startes med de moduler, der ikke kalder andre. Da moduler på højere niveauer ikke er testet, kræver dette, at der udarbejdes testdrivere til hver enkelt modul, inden det kan testes. Figur 5 viser to integrationstrin i en bottom-up-integration. Her er brug for udvikling af to drivere, men ikke for stubbe.

En ulempe ved bottom-up-test er det sene tidspunkt, hvor hovedprogrammet, og dermed totalfunktionen af programmet bliver afprøvet samlet. En fordel er, at input/outputenheder tidligt kan anvendes til test.

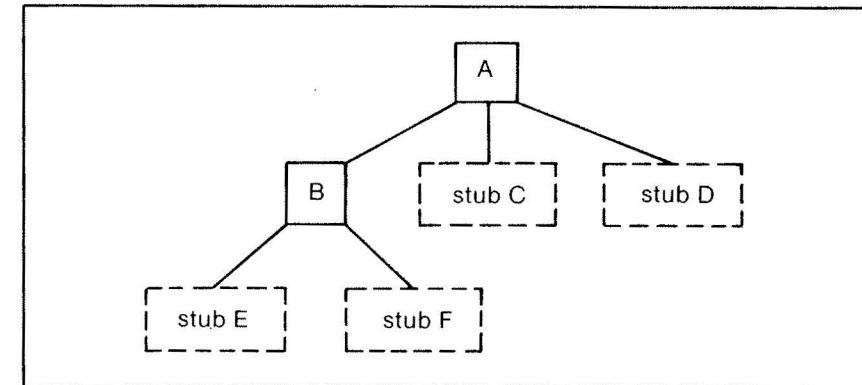


Figur 5. To integrationstrin i en bottom-up-integration af modulhierarkiet i figur 4.

Top-down-test går ud på at afprøve modulerne fra toppen til bunden af hierarkiet. Der startes med hovedprogrammet i toppen af hierarkiet. Dette medfører, at moduler, som kalder underliggende moduler, må have tilfredsstillet disse kald. Underliggende moduler simuleres med teststubb. En simpel stub vil være en RETURN-instruktion, men da der i et undermodul ofte sker en behandling af input og en returnering af output vil en simulering af den forventede funktion være at foretrække. Derfor er stubbe ikke helt simple at udarbejde.

Fordelen ved top-down-test er den tidlige integration af hovedmodul og første niveau af undermoduler. Programmet vil som helhed kunne afvikles, dog med en del simulerede funktioner.

En ulempe ved top-down-test er det faktum, at et modul først bliver gennemtestet, når det underliggende modul kædes på.



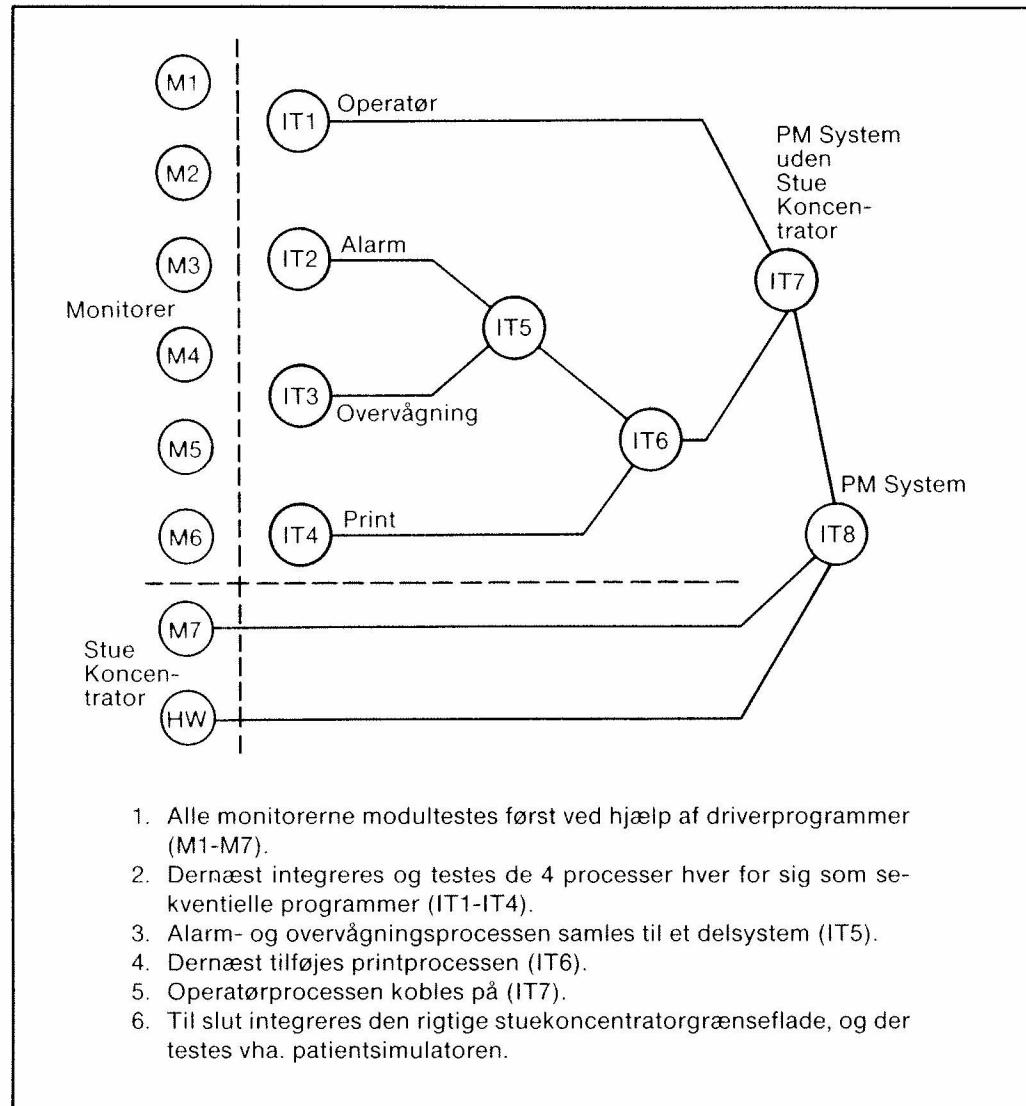
Figur 6. Et integrationstrin i en top-down-integration af modulhierarkiet i figur 4.

I praksis vil man ofte vælge et kompromis mellem de to metoder, idet rækkefølgen og strategien for integration af modulerne er afhængig af mange faktorer:

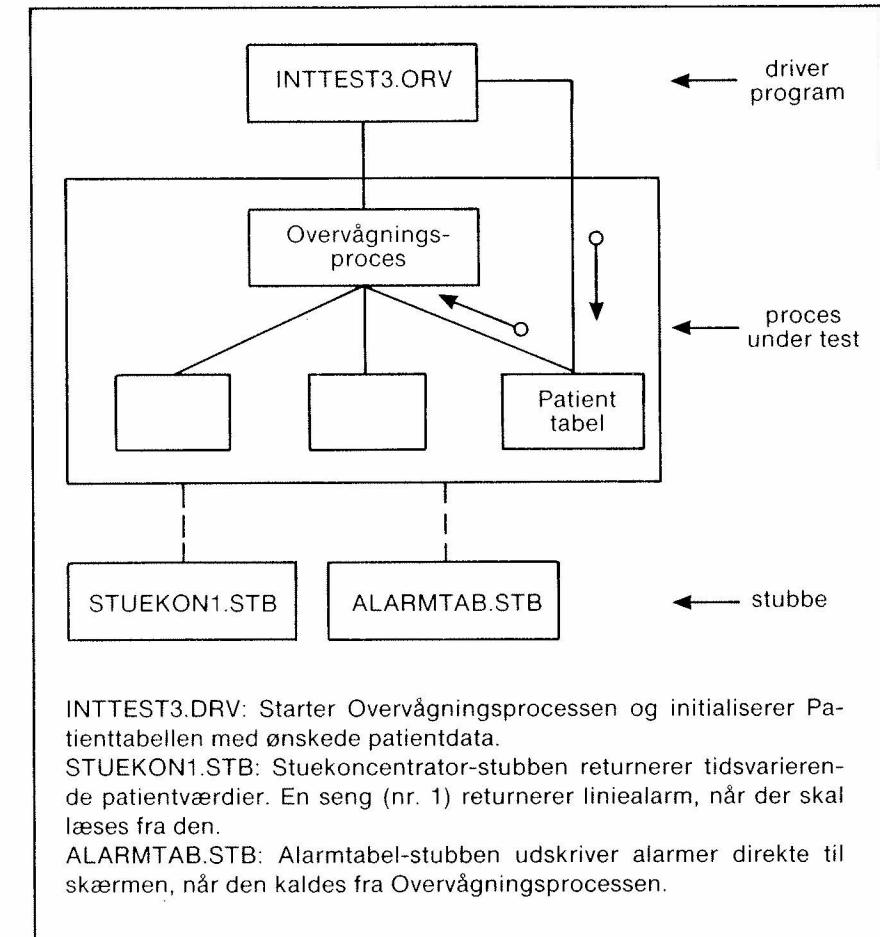
- Programmet, der skal testes, og de ydre enheder knyttet til det.
- Hvilke moduler bliver færdige først?
- Er hardwaren tilgængelig? Er den under udvikling, eller bliver den først leveret i den 11. time?
- Testfaciliteterne. Kan man få noget ud af at teste, hvis nogle enheder mangler?
- Er der funktioner, som gerne skulle demonstreres tidligt?

Procesintegration

Før processerne integreres, er det hensigtsmæssigt at de fælles kommunikationsmoduler, f.eks. i form af monitorer, er modultestede og til rådighed for procesintegrationstesten. Processerne samles trinvis således at man kun i testen skal fokusere på kommunikationen mellem to processer. Kommunikationen imellem to processer kan kun til en vis grænse testes vha. systematisk test, idet parallelisme og det tidstro aspekt medfører, at en test ikke kan gentages på nøjagtig samme måde.



Eksempel 2. Procesintegrationsplan for Patientmonitorsystemet, som indgår i programdokumentationen.



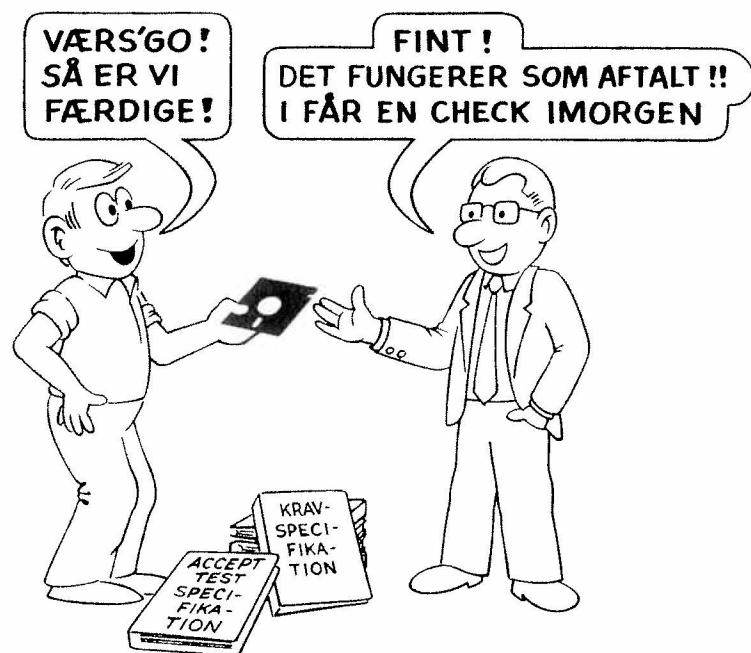
Eksempel 3. Integrationstest af trinnet IT3 med den nødvendige driver og de nødvendige stubbe.

3.3 Accepttest

Accepttesten er den afsluttende test af programmet, og i SPU-sammenhæng har den to forskellige formål. Den anvendes til at demonstrere over for kunden, at de stillede krav er opfyldt og at programmet kan tages i brug. Det betyder formelt at udviklingsfasen afsluttes og det udløser ofte en større del af betalingen. Men projektgruppen anvender også selv testen til at sikre sig at det udviklede program er, som det skal være. Det er vigtigt for den fortsatte arbejdsglæde, at projektgruppen får mulighed for at vurdere sit eget arbejde som en helhed og markere afslutningen på udviklingsfasen.

Accepttesten planlægges og udføres med så høj grad af kundeindflydelse som muligt. Det burde altid være kunden, der specificerede

acceptttesten. Hvis kunden ikke ønsker at specificere en accepttest, bør projektgruppen selv gøre det. Accepttesten bygger på kravspecifikationen med både funktionelle krav og krav til kvalitetsfaktorerne (pålidelighed, vedligeholdsesvenlighed, etc.). Den detaljerede analyse af kravspecifikationen, som er nødvendig for at definere en accepttest, er med til at skabe en dyberegående forståelse for det nye program.



Figur 7. Det er nødvendigt med et aftalegrundlag, som kan sikre at afleveringen af produktet kan ske til alles tilfredshed.

Det er klart, at projektgruppen selv anvender den specificerede accepttest internt i den sidste del af projektet, før testen anvendes som demonstration over for kunden.

Foruden den funktionelle test omfatter accepttesten test eller demonstration af de ønskede kvalitetsfaktorer. Kvalitetsfaktorerne er specificeret ved projektstart i kravspecifikationen, og selv om det er svært, er det vigtigt at forsøge at definere nogle destruktive testcases, som tvinger programmet ud i ekstreme situationer. Følgende uformelle liste af tests kunne måske give ideer til denne del af accepttesten:

- Stress-test. Virker programmet under de værst tænkelige påvirkninger?
- Volumentest. Kan programmet håndtere maksimalt input gennem lang tid?

- Brugertest. Er systemet tilpas brugervenligt?
- Sikkerhedstest. Kan man snyde systemet? Er data sårbarer?
- Test af ydeevne. Opfylder systemet krav til svar-tider, også ved spidsbelastninger?
- Lager-test. Bruger programmet mere lager end estimeret? Hvor mange procent af lageret er i alt i brug?
- Konfigurationstest. Skal systemet kunne fungere i forskellige konfigurationer (anden lagerstørrelse, anden inputenhed, etc.)?
- Kompatibilitetstest. Skal det nye program kunne køre på forskellige datamater? Skal det erstatte et gammelt program i alle henseender?
- Pålidelighedstest. Er der specielle krav til pålidelighed? Kan programmet køre i normal drift, f.eks. 48 timer uden fejl?
- Fejlbehandlingstest. Kommer systemet korrekt ud af enhver fejl-situation (strømsvigt, fejl på ydre enheder, etc.)?

4. Testforberedelse og -udførelse

Modultest, integrationstest og accepttest bygges trinvis op ved hjælp af de samme testaktiviteter:

Forberedelse:

- SPECIFICER testemner. *Hvad* skal egentlig testes?
- DESIGN testen. *Hvordan* skal testen køres, hvilke testomgivelser og hvilke testcases?

Udførelse

- IMPLEMENTER testen. Skriv testprogrammer og sæt testomgivelser op.
- KØR testen. Påtryk testinput og sammenlign resultatet med det forventede.
- EVALUER testforløbet. Her skal gruppen tage ved lære af den netop afsluttede test.

Tidsmæssigt falder forberedelsen af en test i den tidlige fase, dvs. for modultest i moduldesignfasen, for integrationstest i program- og procesdesignfasen, og for accepttest i kravspecifikationsfasen.

Grundspecifikationen svarende til de forskellige tests er vist herunder:

Test	:	Grundspecifikation
Accepttest	:	Kravspecifikation
Procesintegration	:	Programdesign (kap. 2,3 og 4 i Programdokumentationen)
Modulintegration	:	Procesdesign (Afsnit 6.4 ff i Programdokumentationen)
Modultest	:	Modulspecifikation

Udførelsen ligger i den egentlige testfase.

4.1 Specificer testemner

- IDENTIFICER testemner, dvs. de forskellige krav, der skal tes- tes ud fra en grundig gennemlæsning af grundspecifikationen med test for øje.

- IDENTIFICER uklarheder, mangler, modsigelser, etc. Et andet krav kunne være 'Patientopsætningen skal kunne tåle et strømsvigt'. Svaret på dette er ikke trivielt, og der må stilles spørgsmålstege ved, om et sådan krav er tilstrækkeligt specificeret.
- IDENTIFICER vigtige kombinationer og rækkefølger, f.eks. 'start' før 'stop', 'indsætte' før 'fjerne' og 'indsætte' med og uden aktiv overvågning.
- IDENTIFICER andre egenskaber, som skal testes i denne test (hastighed, størrelse, etc.).
- IDENTIFICER forskellige tilstande (servicetilstand, driftstilstand, etc.).
- IDENTIFICER, hvad der *ikke* skal testes, f.eks. operativsystem og biblioteksrutiner.
- IDENTIFICER hvilke ydre omstændigheder, der er med til at specificere formål og godkendelseskriterier for netop denne test. Man kan f.eks. være bundet af et krav fra kunden om at følge en bestemt IEEE-standard, som vil være bestemmende for hvordan testarbejdet skal gribes an.
- SPECIFICER testemnerne ved at udarbejde en nummereret liste, ordnet som følge af den foregående analyse. Listen kan opfattes som en checkliste over alt, der skal testes (se eksempel 4).

Udfra modulspecifikationen til IndlæsPatientVærdier fås følgende testemner til black-box-testen:

'Funktion's-afsnittet giver følgende funktioner, der skal testes:

- TE1: der indlæses patientværdier fra stuekoncentrator,
- TE2: der kan optræde alarm ved 1., 2., 3. eller 4. læsning,
- TE3: tidspunktet registreres,
- TE4: patientværdierne skal skaleres.

'Input'-afsnittet giver følgende testemner:

- TE5: der kan vælges forskellige sengenumre (1-6)
- TE6: der kan vælges hvilke patientværdier, der skal læses (TEMP,PULS,BPSYS og BPDIA)
- TE7: der kan vælges hvor mange patientværdier, der skal læses (1-4)

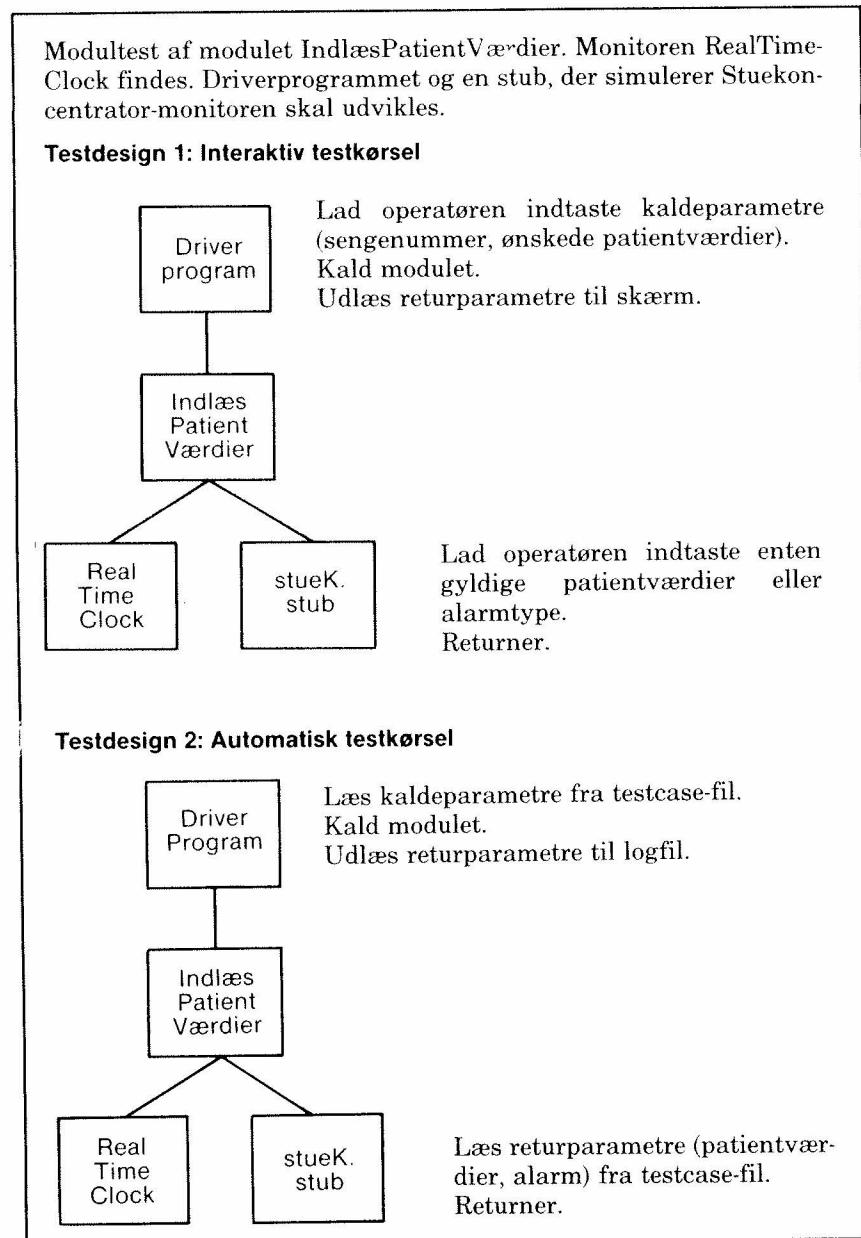
'Output'-afsnittet giver følgende testemner:

- TE8: der returneres læste patientværdier med tidsangivelse (1-250, 19.1-44.0)
- TE9: der returneres alarmtype (ingen alarm, LINIEALARM1-3)
- TE10: modulet selv er en boolsk funktion, der returnerer med alarmstatus (TRUE, FALSE)

Eksempel 4. Eksempel på liste af testemner, som modultesten af modulet IndlæsPatientVærdier skal omfatte.

4.2 Design testen

- **STRUKTURER** testen. Er listen af testemner så omfattende, at en yderligere opdeling af testen skal foretages? Er der testemner, der naturligt kan testes sammen?



Eksempel 5. Et eksempel på at testomgivelserne kan opbygges enten til interaktive testkørsler eller til automatiske testkørsler.

- **DESIGN** testomgivelser Hvorledes påtrykkes testinput, og hvorledes opfanges testresultaterne? Hvilke testomgivelser i form af testprogrammer og hjælpeudstyr er nødvendige (stubbe, drivere, ekstra printer, patientsimulator, etc)? Hvilken grad af automatisering ønskes? Skal testes kunne køres ved en simpel "RUN TEST1"-kommando, eller skal testoperatøren taste al testinput ind over tastaturet? Findes der testfaciliteter, der kan genbruges her? (se eksempel 5)
- **IDENTIFICER** vanskeligheder. Hvis en test er vanskelig (dyr!!) eller umulig at gennemføre, skal det diskuteres. Der kan være problemer i enten formulering af krav eller i designet af systemet. Usynlige datastrukturer synliggøres, f.eks. ved hjælp af testinstrumentering. Ofte kan små indbyggede faciliteter til opsamling af interessante data lette eller muliggøre en test.
- **DESIGN** testcases, dvs. udvælg testinput og FORVENTET OUTPUT til de uafhængige tests, som testforløbet består af. Testcases skal dække både gyldigt og ugyldigt input og output, således at både almindelig brug og fejlhåndtering testes. Det er ikke nødvendigt på dette tidspunkt at udfylde de nøjagtige testdata, som skal anvendes, ligesom det er tilstrækkeligt f.eks. at angive, at forventet output er en fejlmeldelse. I kapitel 5 er beskrevet nogle metoder til udvælgelse af testcases.

4.3 Implementer testcases og testomgivelser

- **IDENTIFICER** nye testemner, som er kommet til siden forberedelsen ud fra versionshistorien på grundsifikationen, og opdater testsifikationen tilsvarende.
- **UDFYLD** testcases med nøjagtigt det input, der skal anvendes i testen og med det forventede output, som nu kendes fuldt ud.
- **SUPPLER** med testcases valgt ud fra viden om kodens struktur, f.eks. at CASE-konstruktioner skal gennemkøres med alle mulige valg, at loops skal afprøves på grænserne etc.
- **BESKRIV** hvorledes testomgivelserne skal opbygges og testen køres, dvs. skriv en brugsanvisning for kørsel af testene så grundigt, at en anden kan køre testen.
- **CHECK** at alle nødvendige testfaciliteter er til stede.
- **BEGYND** på testrapporten.

4.4 Kør testen

- **KØR** testen som beskrevet i testsifikationen og registrer resultaterne. Hvis der findes disk- og printfaciliteter på systemet,

logges resultaterne og printes ved lejlighed. Det er hensigtsmæssigt også at udskrive dato, overskrift og anden identifikation. Foruden selve resultatet er det hensigtsmæssigt også at udskrive påtrykt testinput og det forventede resultat. Hvis resultaterne ikke kan printes, må en tester observere og nedskrive resultaterne. Der kan eventuelt afsættes et felt til dette i testspecifikationen, således at testeren noterer i en dateret kopi af testspecifikationen. Der kan også tankes på alternativer som f.eks. Polaroid-fotografering af skærme og displays. De registrerede resultater indsættes i testrapporten.

- IDENTIFICER afvigelser fra en normal kørsel, f.eks. på grund af en triviel operatørfejl. Det checkes, om dette har nogen indflydelse på testens værdi.
- SAMMENLIGN i fred og ro resultaterne med de forventede resultater. Ved afvigelser udfyldes en problem/aendringsrapport som indsættes i kopi i testrapporten. Det vil normalt være mest optimalt, at udvikleren selv retter sine fejl.
- CHECK om betingelserne for normal testafslutning foreligger. Hvis der er fundet fejl, rettes de, og testen køres igen. Hvis der skal opnås et bestemt testdækningskriterium, f.eks. at alle instruktioner skal have været kørt af testen, måles dette, og der laves eventuelt flere tests. Overvej muligheden for at frigive en midlertidig version.
- UDFYLD testrapportens konklusion.

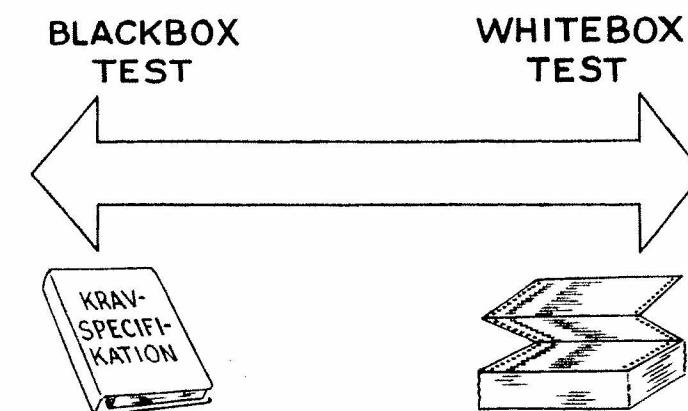
4.5 Evaluering af testforløbet

- EVALUER testforløbet med henblik på fremtidige projekter:
 - Er den tid, der er brugt på testarbejdet, passende sammenlignet med resultatet?
 - Blev der fundet nogle grove fejl? Hvilken test fandt de fleste eller de værste?
 - Var der tests, der føltes overflødige?
 - Blev der overset fejl? Kunne det have været undgået?
 - Har projektgruppen tillid til den afleverede software eller ventes der mange fejmeldinger?
 - Var testomgivelserne OK?
 - Hvad kan der gøres for at gøre det bedre næste gang?

5. Udvælgelse af effektive testcases

En vigtig del af testarbejdet er at udvælge testcases, dvs. testinput og det forventede output, der svarer dertil. For at en test kan blive effektiv, skal der udvælges testcases, der finder mange fejl. Da testeren også skal være realistisk i praksis, kan det ikke nytte at prøve med alle mulige værdier af alle parametre. Der skal udvælges et begrænset, men udfra et pragmatisk synspunkt, dækkende sæt af testcases.

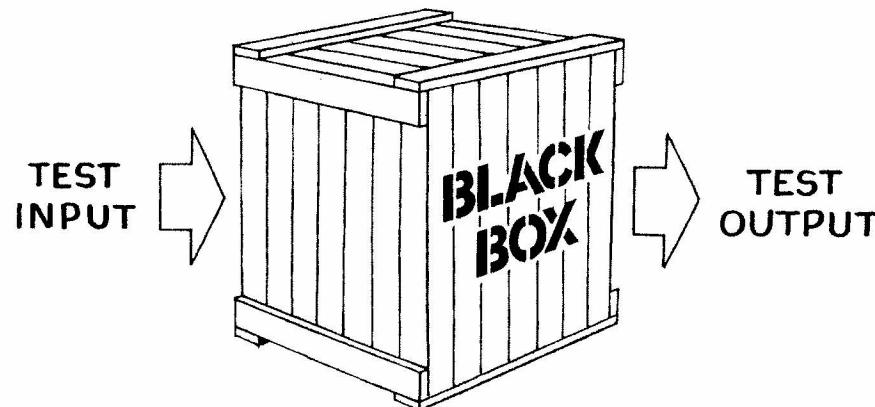
Et sæt af testcases siges at være dækkende, når alle funktioner beskrevet i grundsættelsen er testet og når hele programmetts struktur er blevet testet. Der anvendes forskellige metoder til at udvælge testcases som er dækkende udfra et funktionelt synspunkt (blackboxtests) og udfra et programmæssigt synspunkt (whiteboxtests).



Figur 8. Blackboxtest contra whiteboxtest.

5.1 Design af testcases til blackboxtests

En blackboxtest skal godtgøre, at programmet, processen eller modulet virker efter hensigten set udefra. Det er således en test op imod specifikationerne uden kendskab til, hvorledes enheden internt er designet eller kodet. Man skal ved at påvirke enheden med stimuli kunne se en bestemt opførsel som beskrevet i specifikationen.



Figur 9. Blackboxtest - funktionel afprøvning

Testcases til blackboxtest udledes ud fra grundsifikationen. Arbejdsgangen er følgende:

1. Identificer al muligt input og output svarende til listen af testemner.
2. Identificer det gyldige og det ugyldige område for al input og output.
3. Opdel det gyldige og det ugyldige område i klasser, således at al input inden for en klasse skønnes at blive behandlet ens af programmet. Må et nummer antage værdierne fra 0 til 999, er der tale om mindst tre klasser, nemlig en klasse af gyldigt input ($0 \leq x \leq 999$) og to klasser af ugyldigt input ($x < 0$ og $x > 999$). Hvis grundsifikationen angiver, at numre fra 0 til 10 har én funktion, numrene fra 11-999 en anden funktion, er der tale om to klasser af gyldigt input, som begge skal testes. Ofte kan ugyldigt input ikke forekomme, men det er vigtigt at overveje muligheden.
4. Indtil alle klasser med gyldigt input er dækket, designes testcases, som hver især dækker så mange klasser som muligt. Hvis der f.eks. skal testes 2 forskellige input-parametre, 'Sengenummer' og 'Valg af patientværdi', kan man komprimere antallet af tests ved at teste både 'Sengenummer' og 'Valg af patientværdi' i en og samme test.
5. For hver klasse med ugyldigt input designes testcases, som dækker én og kun én klasse. Man kan ikke forvente at et program kan behandle mere end én fejl på én gang.
6. Nu skal der suppleres med grænseværdier, som ofte giver fejl. Det kunne f.eks. være
 - første element
 - sidste element

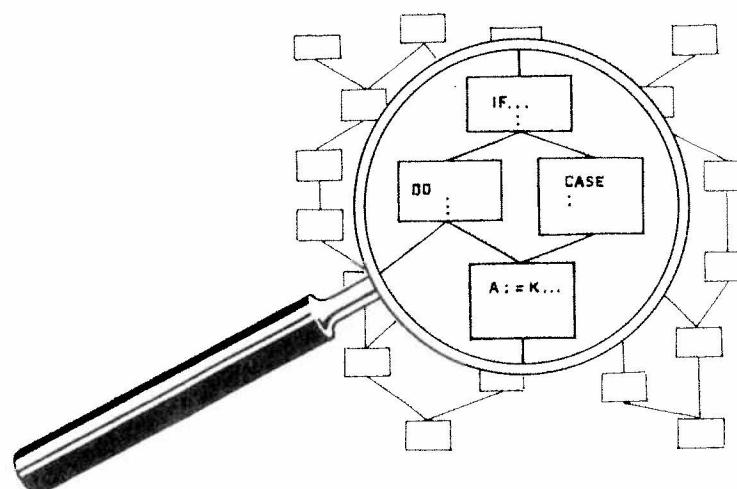
- maksimumsværdi
 - minimumsværdi
 - én over grænsen
 - én under grænsen
 - tallet nul som parameter
 - ASCII-tegnene Null og Blank
 - tomme elementer
7. For hver grænseværdi genereres en ny testcase, som dækker én og kun én grænseværdi.
 8. Til slut suppleres med fejlgætning, som er en meget vigtig del af en god test. Fejlgætning finder mange fejl.
 9. Til både gyldige og ugyldige testcases specificeres det forventede resultat.

Testemne	Arbejdstrin	Gyldigt input	Ugyldigt input
Puls, som er en af de fire patientværdier. Det er en digital 8-bits værdi indlæst fra stuekoncentratoren.			
Klassedeling	$0 < x \leq 250$ (T1)	$x=0$ (T2) $250 < x \leq 256$ (T3)	intet nummer-ikke mulig
Grænseværdi	$x=1$ (T4) $x=2$ (T5) $x=250$ (T6) $x=249$ (T7)	$x=0$ (T2) $x=-1$, ikke mulig $x=251$ (T8) $x=255$ (T9)	
Fejlgætning		Null, ikke mulig blank, ikke mulig	
Valg af patientværdi, som er et boolsk array med 4 parametre.			
Klassedeling	Fra 0000 til 1111 (T1)	ikke mulig	
Grænseværdi	0000 (T2) 0001 (T3) 1110 (T4) 1111 (T5)	ikke mulig	
Fejlgætning	0001 (T6) 0010 (T7) 0100 (T8) 1000 (T9)	0010 (T7) 0100 (T8) 1000 (T9)	

Eksempel 6. Et eksempel på første trin i designet af tests til 'Puls' og 'Valg af patientværdi'.

5.2 Design af testcases til whiteboxtest

En whiteboxtest er en test af den logiske struktur af implementeringen. Den er derfor rettet imod selve programstrukturen dvs. instruktioner, forgreninger, tilstande, etc.



Figur 10. Whiteboxtest - strukturel afprøvning

Testcases vælges ud fra moduldesign eller kildetekst indtil et af følgende kriterier er opfyldt:

1. Instruktionsdækning. Alle instruktioner gennemløbes mindst én gang.
2. Forgreningsdækning. Alle grene i koden gennemløbes mindst én gang. Det vil f.eks. sige, at

IF $a < 4$ THEN ...
afprøves med både $a < 4$ og $a \geq 4$.
3. Betingelsesdækning. Alle sammensatte betingelser afprøves, indtil alle betingelser er dækket. Instruktionen

IF flag=0 AND $a < b$ THEN ...
afprøves med

 1. flag=0 og $a < b$
 2. flag=0 og $a \geq b$
 3. flag=1 og $a < b$
 4. flag=1 og $a \geq b$

Testdækning kan måles automatisk vha. et værktøj, som registrerer hvilke instruktioner, grene og betingelser, der gennemløbes un-

der kørsel af programmet med de anvendte testcases. Det manuelle alternativ er skrivebordskørsel med de udvalgte testdata samtidig med, at der markeres i marginen på kildeteksten med kulørte filtpenne. Det kan give et godt fingerpeg om, hvilke dele af koden, der ikke er omfattet af testen.

Det kan være vanskeligt at opnå 100% betingelsesdækning, men det bør være et klart krav, at alle instruktioner gennem test har været eksekveret mindst én gang.

Testdækning er vigtig som mål for testens kvalitet, men man skal selvfølgelig huske også at kontrollere, at de egentlige testresultater svarer til det forventede.

5.3 Strategi for udvælgelse af testdata

En komplet test består af både blackboxtest og whiteboxtest.

En modultest udvikles som blackboxtest, indtil alle ydre krav til modulet bliver testet. Herefter suppleres med testdata til whiteboxtest, indtil det ønskede dækningskriterium er opfyldt.

Integrationstest består af blackboxtest suppleret med whiteboxtest, som tester implementationsdetaljer. Det kunne være en datastruktur, som kræver ekstra tests til at afprøve:

- initialiseringen
- fyld et element i
- ændre et element
- tage et element ud
- fyld op med elementer indtil overløb
- osv.

Accepttest er altid kun en blackboxtest.

Det vil til enhver test være tilladt, ja, endog anbefalelsesværdigt at supplere med ikke-kategoriserede testdata baseret på erfaring, intuition og nysgerrighed.

Metoder til udvælgelse af testdata er beskrevet kort i /Klim84/ og detaljeret med eksempler i /Myers79/.

6. Testdokumentation

Det er et vigtigt skridt i retning af systematisk test at erkende nødvendigheden af at skrive ting ned. Ved at skrive testdokumentation opnås:

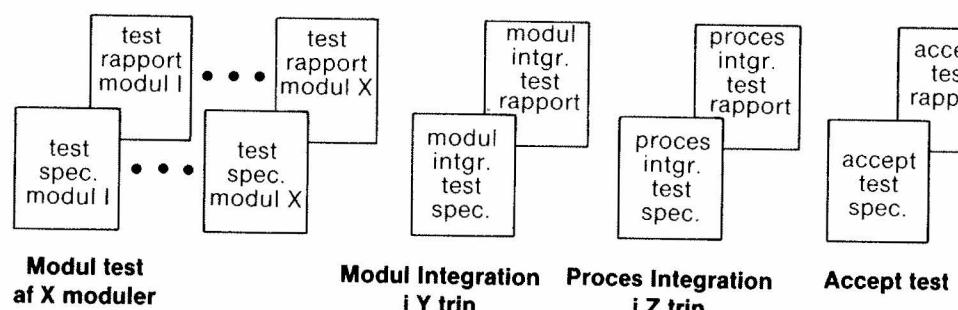
- Bedre oversigt over opgaven, både over testopgaven og over den egentlige udviklingsopgave.
- Tests kan løbende forbedres.
- Mulighed for diskussioner og/eller reviews af et testforløb.
- Sammenligning af det forventede resultat med det opnåede resultat kan foregå i fred og ro.
- Mulighed for gentagelse af en test.
- Mulighed for at analysere, om en test var dækkende eller ej.
- Solidt grundlag for en formel godkendelse af test.

Testdokumentationen består af følgende dele:

- Et antal testspecifikationer, idet en testspecifikation dokumenterer alle forberedelser til en bestemt test.
- Et antal testrapporter, idet én testrapport indeholder resultaterne fra testkørslerne defineret i én testspecifikation. Da der kan tænkes flere kørsler af en test, kan der være mange testrapporter.

Testdokumentationen for et projekt kunne bestå af følgende:

- X modultestspezifikationer plus testrapporter, hvis der er X moduler.
- 1 modulintegrationsspezifikation plus testrapport, som beskriver modulintegrationens Y trin.
- 1 procesintegrationsspezifikation plus testrapport, som beskriver procesintegrationens Z trin.



Figur 11. Testdokumentation for et projekt.

- 1 accepttestspezifikation plus accepttestrapport.

I SPU-sammenhæng er dokumentationen organiseret således, at kun accepttestdokumentationen fremstår som selvstændige dokumenter, mens de øvrige dokumenter indgår som afsnit i programdokumentationen.

Indholdsfortegnelse for testspezifikationer og testrapporter er beskrevet i afsnit 6.2 og 6.3.

6.1 Referencedokumenter

Følgende dokumenter er en forudsætning for systematisk planlagt test:

- Kontrakt eller projektgrundlag med eventuelle krav til testforløb, godkendelseskriterier, tidsplan, o.lign.
- Kravspecifikation.
- Program- og procesdesign.
- Modulspezifikationer.
- Eventuelle firma- eller projektstandarder og retningslinier.
- Generelle standarder og retningslinier.

6.2 Indholdsfortegnelse for en testspezifikation

Indholdsfortegnelsen for testspezifikationen til modultest, integrationstest og accepttest er ens. Da målsætningen med testen og den specifikation, som ligger til grund for den enkelte test, er forskellig, bliver indholdet også forskelligt.

- | |
|-------------------------------------|
| 1. Indledning |
| 1.1 Formål |
| 1.2 Referencer |
| 1.3 Testens omfang og begrænsninger |
| 1.4 Godkendelse |
| 2. Testemner |
| 3. Testdesign |
| 4. Testimplementation |
| 5. Udførelse af test |
| Bilag |

Figur 12. Indholdsfortegnelse til en testspezifikation

1. Indledning

Her angives, hvorledes dette dokument og den test, der beskrives heri, hænger sammen med de øvrige aktiviteter i projektet.

1.1 Formål

Her præciseres formålet med den test, som er specificeret i testspecifikationen.

1.2 Referencer

Enhver test har et udviklingsdokument som baggrund. I dette afsnit identificeres den grundspecifikation, der har været udgangspunkt for denne testspecifikation.

1.3 Testens omfang og begrænsninger

Her angives, hvilke programmer, moduler eller lignende, som denne testspecifikation dækker, og hvilke den ikke dækker, hvis der kan være tvivl om det.

1.4 Godkendelse



Figur 13. Det er desværre ofte tidspunktet mere end objektive godkendelseskriterier, der bestemmer, når et produkt skal afleveres.

Her angives hvem der er ansvarlig for frigivelse eller aflevering af det testede program og hvilke kriterier der anvendes for at afgøre, om en test er bestået eller ej. Ved at opstille f.eks. det godkendelseskriterium, at denne testspecifikation skal være kørt igennem fejlfrit, undgår man at afleveringen af et program udelukkende bestemmes af, at en bestemt dato nås.

2. Testemner

Testemner er alle de ting, der skal testes. Man kunne kalde kapitlet for "testens kravspecifikation", idet det identifierer *hvad* teksten skal omfatte. Kapitlet udfyldes gennem analyse af grundspecifikationen identificeret i afsnit 1.2. Det mest praktiske er at opstille testemnerne nummereret på listeform. For hvert testemne skal der refereres tilbage til et enkelt-afsnit i grundspecifikationen.

3. Testdesign

Dette kapitel beskriver overordnet, hvorledes testen designes, således at alle testemnerne testes på den mest hensigtsmæssige måde. Beskrivelsen består af en nedbrydning til vel-afgrænsede enkelt-tests, som kan specificeres uafhængigt og udføres i vilkårlig eller veldefineret rækkefølge.

For hver test beskrives enten verbalt eller ved hjælp af en tegning af testomgivelserne, hvordan testen tænkes udført, så godt som det er muligt på planlægningstidspunktet, hvor ingen eller kun meget få implementationsdetaljer er kendt. Af hensyn til projektplanlægningen er det imidlertid vigtigt at kunne identificere, hvilke testprogrammer og hvilket hjælpeudstyr, der er behov for (stubbe, drivere, ekstra printer, patientsimulator, etc).

Ud fra en blackbox-analyse af de testemner, som den enkelte test omfatter, beskrives overordnet for hver test de forskellige testcases, som skal anvendes i testen omfattende både gyldigt og ugyldigt input.

4. Testimplementation

Svarende til testdesignet beskrevet i kapitel 3 angives for hhv. gyldigt og ugyldigt input nøjagtigt, hvordan input ser ud (Patient: PETER HANSEN, Patientnummer 637). Ligeledes skal der angives, hvordan det forventede output ser ud. I mange tilfælde vil man enkelt kunne afgøre, om det faktiske resultat er i

overensstemmelse med det forventede. I andre tilfælde, f.eks. hvor resultaterne godt kan variere inden for visse tolerancer, må dette angives.

I det omfang, hvor testdata opbevares på selvstændige datafiler, vil kapitel 4 kunne begrænses til en reference til eller kopi af disse filer.

De testcases, som tilføjes med det formål at nå et bestemt testdækningskriterium, mærkes separat, således at formålet med alle testcases er sporbart.

5. Udførelse af test

For hver test defineret i kapitel 3 angives en brugsanvisning for afvikling af testen. En sådan brugsanvisning skal indeholde:

- Hvordan bygges testomgivelserne op?
- Hvordan startes testen?
- Hvordan køres testen?
- Hvordan observeres og opsamles resultaterne?
- Hvordan afbrydes testen midlertidigt og hvordan genstartes?
- Hvordan afsluttes testen normalt?

Bilag

I bilaget kunne være beskrevet nogle af de faciliteter, der benyttes specielt i denne test, f.eks. beskrivelse af og kildetekst til testprogrammer.

6.3 Indholdsfortegnelse for en testrapport

Testrapporten behøver ikke nødvendigvis at være et selvstændigt dokument. Den vil kunne udformes som en kopi af testspecifikationen, påført testdato, kommentarer og konklusion. Indholdsfortegnelsen for en egentlig testrapport er følgende:

1. Indledning
1.1 Reference
1.2 Identifikation
2. Testresultater
3. Afgigelser og kommentar
3.1 Afgigelser fra normal afvikling
3.2 Problem/ændringsrapporter
4. Konklusion

Figur 14. Indholdsfortegnelse til en testrapport

1. Indledning

1.1 Reference

Reference til testspecifikation angives.

1.2 Identifikation

Identifikation af den testede software, nu også med faktisk versionsnummer, opbevaringsmedium osv.

De faktiske testomgivelser identificeres så præcist, at de kan genereres.

2. Testresultater

I dette kapitel registreres testresultaterne. Det vil være forskelligt fra system til system, hvorledes resultaterne kan præsenteres, så det overlades til den enkelte at lave en rimelig opdeling og formatering af resultater. Husk at for den enkelte test refereres tilbage til et underpunkt i testspecifikationen.

3. Afgigelser og kommentarer

3.1 Afgigelser fra normal afvikling

Afgigelser i afvikling af testkørslen, som f.eks. sekvenser, der er kørt om på grund af trivielle operatørfejl.

3.2 Problem/ændringsrapporter

Detekterede fejl påvist ved uoverensstemmelse mellem det forventede og det opnåede resultat registreres. En fortrykt formular til

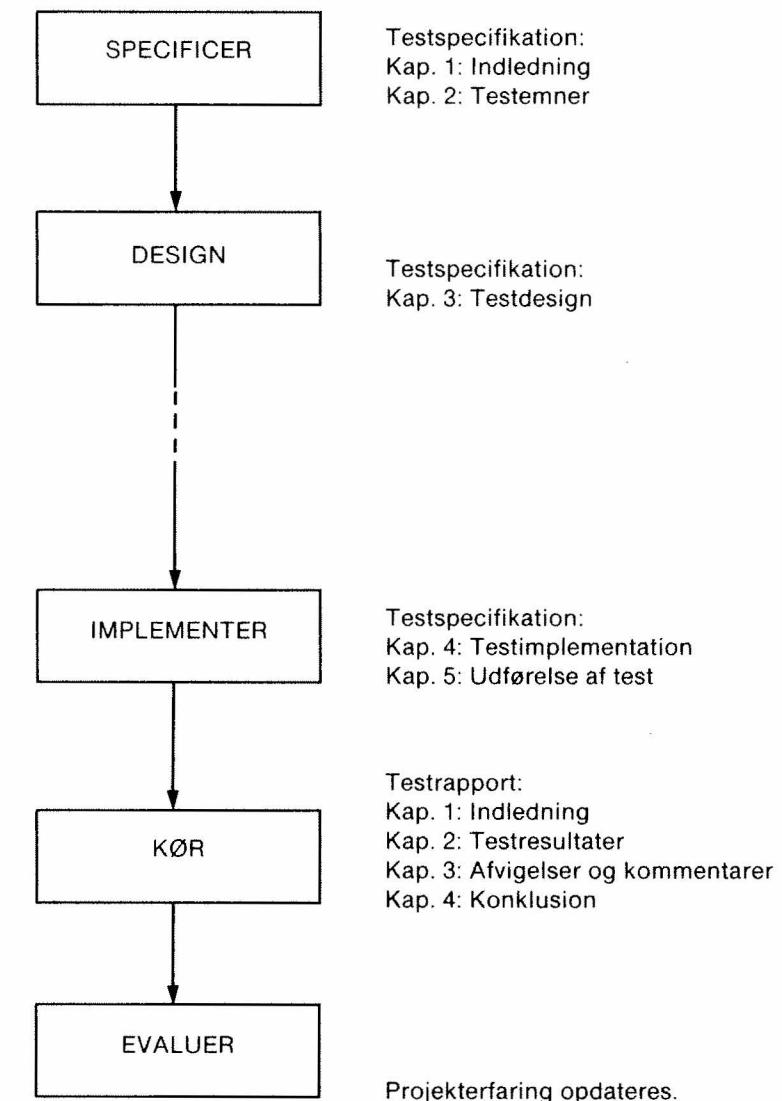
dette formål findes i *Vejledning i Konfigurationsstyring*. En kopi af problem/ændringsrapporten indsættes i dette afsnit.

4. Konklusion

Hvordan gik testen? Er den testede software i orden, eller skal alt laves om? Kan man eventuelt frigive en midlertidig version?

6.4 Testdokumentationens udvikling gennem faserne

Som beskrevet i vejledningen udføres testaktiviteter løbende gennem udviklingsprojektet, og testdokumentationen udarbejdes tilsvarende. På følgende figur er vist sammenhængen mellem testaktiviteterne (beskrevet i kapitel 4) og testdokumentationen (beskrevet ovenfor i kapitel 5). For alle integrationstests og modultests indgår testdokumentationen i programdokumentationen, mens accepttestdokumentationen er selvstændige dokumenter på lige fod med kravspecifikationen.



Figur 15. Tidsmæssig sammenhæng mellem testaktiviteter og udarbejdelsen af testdokumentationen.

Sandford Friedenthal, Alan Moore, Rich Steiner

A Practical Guide to SysML

Morgan Kaufmann, ISBN 978-0-123-74379-4.

Chapter 3 SysML Language Overview, pp 29 – 60

SysML Language Overview

3

This chapter provides an overview of SysML that includes a simple example showing how the language is applied to the system design of an automobile that was introduced in Chapter 1. The example includes references to chapters in Part II that provide a detailed description of the diagrams and language concepts. Part III includes two more detailed examples of how MBSE methods can be used with SysML to specify and design a system.

3.1 SysML Purpose and Key Features

SysML is a general-purpose graphical modeling language that supports the analysis, specification, design, verification, and validation of complex systems. These systems may include hardware, software, data, personnel, procedures, facilities, and other elements of man-made and natural systems. The language is intended to help specify and architect systems and specify its components that can then be designed using other domain-specific languages such as UML for software design and VHDL for hardware design.

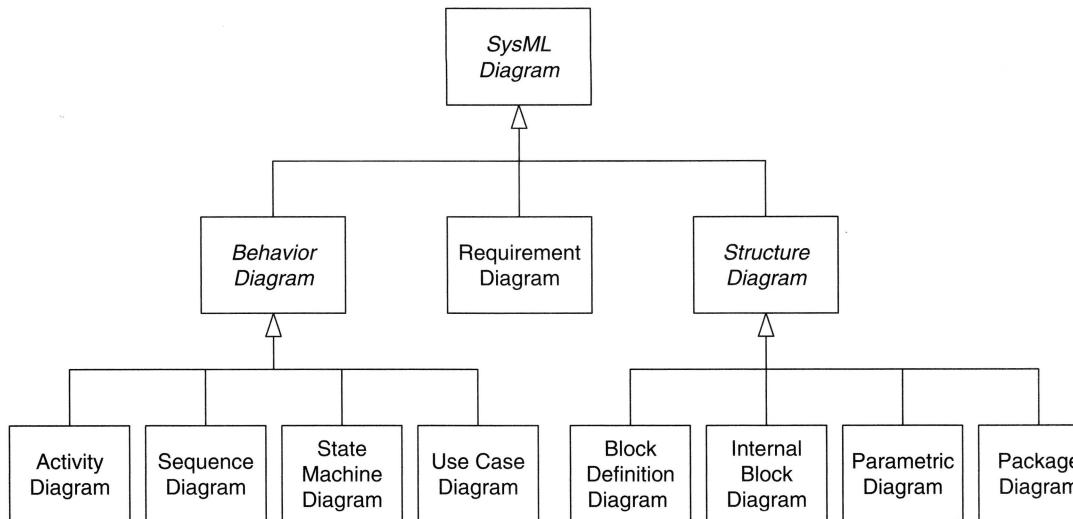
SysML can represent systems, components, and other entities as follows:

- Structural composition, interconnection, and classification
- Function-based, message-based, and state-based behavior
- Constraints on the physical and performance properties
- Allocations between behavior, structure, and constraints (e.g., functions allocated to components)
- Requirements and their relationship to other requirements, design elements, and test cases

3.2 SysML Diagram Overview

SysML includes nine diagrams as shown in the diagram taxonomy in Figure 3.1. Each diagram type is summarized here, along with its relationship to UML diagrams:

- *Requirement diagram* represents text-based requirements and their relationship with other requirements, design elements, and test cases to support requirements traceability (not in UML)

**FIGURE 3.1**

SysML diagram taxonomy.

- *Activity diagram* represents behavior in terms of the ordering of actions based on the availability of inputs, outputs, and control, and how the actions transform the inputs to outputs (modification of UML activity diagram)
- *Sequence diagram* represents behavior in terms of a sequence of messages exchanged between parts (same as UML sequence diagram)
- *State machine diagram* represents behavior of an entity in terms of its transitions between states triggered by events (same as UML state machine diagram)
- *Use case diagram* represents functionality in terms of how a system or other entity is used by external entities (i.e., actors) to accomplish a set of goals (same as UML use case diagram)
- *Block definition diagram* represents structural elements called blocks, and their composition and classification (modification of UML class diagram)
- *Internal block diagram* represents interconnection and interfaces between the parts of a block (modification of UML composite structure diagram)
- *Parametric diagram* represents constraints on property values, such as $F = m*a$, used to support engineering analysis (not in UML)
- *Package diagram* represents the organization of a model in terms of packages that contain model elements (same as UML package diagram)

A diagram graphically represents a particular aspect of the system model as described in Section 2.1.2. The diagram type constrains the type of model elements and associated symbols that can appear on a diagram. For example, an activity

diagram can include diagram elements that represent actions, control flow, and input/output flow, but not diagram elements for connectors and ports. As a result, a diagram represents a subset of the underlying model repository, as described in Chapter 2. Tabular representations are also supported in SysML as a complement to diagram representations to capture model information such as allocation tables.

3.3 Using SysML in Support of MBSE

SysML provides a means to capture the system modeling information as part of an MBSE approach without imposing a specific method on how this is performed. The selected method determines which activities are performed, the ordering of the activities, and which modeling artifacts are created to represent the system. For example, traditional structured analysis methods can be used to decompose the functions and allocate the functions to components. Alternatively, one can apply a use case driven approach that derives functionality based on scenario analysis and associated interactions among parts. The two methods may produce different combinations of diagrams in different ways to represent the system specification and design.

A typical use of the language may include one or more iterations of the following activities to specify and design the system:

- Capture and analyze black box system requirements
 - Capture text-based requirements in a requirements management tool
 - Import requirements into the SysML modeling tool
 - Identify top-level functionality in terms of system use cases
 - Capture the traceability between the use cases and requirements
 - Model the use case scenarios as activity diagrams, sequence diagrams, and/or state machine diagrams
 - Create the system context diagram
 - Identify system test cases to support system verification
- Develop one or more candidate system architectures to satisfy the requirements
 - Decompose the system using the block definition diagram
 - Define the interaction among the parts using activity or sequence diagrams
 - Define the interconnection among the parts using the internal block diagram
- Perform engineering and trade-off analysis to evaluate and select the preferred architecture
 - Capture the constraints on system properties using the parametric diagram to support analysis of performance, reliability, cost, and other critical properties
 - Perform the engineering analysis to determine the budgeted values of the system properties (typically done in separate engineering analysis tools)

- Specify component requirements and their traceability to system requirements
 - Capture the functional, interface, and performance requirements for each component (block) in the architecture
 - Trace component requirements to the system requirements
- Verify that the system design satisfies the requirements by executing system-level test cases

Other systems engineering activities are performed in conjunction with the preceding modeling activities such as configuration management and risk management. Detailed examples of how SysML can be used to support two different MBSE methods are included in the modeling examples in Part III. A simplified example is described next.

3.4 A Simple Example Using SysML for an Automobile Design

The following example was introduced in Chapter 1 without introducing the model-based approach. It is a simplified example that illustrates how SysML can be applied to specify and design a system.

3.4.1 Example Background and Scope

The example includes at least one diagram for each SysML diagram type, but only highlights selected features of the language. It includes multiple references to the chapters in Part II for the detailed language description. The way the diagrams are used to represent the system and the ordering of the diagrams is intended to be representative of applying a typical model-based approach. However, this will vary depending on the specific process and method used.

3.4.2 Problem Summary

The sample problem describes the use of SysML as it applies to the design of an automobile. A marketing analysis that was done indicated the need to increase the automobile's acceleration and fuel efficiency from its current capability. A variant of the process described in Section 3.3 is used to design the system to satisfy the requirements. In this simplified example, selected aspects of the design are considered to support an initial trade-off analysis. The trade-off analysis included evaluation of alternative vehicle configurations that included a 4-cylinder engine and a 6-cylinder engine to determine whether they can satisfy the acceleration and fuel efficiency requirement.

In addition, the proposed vehicle design includes a vehicle controller and associated software to control the fuel-air mixture and maximize fuel efficiency and engine performance. Only a small subset of the design is addressed to

Table 3.1 Diagrams Used in the Automobile Example

Figure	Diagram Kind	Diagram Name
3.2	Requirement diagram	Automobile System Requirements
3.3	Block definition diagram	Automobile Domain
3.4	Use case diagram	Operate Vehicle
3.5	Sequence diagram	Drive Vehicle
3.6	Sequence diagram	Start Vehicle
3.7	Activity diagram	Control Power
3.8	State machine diagram	Drive Vehicle States
3.9	Internal block diagram	Vehicle Context
3.10	Block definition diagram	Vehicle Hierarchy
3.11	Activity diagram	Provide Power
3.12	Internal block diagram	Power Subsystem
3.13	Block definition diagram	Analysis Context
3.14	Parametric diagram	Vehicle Acceleration Analysis
3.15	Timing diagram (not SysML)	Vehicle Performance Timeline
3.16	Activity diagram	Control Engine Performance
3.17	Block definition diagram	Engine Specification
3.18	Requirement diagram	Max Acceleration Requirement Traceability
3.19	Package diagram	Model Organization

highlight the use of the language. The diagrams used in this example are shown in Table 3.1.

SysML diagrams include a **diagram frame**. The **diagram header** in the diagram frame describes the kind of diagram, the diagram name, and some additional information that provides context for the **diagram content**. Detailed information on diagram frames, diagram headers, and other common diagram elements that apply to all SysML diagrams is described in Chapter 4.

The example includes the following user-defined notations, called a stereotype, that are added for this example. Chapter 14 describes how stereotypes are used to further customize the language for domain-specific applications.

- «hardware»
- «software»
- «store»
- «system of interest»

3.4.3 Capturing the *Automobile Specification* in a Requirement Diagram

The **requirement diagram** for the *Automobile System Requirements* is shown in Figure 3.2. The kind of diagram (e.g., req) and the diagram name are shown in the diagram header at the upper left. The diagram depicts the requirements that are typically captured in a text specification. The requirements are shown in a containment hierarchy to depict the hierarchical relationship among them. The *Automobile Specification* is the top-level requirement that contains the lower-level requirements. The line with the crosshairs symbol at the top is the **containment** relationship.

The specification contains requirements for *Passenger and Baggage Load*, *Vehicle Performance*, *Riding Comfort*, *Emissions*, *Fuel Efficiency*, *Production Cost*, *Vehicle Reliability*, and *Occupant Safety*. The Vehicle Performance requirement contains requirements for *Maximum Acceleration*, *Top Speed*, *Braking Distance*, and *Turning Radius*. Each requirement includes a unique identification, its text, and can include other user-defined properties, such as verification status and risk, that are typically associated with requirements. The text for the *Maximum Acceleration* requirement is “the vehicle shall accelerate from 0 to 60 mph in less than 8 seconds” and the text for the *Fuel Efficiency* requirement is “the vehicle shall achieve at least 25 miles per gallon under the stated driving conditions.”

The requirements may have been created in the modeling tool, or alternatively, they may have been imported from a requirements management tool or a text document. The requirements can be related to other requirements, design elements, and test cases using **derive**, **satisfy**, **verify**, **refine**, **trace**, and **copy** relationships. These relationships can be used to establish requirements traceability with a high degree of granularity. Some of these relationships are highlighted in Section 3.4.19. Chapter 12 provides a detailed description of how requirements are modeled in SysML. Requirements can be represented using multiple display options to view the requirements, their properties, and their relationships, which includes a tabular representation.

3.4.4 Defining the *Vehicle* and Its External Environment Using a Block Definition Diagram

In system design, it is important to identify what is external to the system that may either directly or indirectly interact with it. The **block definition diagram** for the *Automobile Domain* in Figure 3.3 defines the *Vehicle* and the external systems, users, and other entities that the vehicle may directly or indirectly interact with.

A **block** is a very general modeling concept in SysML that is used to model a wide variety of entities that have structure such as systems, hardware, software, physical objects, and abstract entities. That is, a block can represent any real or abstract entity that can be conceptualized as a structural unit with one or more distinguishing features. The block definition diagram captures the relation between blocks such as a block hierarchy.

The *Automobile Domain* is the top-level block in the block definition diagram in Figure 3.3. It is **composed** of other blocks as indicated by the black diamond

req Requirements [Automobile System Requirements]

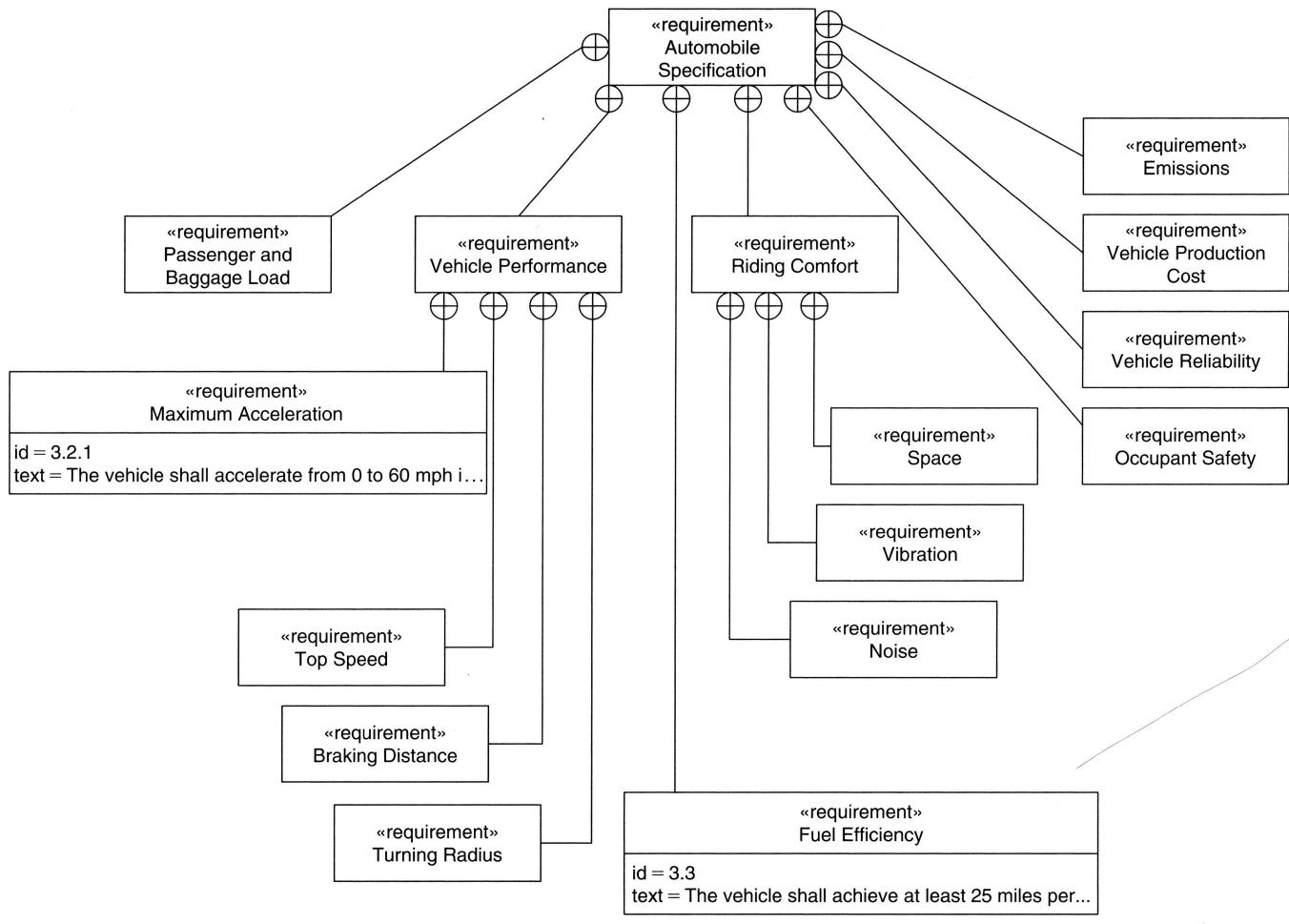
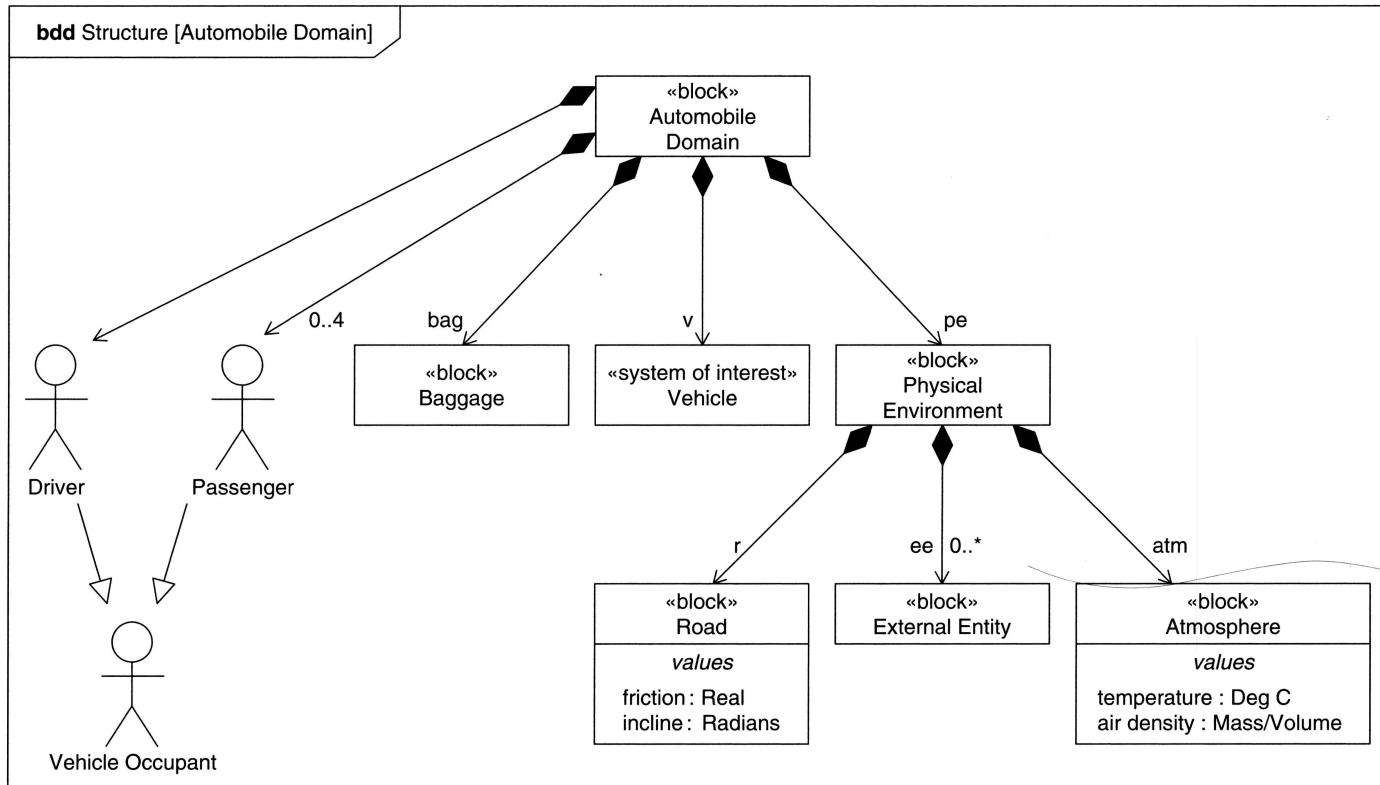


FIGURE 3.2

Requirement diagram showing the system requirements contained in the *Automobile Specification*.

**FIGURE 3.3**

Block definition diagram of the *Automobile Domain* showing the *Vehicle* and its external users and physical environment.

symbol and line with the arrowhead pointing to the blocks that compose it. The differences between the composition hierarchy (i.e., black diamond) and the containment hierarchy (i.e., crosshairs symbol) shown in Figure 3.2 are explained in Part II. The name next to the arrow identifies a particular usage of a block as described later in this section. The *Vehicle* block is referred to as the «system of interest» using the bracket symbol called **guillemet**. The other blocks are external to the vehicle. These include the *Driver*, *Passenger*, *Baggage*, and *Physical Environment*. The *Driver* and *Passenger* are shown using stick-figure symbols. Notice that even though the *Driver*, *Passenger*, and *Baggage* are assumed to be physically inside the *Vehicle*, they are not part of the *Vehicle* structure, and therefore are external to it.

The *Driver* and *Passenger* are **subclasses** of *Vehicle Occupant* as indicated by the hollow triangle symbol. This means that they are kinds of vehicle occupants that inherit common features from *Vehicle Occupant*. In this way, a classification can be created by specializing blocks from more generalized blocks.

The *Physical Environment* is composed of the *Road*, *Atmosphere*, and multiple *External Entities*. The *External Entity* can represent any physical object, such as a traffic light or another vehicle, that the *Driver* interacts with. The interaction between the *Driver* and an *External Entity* can impact how the *Driver* interacts with the *Vehicle*, such as when the traffic light changes color. The **multiplicity** symbol $0..*$ represents an undetermined maximum number of external entities. The multiplicity symbol can also represent a single number or a range, such as the multiplicity of $0..4$, for the number of *Passengers*.

Each block defines a structural unit, such as a system, hardware, software, data element, or other conceptual entity, as described earlier. A block can have a set of **features** that further define it. The features of the block define its **properties** (e.g., weight), its **behavior** in terms of activities **allocated** to the block or **operations** of the block, and its interfaces as defined by its **ports**. Together, these features enable a modeler to specify each block at the level of detail that is appropriate for the application.

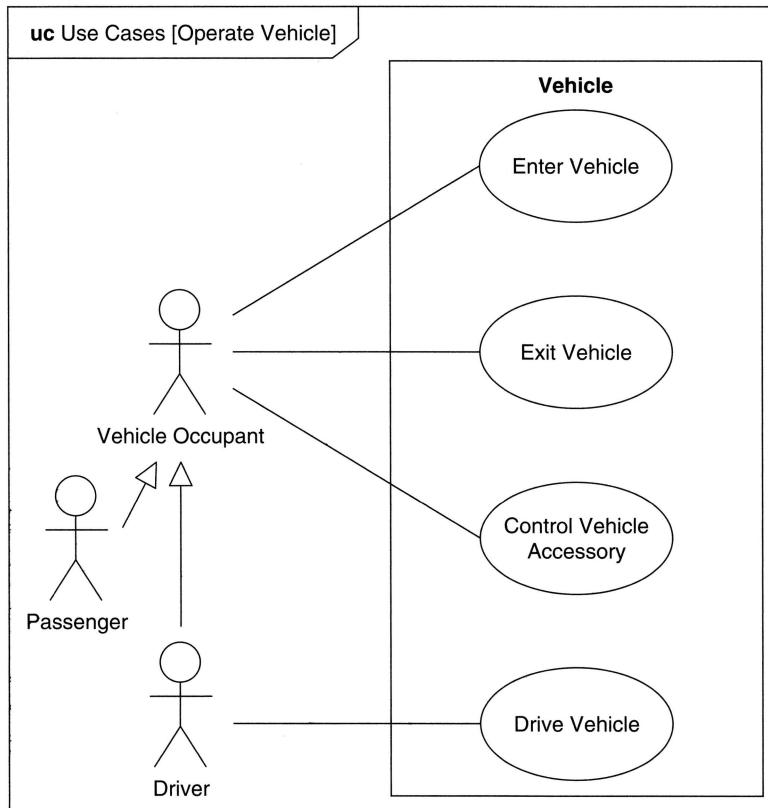
The *Road* is a block that has a property called *incline* with units of *Radians* and a property called *friction* that is defined as a real number. Similarly, *Atmosphere* is a block that has two properties for *temperature* and *air density*. These properties are used along with other properties to support analysis of vehicle acceleration and fuel efficiency, which are discussed in Sections 3.4.12 and 3.4.13.

The block definition diagram specifies the blocks and their interrelationships. It is often used in systems modeling to depict multiple levels of the system hierarchy from the top-level domain block (e.g., *Automobile Domain*) down to vehicle components. Chapter 6 provides a detailed description of how blocks are modeled in SysML, including their features and relationships.

3.4.5 Use Case Diagram for *Operate Vehicle*



The **use case diagram** for *Operate Vehicle* in Figure 3.4 depicts the major functionality for operating the vehicle. The **use cases** include *Enter Vehicle*, *Exit Vehicle*, *Control Vehicle Accessory*, and *Drive Vehicle*. The *Vehicle* is the **subject**

**FIGURE 3.4**

Use case diagram describes the major functionality in terms of how the *Vehicle* is used by the actors to *Operate Vehicle*. The actors are defined on the block definition diagram in Figure 3.3.

of the use cases and is represented by the rectangle. The *Vehicle Occupant* is an **actor** who is external to the vehicle and is represented as the stick figure. The subject and actors correspond to the blocks in Figure 3.3. In a use case diagram, the subject (e.g., *Vehicle*) is used by the actors (e.g., *Vehicle Occupant*) to achieve the goals defined by the use cases (e.g., *Drive Vehicle*).

The *Passenger* and *Driver* are both kinds of vehicle occupants as described in the previous section. All vehicle occupants participate in entering and exiting the vehicle and controlling vehicle accessories, but only the driver participates in *Drive Vehicle*. There are several other relationships between use cases that are not shown here. Chapter 11 provides a detailed description of how use cases are modeled in SysML.

Use cases define how the system is used to achieve a user goal. Other use cases can represent how the system is used across its life cycle, such as when

manufacturing, operating, and maintaining the vehicle. The primary emphasis for this example is on the *Drive Vehicle* use case to address the acceleration and fuel efficiency requirements.

The requirements are often related to use cases since use cases represent the high-level functionality or goals for the system. Sometimes, use case textual descriptions are defined to accompany the use case definition. One approach to relate requirements to use cases is to capture the use case descriptions as SysML requirements and relate them to the use case using a refine relationship.

The use cases describe the high-level goals of the system as described previously. The goals are accomplished by the interactions between the actors (e.g., *Driver*) and the subject (e.g., *Vehicle*). These interactions are realized through more detailed descriptions of behavior as described in the next section.

3.4.6 Representing *Drive Vehicle* Behavior with a Sequence Diagram

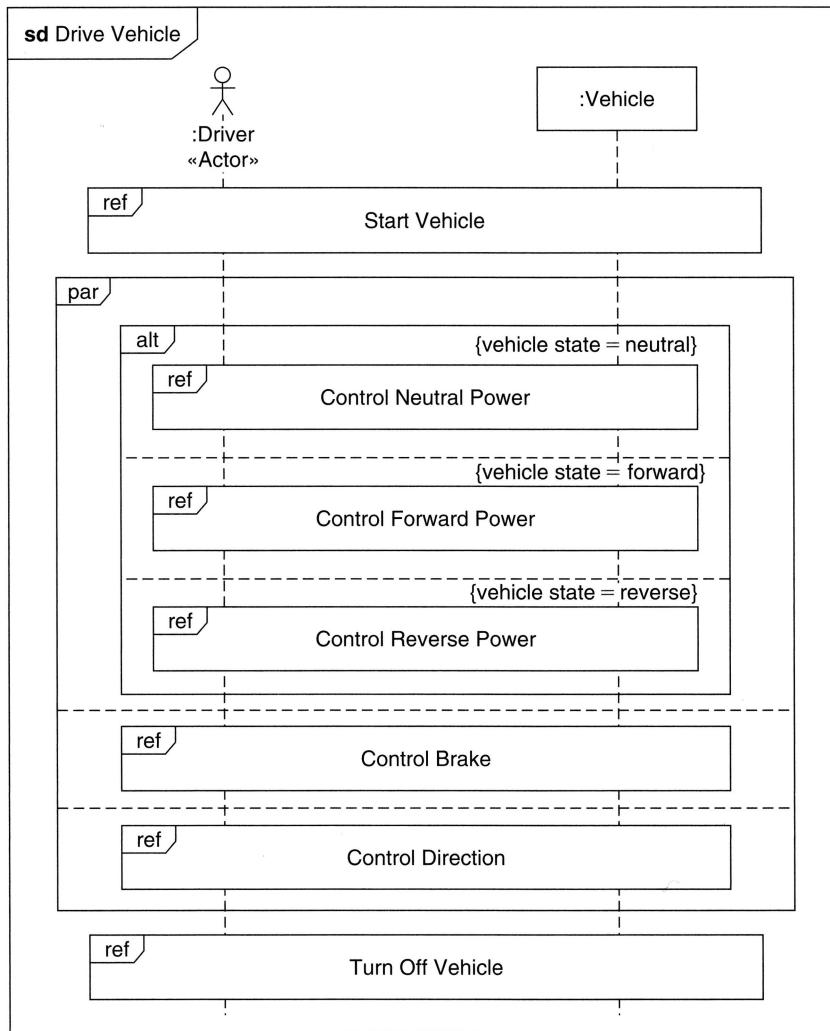
The behavior for the *Drive Vehicle* use case in Figure 3.4 is represented by the **sequence diagram** in Figure 3.5. The sequence diagram specifies the **interaction** between the *Driver* and the *Vehicle* as indicated by the names at the top of the **lifelines**. Time proceeds vertically down the diagram. The first interaction is *Start Vehicle*. This is followed by the *Driver* and *Vehicle* interactions to *Control Power*, *Control Brake*, and *Control Direction*. These three interactions occur in parallel as indicated by **par**. The **alt** on the *Control Power* interaction stands for alternative, and indicates that the *Control Neutral Power*, *Control Forward Power*, or *Control Reverse Power* interaction occurs as a condition of the *vehicle state* shown in brackets. The state machine diagram in Section 3.4.9 specifies the *vehicle state*. The *Turn-off Vehicle* interaction occurs following these interactions.

The **interaction occurrences** in the figure each reference a more detailed interaction as indicated by **ref**. The referenced interaction for *Start Vehicle* is another sequence diagram that is illustrated in Section 3.4.7. The remaining interaction occurrences are references allocated to activity diagrams as described in Section 3.4.8.

3.4.7 Referenced Sequence Diagram to *Start Vehicle*

The *Start Vehicle* sequence diagram in Figure 3.6 is an interaction that is referenced in the sequence diagram in Figure 3.5. As stated previously, time proceeds vertically down the diagram. In this example, the more detailed interaction shows the driver sending a **message** requesting the vehicle to start. The vehicle responds with the **vehicle on reply message** shown as a dashed line. Once the reply has been received, the driver and vehicle can proceed to the next interaction.

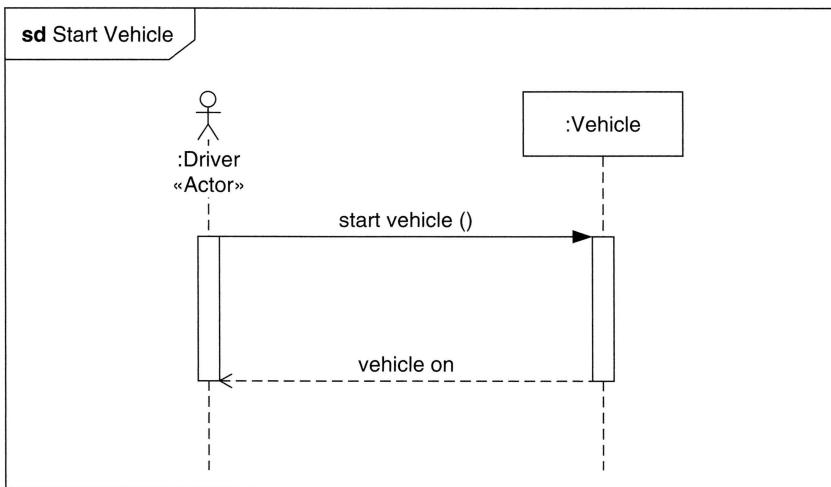
The sequence diagram can include multiple types of messages. In this example, the message is **synchronous** as indicated by the filled arrowhead on the message. The messages can also be **asynchronous** represented by an open arrowhead, where the sender does not wait for a reply. The synchronous messages represent

**FIGURE 3.5**

Drive Vehicle sequence diagram describes the interactions between the *Driver* and the *Vehicle* to realize the *Drive Vehicle* use case in Figure 3.4.

an operation call that specifies a request for service. The arguments of the operation call represent the input data and return.

Sequence diagrams can include multiple message exchanges between multiple lifelines that represent interacting entities. The sequence diagram also provides considerable additional capability to express behavior that includes other message types, timing constraints, additional control logic, and the ability to decompose the behavior of a lifeline into the interaction of its parts. Chapter 9 provides a detailed description of how interactions are modeled with sequence diagrams.

**FIGURE 3.6**

Sequence diagram for the *Start Vehicle* interaction that was referenced in the *Drive Vehicle* sequence diagram, showing the message from *Driver* requesting *Vehicle* to start and the *vehicle on* reply from the *Vehicle*.

3.4.8 Control Power Activity Diagram

The sequence diagram is effective for communicating discrete types of behavior as indicated with the *Start Vehicle* sequence diagram in Figure 3.6. However, continuous types of behaviors associated with the interactions to *Control Power*, *Control Brake*, and *Control Direction* can sometimes be more effectively represented with activity diagrams.

The *Drive Vehicle* sequence diagram in Figure 3.5 includes the control power interactions that we would like to represent with an activity diagram instead of a sequence diagram. To accomplish this, the *Control Neutral Power*, *Control Forward Power*, and *Control Reverse Power* interactions in Figure 3.5 are **allocated** to a corresponding *Control Power* activity diagram using the SysML allocation relationship (not shown).

The **activity diagram** in Figure 3.7 shows the **actions** required of the *Driver* and the *Vehicle* to *Control Power*. The **activity partitions** (or **swimlanes**) represent the *Driver* and the *Vehicle*. The actions in the activity partitions specify functional requirements that the *Driver* and *Vehicle* must perform.

When the activity is initiated, it starts execution at the **Initial Node** and transitions to the *Control Accelerator Position* action that is performed by the *Driver*. The output of this action is the *Accelerator Cmd*, which is a continuous input to the *Provide Power* action that the *Vehicle* must perform. The output of the *Provide Power* action is the *Torque* generated by the wheels to the road to produce the force that accelerates the *Vehicle*. When the *ignition off* signal is received by the *Vehicle*, the activity terminates at the **Activity Final**. Based

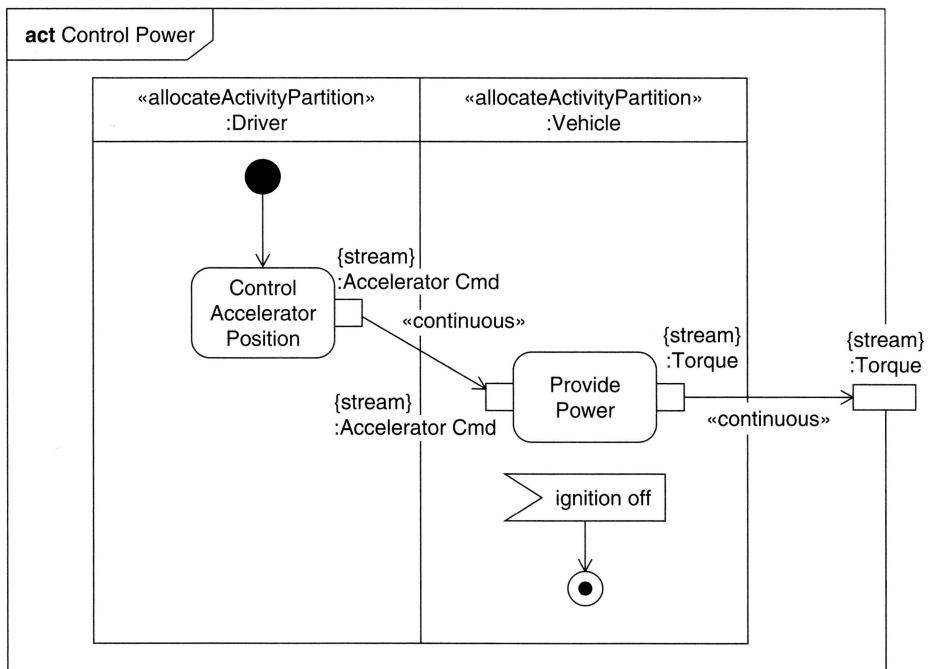


FIGURE 3.7

Activity diagram allocated from the *Control Power* interaction that was referenced in the *Drive Vehicle* sequence diagram in Figure 3.5. It shows the continuous *Accelerator Cmd* input from the *Driver* to the *provide power* action that the *Vehicle* must perform.

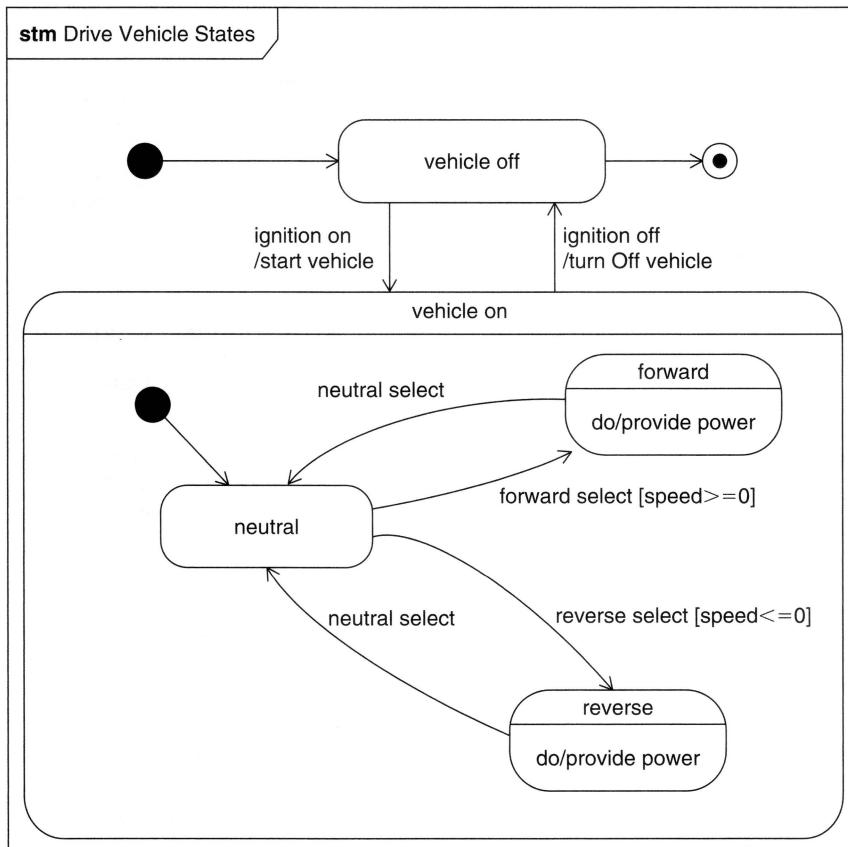
on this scenario, the *Driver* is required to *Control Accelerator Position* and the *Vehicle* is required to *Provide Power*.

Activity diagrams include semantics for precisely specifying the behavior in terms of the flow of control and inputs and outputs. Chapter 8 provides a detailed description of how activities are modeled.

3.4.9 State Machine Diagram for *Drive Vehicle States*

The **state machine diagram** for the *Drive Vehicle States* is shown in Figure 3.8. This diagram shows the states of the *Vehicle* and the **events** that can **trigger** a transition between the **states**.

When the *Vehicle* is ready to be driven, it starts in the *vehicle off* state. The *ignition on* event triggers a transition to the *vehicle on* state. The text on the transition indicates that the *Start Vehicle* behavior is executed prior to entering the *vehicle on* state. After entry to the *vehicle on* state, the *Vehicle* immediately transitions to the *neutral* state. A *forward select* event triggers a transition to the *forward* state if the **guard condition** *[speed >= 0]* is true. While in the *forward* state, the *Vehicle* performs the *Provide Power* behavior that was referred to in the activity diagram in Figure 3.7. The *neutral select* event triggers the transition from

**FIGURE 3.8**

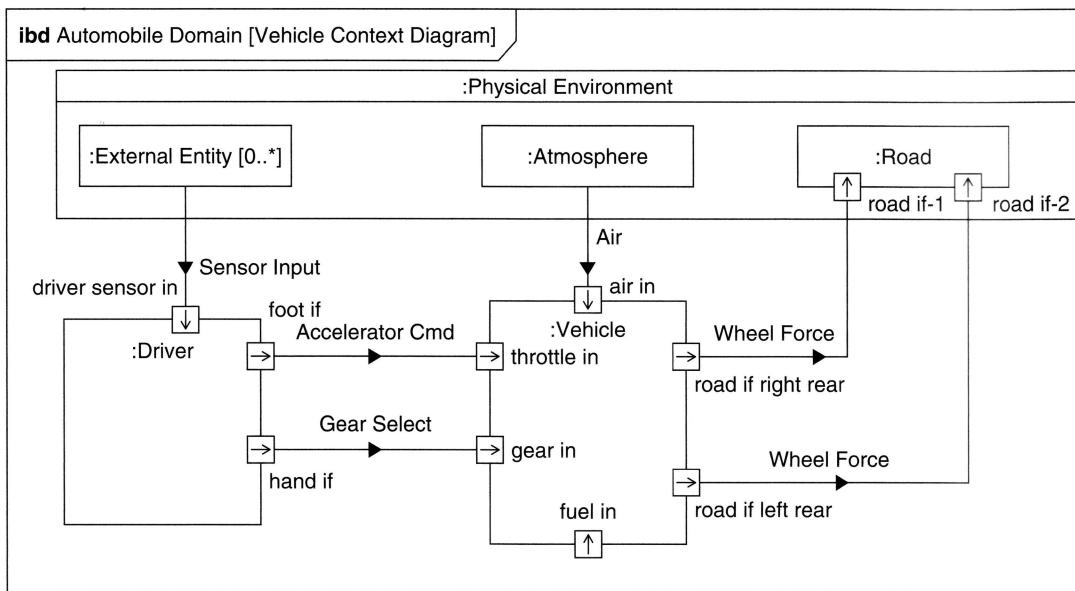
State machine diagram that shows the *Drive Vehicle States* and the transitions between them.

the *forward* state back to the *neutral* state. The state machine diagram shows the additional transitions between the *neutral* and *reverse* states. An *ignition off* event triggers the transition back to the *vehicle off* state. The *Vehicle* can reenter the *vehicle on* state when an *ignition on* event occurs.

A state machine can specify the life-cycle behavior of a block in terms of its states and transitions, and are often used with sequence and activity diagrams, as shown in this example. State machines have many other features including **orthogonal regions** and additional transition semantics that are described in Chapter 10.

3.4.10 Vehicle Context Using an Internal Block Diagram

The *Vehicle Context Diagram* is shown in Figure 3.9. The diagram shows the interfaces between the *Vehicle*, the *Driver*, and the *Physical Environment* (i.e., *Road*, *Atmosphere*, and *External Entity*) that were defined in the block definition diagram in Figure 3.3. The *Vehicle* has interfaces with the *Driver*, the *Atmosphere*,

**FIGURE 3.9**

Internal block diagram for the *Vehicle Context* shows the *Vehicle* and its external interfaces with the *Driver* and *Physical Environment* that were defined in Figure 3.3.

and the *Road*. The *Driver* has interfaces with the *External Entities* such as a traffic light or another vehicle, via the driver sensor inputs (e.g., seeing, hearing). However, the *Vehicle* does not directly interface with the *External Entities*. The multiplicity on the *External Entity* is consistent with the multiplicity shown in the block definition diagram in Figure 3.3.

This context diagram is an **internal block diagram** that shows how the **parts** of the *Automobile Domain* block from Figure 3.3 are connected. It is called an internal block diagram because it represents the internal structure of a higher-level block, which in this case is the *Automobile Domain* block. The *Vehicle ports* specify interaction points with other parts and are represented as the small squares on the boundary of the parts. **Connectors** define how the parts connect to one another via their ports and are represented as the lines between the ports. Parts can also be connected without ports as indicated by some of the interfaces in the figure when the details of the interface are not of interest to the modeler.

In Figure 3.9, only the external interfaces needed for the *Vehicle* to provide power are shown. For example, the interfaces between the rear tires and the road are shown. It is assumed to be a rear wheel-drive vehicle where power can be distributed differently to the rear wheels depending on tire-to-road traction and other factors. The interface between the front tires and the road is not shown in this diagram, but it would be shown when representing the external interfaces for the steering subsystem where the front tires would play a significant role. It is common modeling practice to only represent the aspects of interest on a particular diagram, even though additional information is included in the model.

The black-filled arrowheads on the connector are called **item flows** that represent the items flowing between parts and may include mass, energy, and/or information. In this example, the *Accelerator Cmd* that was previously defined in the activity diagram in Figure 3.8 flows from the *Driver* port to the *throttle in* port of the *Vehicle*, and the *Gear Select* flows from another *Driver* port to the *gear in* port on the *Vehicle*. The inputs and outputs from the activity diagram are **allocated** to the item flows on the connectors. Allocations are discussed as a general-purpose relationship for mapping one model element to another in Chapter 13.

In SysML, there are two different kinds of ports. The **flow port** specifies the kind of item that can flow in or out of an interaction point, and a **standard port** specifies the services that either are required or provided by the part. The port provides the mechanism to integrate the behavior of the system with its structure.

The internal block diagram enables the modeler to specify both external and internal interfaces of a system or component. An internal block diagram shows how parts are connected, as distinct from a block definition diagram that does not show connectors. Details of how to model internal block diagrams are described in Chapter 6.

3.4.11 *Vehicle Hierarchy Represented on a Block Definition Diagram*

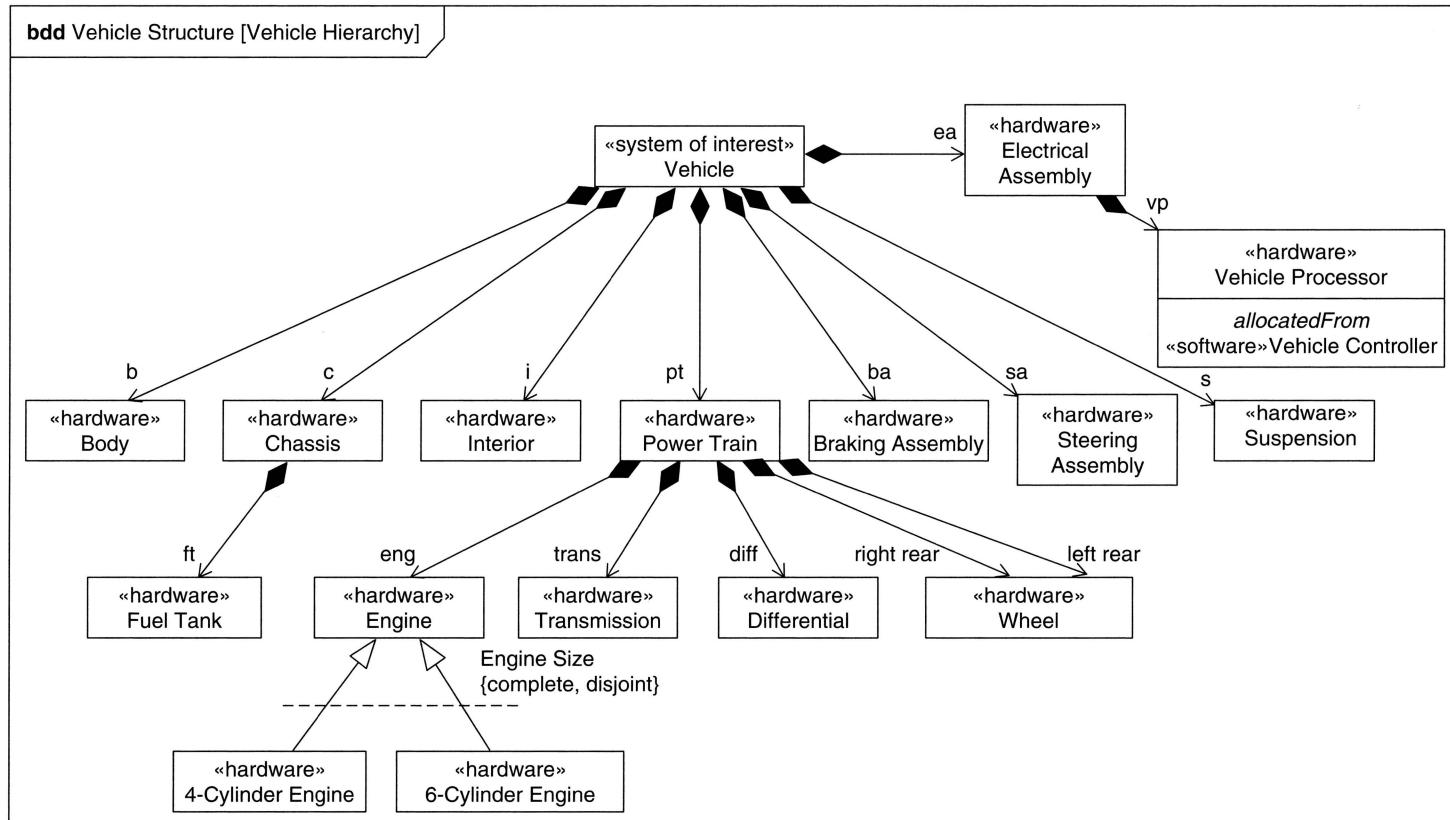
The example to this point has focused on specifying the vehicle in terms of its external interactions and interfaces. The *Vehicle Hierarchy* in Figure 3.10 is a block definition diagram that shows the decomposition of the *Vehicle* into its components. The *Vehicle* is composed of the *Body*, *Chassis*, *Interior*, *Power Train*, and other types of components. Each component type is designated as «hardware».

The *Power Train* is further decomposed into the *Engine*, *Transmission*, *Differential*, and *Wheel*. Note that the *right rear* and *left rear* indicate different usages of a *Wheel* in the context of the *Power Train*. Thus, each rear wheel has a different role and may be subject to different forces, such as is the case when one wheel loses traction. The front wheels are not shown, but could be part of the chassis or part of the steering assembly and would have different roles as well.

The engine may be either 4 or 6 cylinders as indicated by the specialization relationship. The 4- and 6-cylinder vehicle configuration alternatives are being considered to satisfy the acceleration and fuel efficiency requirements. Only one engine type is part of a particular *Vehicle* as indicated by the *{complete, disjoint}* constraint. This implies that the 4- and 6-cylinder engines represent a complete set of subclasses and are mutually exclusive or disjoint.

The *vehicle:Controller* «software» has been allocated to the *Vehicle Processor* as indicated by the allocation compartment. The *Vehicle Processor* is the execution platform for the vehicle control software. The software is being enhanced to control many of the automobile engine and transmission functions to optimize engine performance and fuel efficiency.

The interaction and interconnection between these components is analyzed in a similar way to what was done at the *Vehicle* black box level, and is used to specify the components of the *Vehicle* system as described in the next sections.

**FIGURE 3.10**

Block definition diagram of the *Vehicle Hierarchy* shows the *Vehicle* and its components. The *Power Train* is further decomposed into its components and the *Vehicle Processor* includes the *Controller* software.

3.4.12 Activity Diagram for *Provide Power*

The activity diagram in Figure 3.7 showed that the vehicle must *provide power* in response to the driver accelerator command and generate wheel–tire torque at the road surface. The *Provide Power* activity diagram in Figure 3.11 shows how the vehicle components generate this torque.

The external inputs to the activity include the *Accelerator Cmd* and *Gear Select* from the *Driver*, and *Air* from the *Atmosphere* to support engine combustion. The outputs are the torque from the right and left rear wheels to the road that provides the force to accelerate the *Vehicle*. Some of the other inputs and outputs, such as exhaust from the engine, are not included for simplicity. The activity partitions represent the vehicle components shown in the block definition diagram in Figure 3.10.

The fuel tank stores and dispenses the fuel to the *Engine*. The accelerator command and air and fuel are input to the *generate torque* action. The engine torque is input to the *amplify torque* action performed by the *Transmission*. The amplified torque is input to the *distribute torque* action performed by the *Differential* that distributes torque to the right and left rear wheels to *provide traction* to the road surface to generate the force to accelerate the *Vehicle*.

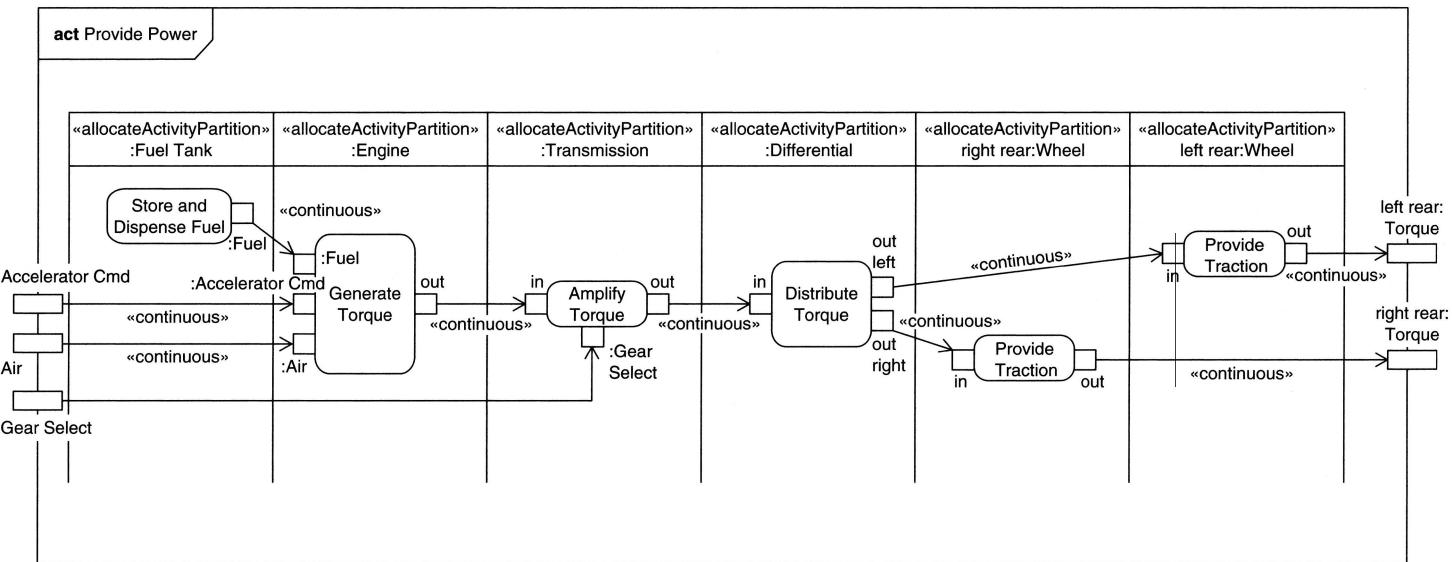
The actions that are allocated to the *Vehicle* components realize the *provide power* action that the *Vehicle* performs, as shown in Figure 3.7. This approach is used to decompose the system behavior.

A few other items are worth noting in this example. The flows are shown to be continuous for all but the *Gear Select*. The inputs and outputs continuously flow in and out of the actions. *Continuous* means that the delta time between arrival of the inputs or outputs approaches zero. Continuous flows build on the concept of streaming inputs and outputs, which means that inputs are accepted and outputs are produced while the action is executing. Conversely, nonstreaming inputs are only available prior to the start of the action execution, and nonstreaming outputs are produced only at the completion of the action execution. The ability to represent streaming and continuous flows adds a significant capability to classic behavioral modeling associated with functional flow diagrams. The continuous flows are assumed to be streaming.

Many other activity diagram features are explained in Chapter 8; they provide a capability to precisely specify behavior in terms of the flow of control and data, and the ability to reuse and decompose behavior.

3.4.13 Internal Block Diagram for the *Power Subsystem*

The previous activity diagram described how the parts of the system interact to *provide power*. The parts of the system are represented by the activity partitions in the activity diagram. The internal block diagram for the vehicle in Figure 3.12 shows how the parts are interconnected to achieve this functionality and is used to specify the interfaces between the parts. This is a structural view of the system versus the behavioral view that was expressed in the activity diagram.

**FIGURE 3.11**

Activity diagram for *Provide Power* shows how the *Vehicle* components generate the torque to move the vehicle. This activity diagram realizes the *provide power* action in Figure 3.7 with activity partitions that correspond to the components in Figure 3.10.

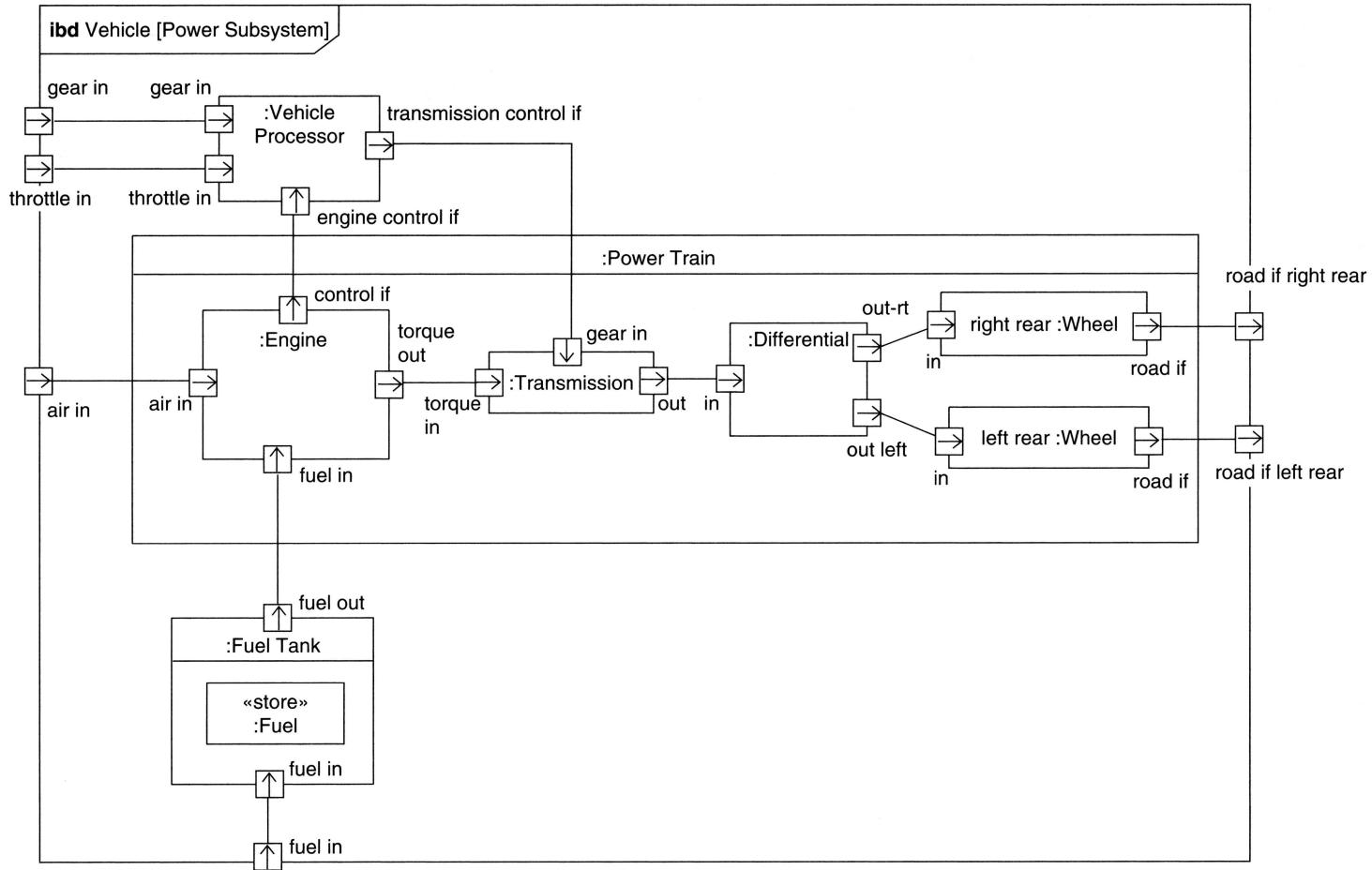


FIGURE 3.12

Internal block diagram for the *Power Subsystem* shows how the parts that *provide power* are interconnected. The parts are represented as the activity partitions in Figure 3.11.

The internal block diagram represents the *Power Subsystem* that only includes the parts of the *Vehicle* that collaborate to *provide power*. The frame of the diagram represents the *Vehicle* black box. The ports on the diagram frame correspond to the same ports shown on the *Vehicle* in the *Vehicle Context* diagram in Figure 3.9. This enables the external interfaces to be preserved as the internal structure of the *Vehicle* is further specified.

The *Engine*, *Transmission*, *Differential*, *right rear* and *left rear Wheel*, *Vehicle Processor*, and *Fuel Tank* are interconnected via their ports. The *Fuel* is stored in the *Fuel Tank* as indicated by «store». The item flows on the connectors are not shown, but represent the items that flow through the system and are allocated from the inputs and outputs on the *Provide Power* activity diagram in Figure 3.11.

Additional subsystems can be created in a similar way to realize specific functionality such as provide braking and provide steering. A composite view of all of the interconnected parts across all subsystems can also be created in a composite *Vehicle* internal block diagram. An example of this is included in the residential security example in Chapter 16.

It is appropriate to elaborate on the usage concept that was first introduced in Section 3.4.11 when discussing the *right rear* and *left rear* wheels. A part in an internal block diagram represents a particular usage of a block. The block represents the generic definition, whereas the part represents a usage of a block definition in a particular context. Thus, the right rear and left rear are different usages of the *Wheel* block in the context of the *Vehicle*. A usage (or part) is the same as a role. The parts in the diagram are indicated by the colon (:) notation. A part enables the same block to be reused in many different contexts and be uniquely identified by its usage. Each part may have unique behaviors, properties, and constraints that apply to its particular usage.

The concept of definition and usage is applied to many other SysML language constructs as well. One example is that the item flows themselves can have a definition and usage. For example, an item flow entering the fuel tank can be in :Fuel and the item flow exiting the fuel tank can be out :Fuel. Both flows are defined by fuel, but “in” and “out” represent different usages of Fuel in the *Vehicle* context.

As mentioned previously, Chapter 6 provides the detailed language description for both block definition diagrams and internal block diagrams, and includes the concept of role, references, and many other key concepts for modeling blocks and parts.

3.4.14 Defining the Equations to Analyze Vehicle Performance

Critical requirements for the design of this automobile are to accelerate from 0 to 60 mph in less than 8 seconds, while achieving a fuel efficiency of greater than 25 miles per gallon. These two requirements impose conflicting requirements on the design space, such that increasing the acceleration capability can result in a design with lower fuel efficiency. Two alternative configurations, including a 4- and 6-cylinder engine, are evaluated to determine which configuration is the preferred solution to meet the acceleration and fuel efficiency requirements.

The 4- and 6-cylinder engine alternatives are shown in the *Vehicle Hierarchy* in Figure 3.10. There are other design impacts that may result from the automobile configurations with different engines, such as the vehicle weight, body shape, and electrical power. This simplified example only considers the impact on the *Power Subsystem*. The vehicle controller is assumed to control the fuel and air mixture, and control when the gear changes the automatic transmission to optimize engine and overall performance.

The block definition diagram in Figure 3.13 introduces a new type of block called a **constraint block**. Instead of defining systems and components, the constraint block defines constraints in terms of equations and their **parameters**.

In this example, the *Analysis Context* block is composed of a series of constraint blocks to analyze the vehicle acceleration to determine whether either the 4- or 6-cylinder vehicle configuration can satisfy its requirement. The constraint blocks define generic equations for *Gravitational Force*, *Drag Force*, *Power Train*

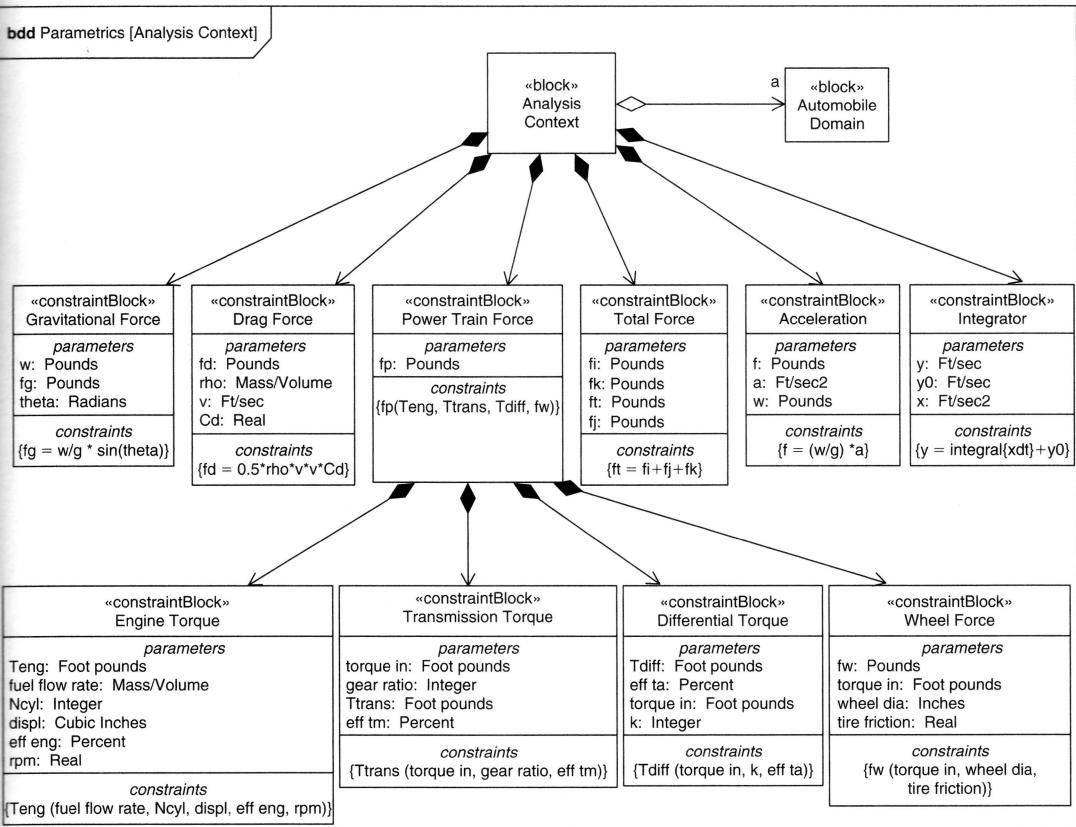


FIGURE 3.13

Block definition diagram for the *Analysis Context* that defined the equations for analyzing the vehicle acceleration requirement. The equations and their parameters are specified using constraint blocks. The *Automobile Domain* block from Figure 3.3 is referenced since it is the subject of the analysis.

Force, Total Force, Acceleration, and an Integrator. The *Total Force* equation, as an example, shows that ft is the sum of fi , ff , and fk . Note that the parameters are defined along with their units and/or dimensions in the constraint block.

The *Power Train Force* is further decomposed into other constraint blocks that represent the equations for torque from the *Engine*, *Transmission*, *Differential*, and *Wheels*. The equations are not explicitly defined, but the critical parameters of the equations are. This is important since it may be of value to identify the critical parameters, and to defer definition of the equations until the detailed analysis is performed.

The *Analysis Context* block also references the *Automobile Domain* block that was originally shown in the block definition diagram in Figure 3.3. The intent of this diagram is to identify both the equations for the analysis and the subject of the analysis. Referencing the *Automobile Domain* enables the equations to constrain the properties of the *Vehicle*, its components, and the physical environment. The parameters of the generic equations are bound to the properties of the system and the environment that is being analyzed, as described in the next section.

3.4.15 Analyzing *Vehicle Acceleration* Using the Parametric Diagram

The previous block definition diagram defined the equations and associated parameters needed to analyze the system. The **parametric diagram** in Figure 3.14 shows how these equations are used to analyze the vehicle acceleration to determine the time for the *Vehicle* to accelerate from 0 to 60 mph.

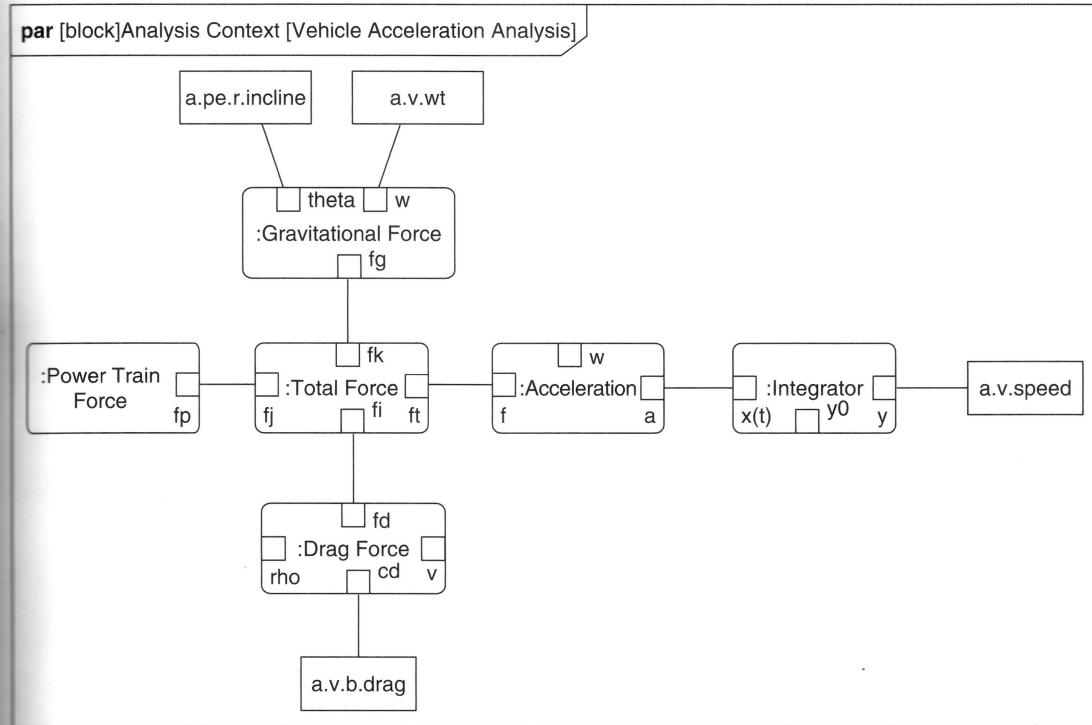
The parametric diagram shows a network of constraints (equations). Each constraint is a usage of a constraint block defined in the block definition diagram in Figure 3.13. The parameters of the equation are shown as small rectangles flush with the inside boundary of the constraint.

A parameter in one equation can be bound to a parameter in another equation by a **binding connector**. An example of this is the parameter ft in the *Total Force* equation that is bound to the parameter f in the *Acceleration* equation. This means that ft in the *Total Force* equation is equal to f in the *Acceleration* equation.

The parameters can also be bound to **properties** of blocks to make the parameter equal to the property. The properties of blocks are shown as the rectangles in the diagram. An example is the binding of the coefficient of drag parameter cd in the *Drag Force* equation to the drag property called *drag*, which is a property of the vehicle *Body*. The dot notation “*a.v.b.*” that precedes the drag property specifies that this is a property of the body, which is part of the vehicle that is part of the *Automobile Domain*. Another example is the binding of the road *incline* angle to the angle *theta* in the gravity force equation. This binding enables parameters of generic equations to be set equal to specific properties of the blocks. In this way, generic equations can be used to analyze many different designs.

The parametric diagram and related modeling information can be provided to the appropriate simulation and/or analysis tools to support execution. This engineering analysis is used to perform sensitivity analysis and determine which property values are required to satisfy the acceleration requirement.

Other analysis can be performed to determine the required property values for the system components (e.g., *Body*, *Chassis*, *Engine*, *Transmission*, *Differential*,

**FIGURE 3.14**

Parametric diagram that uses the equations defined in Figure 3.13 to analyze *Vehicle Acceleration*. The parameters of the equations are bound to parameters of other constraints and to properties of the *Vehicle* and its environment.

Brakes, Steering Assembly) to satisfy the overall system requirements. In addition to the acceleration and fuel efficiency requirements, other analyses may address requirements for braking distance, vehicle handling, vibration, noise, safety, reliability, production cost, and so on. The parametrics enable the critical properties of the system to be identified and integrated with analysis models. Details of how to model constraint blocks and their usages in parametric diagrams are described in Chapter 7.

3.4.16 Analysis Results from Analyzing *Vehicle Acceleration*

As mentioned in the previous section, the parametric diagram is expected to be executed in an engineering analysis tool to provide the results of the analysis. This may be a separate specialized analysis tool that is not provided by the SysML modeling tool, such as a simple spreadsheet or a high-fidelity performance simulation depending on the need. The analysis results from the execution then provide values that can be incorporated back into the SysML model.

The analysis results from executing the constraints in the parametric diagram are shown in Figure 3.15. This example uses the **UML timing diagram** to display the results. Although the timing diagram is not currently one of the SysML diagram

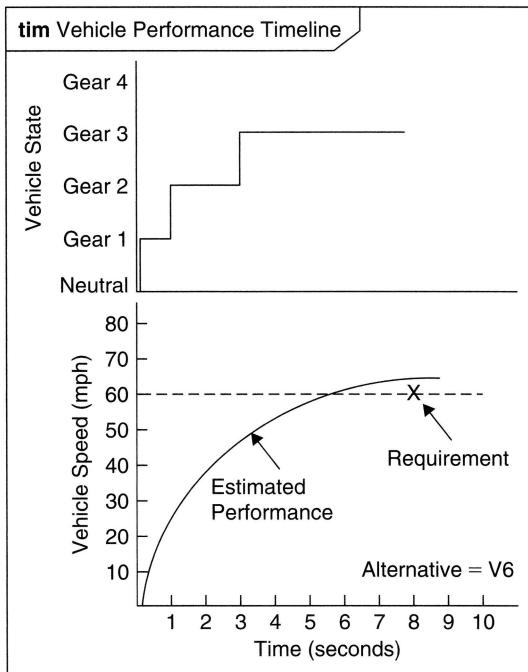


FIGURE 3.15

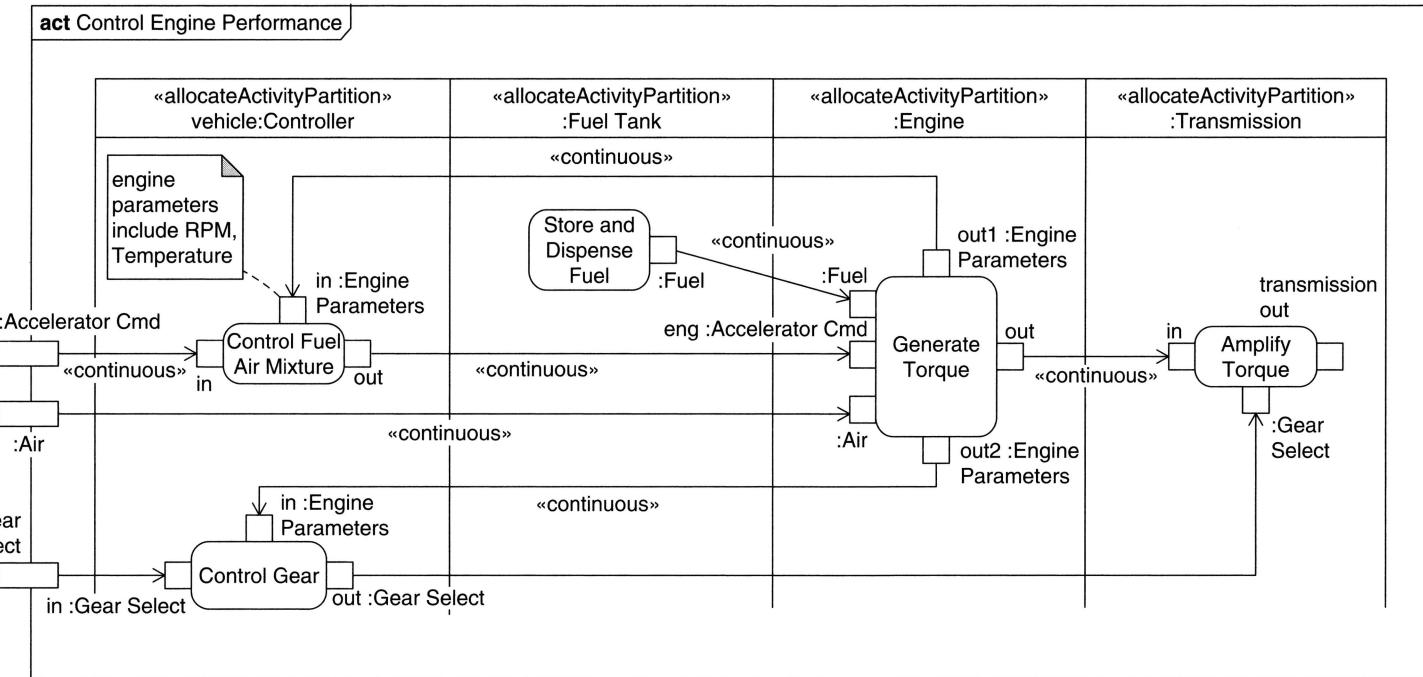
Analysis results from executing the constraints in the parametric diagram in Figure 3.14 showing the vehicle speed property and *Vehicle* state as a function of time. This is captured in a UML timing diagram.

types, it can be used in conjunction with SysML if it is useful for the analysis, along with other more robust visualization methods. The vehicle speed property is shown as a function of time, and the *Vehicle* state is shown as a function of time. The *Vehicle* states correspond to nested states within the *forward* state in Figure 3.8. Based on the analysis performed, the 6-cylinder (V6) vehicle configuration is able to satisfy its acceleration requirement but a similar analysis showed that the 4-cylinder (V4) vehicle configuration does not satisfy the requirement.

3.4.17 Using the *Vehicle Controller* to Optimize Engine Performance

The analysis results showed that the V6 configuration is needed to satisfy the vehicle acceleration requirement. Additional analysis is needed to assess whether the V6 configuration can satisfy the fuel efficiency requirement for a minimum of 25 miles per gallon under the stated driving conditions as specified in the *Fuel Efficiency* requirement in Figure 3.2.

The activity diagram in Figure 3.16 is a refinement of a portion of the *Provide Power* activity diagram in Figure 3.11. In this figure, the *vehicle controller* software has been added as an activity partition to support the analysis needed to optimize

**FIGURE 3.16**

Activity diagram used to analyze the vehicle controller software interaction with the engine and transmission to optimize fuel efficiency and engine performance. This diagram is a refinement of a portion of the activity diagram in Figure 3.11.

fuel efficiency and engine performance. The *vehicle Controller* includes an action to *control fuel-air mixture* that in turn produces the engine accelerator command. The inputs to this action include the *Accelerator Cmd* from the *Driver* and *Engine Parameter* such as revolutions per minute (RPM) and engine temperature. The *vehicle Controller* also includes the *Control Gear* action to determine when to change gears based on engine speed (i.e., RPM) to optimize performance. The specification of the vehicle controller software can include a state machine diagram that changes state in response to the inputs consistent with the state machine diagram in Figure 3.8.

The specification of the algorithms to realize these actions requires further analysis. A parametric diagram can specify the required fuel and air mixture in terms of RPM and engine temperature to achieve optimum fuel efficiency, and they can be used to constrain the input and output of the actions. The algorithms must implement these constraints by controlling fuel flow rate and air intake, and perhaps other parameters. The algorithms, which consist of mathematical and logical expressions, can be captured in an activity diagram or directly in code. Based on the previous engineering analysis—the details of which are omitted here—the V6 engine is able to satisfy the fuel efficiency requirements and is selected as the preferred vehicle system configuration.

3.4.18 Specifying the *Vehicle* and Its Components

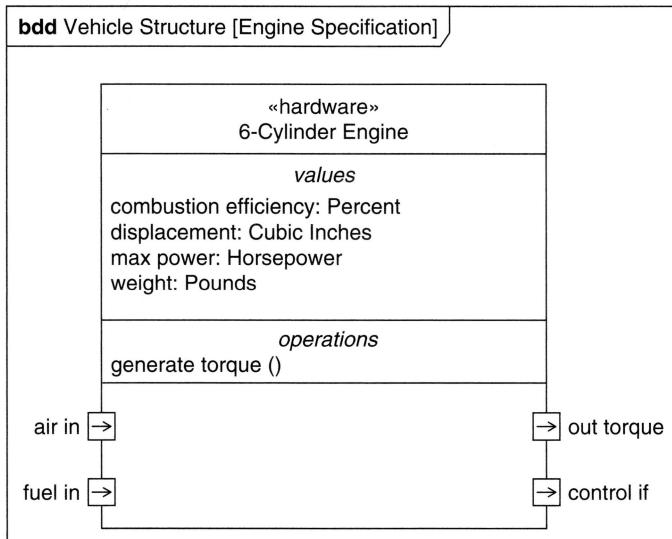
The block definition diagram in Figure 3.10 defined the blocks for the *Vehicle* and its components. The preceding analysis is used to specify the features of the blocks in terms of the functions they perform, their interfaces, and their performance and physical properties. Other aspects of the specification may include a state machine for state-based behavior and definitions of items that are stored by the block, such as fuel.

A simple example is the specification of the *Engine* block shown in Figure 3.17. This block was originally shown in the *Vehicle Hierarchy* block definition diagram in Figure 3.10. In this example, the *Engine* hardware element performs a function called *generate torque*, with ports that specify its interfaces to *air in*, *fuel in*, *control if*, and *out torque*. Selected properties are shown that represent performance and physical properties including its *displacement*, *combustion efficiency*, *max power*, and *weight* along with their **units**. The property values may also be represented as either a single value or a distributed value. Other blocks are specified in a similar way.

3.4.19 Requirements Traceability

The *Automobile System Requirements* were shown in Figure 3.2. Capturing the text-based requirements in the SysML model provides the means to establish traceability between the text-based requirements and other parts of the model.

The requirements traceability for the *Maximum Acceleration* requirement is shown in Figure 3.18. The requirement is **satisfied** by the *Power Subsystem*. The **rationale** refers to the engineering analysis based on the *Vehicle Acceleration Analysis* parametric diagram in Figure 3.14. The *Max Acceleration test case* is

**FIGURE 3.17**

The block definition diagram shows the *Engine* block and the features used to specify the block. This block was previously shown in the *Vehicle Hierarchy* block definition diagram in Figure 3.10.

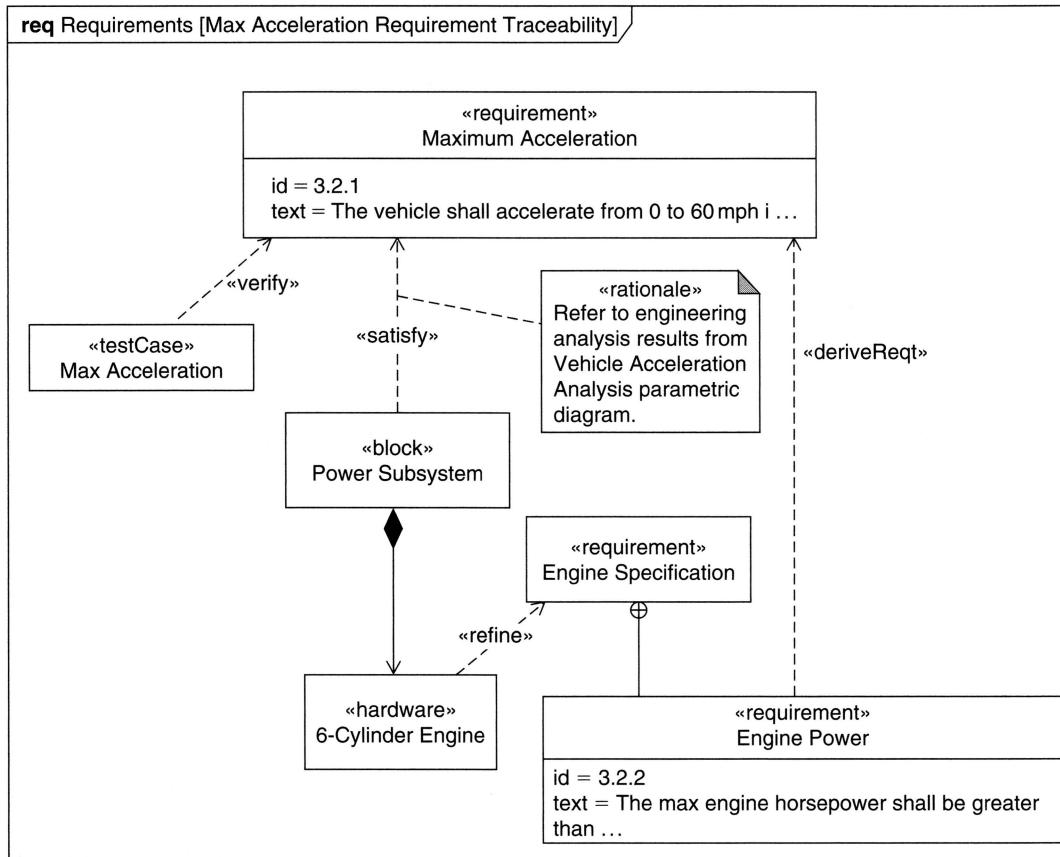
also shown as the method to verify that the requirement is satisfied. In addition, the *Engine Power* requirement is derived from the *Max Acceleration* requirement and contained in the *Engine Specification*. The *Engine* block refines the *Engine Specification* by restating the text requirements in the model. In this way, the system requirements can be traced to the system design and test cases, along with rationale.

The direction of the arrows points from the *Power Subsystem* design, *Max Acceleration* test case, and *Engine Power* requirement to the *Max Acceleration* as the source requirement. This is in the opposite direction that is often used to represent requirements flow-down. The direction represents a dependency from the design, test case, and derived requirement to the source requirement, such that if the source requirement changes, the design, test case, and derived requirement should also change.

As stated previously, there are other requirements relationships for trace and copy. The requirements are supported by multiple notation options including a tabular representation. Details of how SysML requirements and their relationships are modeled are described in Chapter 12.

3.4.20 Package Diagram for Organizing the Model

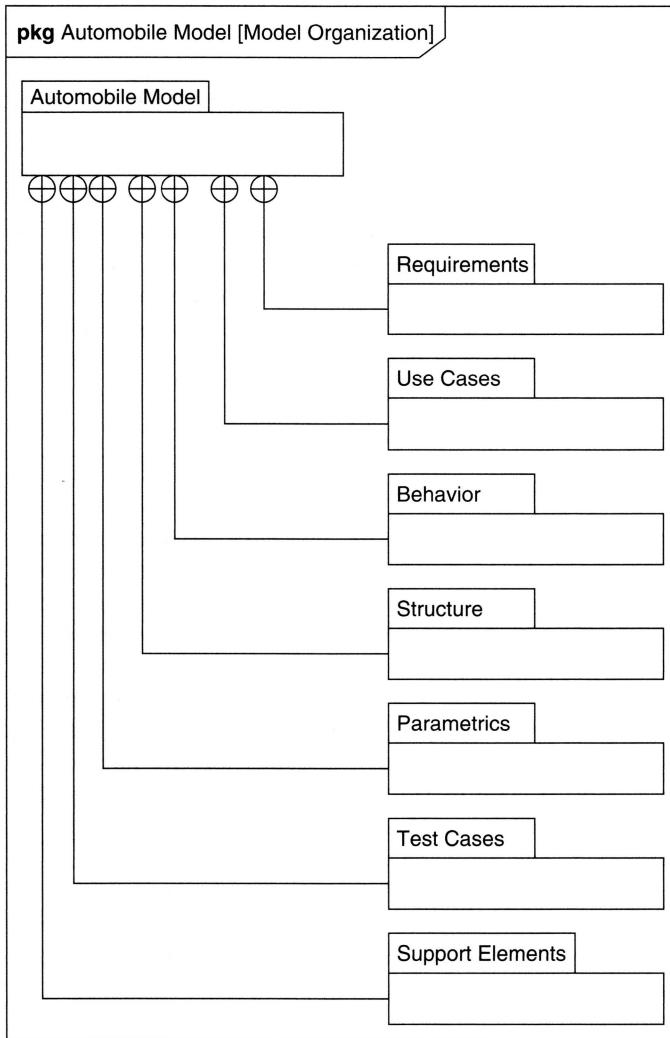
The concept of an integrated system model is a foundational concept for MBSE as described in Chapter 2. The **model** contains all of the **model elements**. The

**FIGURE 3.18**

Requirement diagram showing the traceability of the *Max Acceleration* requirement that was shown in the *Automobile Specification* in Figure 3.2. The traceability to a requirement includes the design elements that satisfy it, other requirements derived from it, and test cases to verify it. Rationale for the traceability relationships is also shown.

model elements and their relationships are captured in a model repository and can be displayed on diagrams. The model elements are integrated such that a model element that appears on one diagram may have relationships to model elements that appear on other diagrams. An example is the *Road* property, such as the *incline* angle that appears as a property of *Road* in the block definition diagram in Figure 3.3, and also is bound to a parameter of a constraint in the parametric diagram in Figure 3.14. The diagrams represent a view into this model.

A model organization is essential to managing the model. A well-organized model is akin to having a set of drawers to organize your supplies, where each supply element is contained in a drawer, and each drawer is contained in a particular cabinet. This facilitates understandability, access control, and change management of the model.

**FIGURE 3.19**

Package diagram showing how the model is organized into packages that contain model elements that comprise the *Automobile Domain*. Model elements in packages are displayed on diagrams. Model elements in one package can be related to model elements in another package.

The **package diagram** in Figure 3.19 shows how the model elements for this example are organized into **packages**. Each package **contains** a set of model elements. Model elements in one package can be related to model elements in another package. However, model organization enables each model element to be uniquely identified by the package that contains it. The model organization is generally similar to the view that is shown in the tool browser. Details on how to organize the model with packages are given in Chapter 5.

3.4.21 Model Interchange

A SysML model that is captured in a model repository can be imported and exported from a SysML-compliant tool in a standard format called **XML metadata** interchange (XMI). This enables other tools to exchange this information if they also support XMI. An example may be the ability to export selected parts of the SysML model to another UML tool to support software development of the controller software, or to import and export the requirements from a requirements management tool, or to import and export the parametric diagrams and related information to engineering analysis tools. The ability to achieve seamless interchange capability may be limited by the quality of the model and how the tool implements the standard, but this capability continues to improve. A description of XMI is included in Chapter 17.

3.5 Summary

SysML is a general-purpose graphical language for modeling systems that may include hardware, software, data, people, facilities, and other elements within the physical environment. The language supports modeling of requirements, structure, behavior, and parametrics to provide a robust description of a system, its components, and its environment.

The semantics of the language enable a modeler to develop an integrated model where model elements on one diagram can be related to model elements on other diagrams. The diagrams enable capturing and viewing the information in the model repository to help specify, design, analyze, and verify systems. The repository information can be imported and exported to exchange model data via the XMI standard and other exchange mechanisms.

The SysML language is a critical enabler of MBSE and can be used with a variety of processes and methods. However, effective use of the language requires a well-defined MBSE method. The automobile example illustrated the use of one such method. Other examples are included in Part III.

3.6 Questions

1. What are some of the aspects of a system that SysML can represent?
2. What is a requirement diagram used for?
3. What is an activity diagram used for?
4. What is a sequence diagram used for?
5. What is a state machine diagram used for?
6. What is a use case diagram used for?
7. What is the primary unit of structure in SysML?
8. What is the block definition diagram used for?
9. What is an internal block diagram used for?
10. What is a parametric diagram used for?
11. What is a package diagram used for?

Stephen Biering-Sørensen, Finn Overgaard Hansen, Susanne Klim, Preben Thalund Madsen

Struktureret Program-Udvikling (SPU),

Teknisk Forlag, ISBN 87-571-1046-8.

Vejledning i review, pp 215 - 236

Vejledning i review

Indhold

- 1. Indledning** 217
 - 1.1 Reviewets komponenter 218
 - 1.2 Reviewets faser 219
- 2. Planlægning** 220
 - 2.1 Fastsættelse af tidspunkt 220
 - 2.2 Udvælgelse af deltagere 221
 - 2.3 Klargøring af dokument 223
 - 2.4 Fremfinding af materiale 223
 - 2.5 Indkaldelse 224
- 3. Formøde** 225
 - 3.1 Reviewets formål 225
 - 3.2 Deltagernes roller 225
 - 3.3 Overordnet gennemgang af produktet 226
 - 3.4 Overordnet gennemgang af dokumentet 226
- 4. Forberedelse** 227
- 5. Reviewmøde** 228
 - 5.1 Reviewmødets forløb 228
 - 5.2 Andre forløb 229
 - 5.3 Reviewernes opgaver 229
 - 5.4 Reviewlederens opgaver 231
 - 5.5 Referentens opgaver 232
 - 5.6 Tilhørernes opgaver 232
- 6. Efterbehandling** 233
 - 6.1 Referat 233
 - 6.2 Opfølgning 233
 - 6.3 Registrering af tidsforbrug 233
- 7. Variationer af reviewteknikken** 234
 - 7.1 Uformelle reviews 234
 - 7.2 Korrekturlæsning 234
 - 7.3 Tekniske gennemgange 234
 - 7.4 Kodegranskning 235
 - 7.5 Inspektioner 235
- 8. Vejledningens hovedpunkter** 236
- Litteraturliste** 237

1. Indledning

Et review er et møde, hvor folk uden for projektet ("reviewere") fremlægger deres kritik af et dokument.

Dokumenter, der kan reviewes, er f.eks.

- kravspecifikationer
- testspecifikationer
- program- og procesdesign
- modulspecifikationer
- kildetekster

dvs. det meste skriftlige materiale inden for projektet. Også ikke-tekniske dokumenter, som f.eks. projektgrundlag, kan reviewes. I *Vejledning i struktureret programudvikling* beskrives, hvornår i udviklingsforløbet der holdes review. Review benyttes som regel ved *milepæle* i programudviklingen, f.eks. når man er nået til slutningen af en fase.

Formålet med et review er dels at finde *fejl og mangler* ved dokumentet, dels at påpege hvilke dele af dokumentet, der er gode (*fortræffeligheder*).

Nogle gange kan formålet endvidere være at godkende eller forkaste dokumentet. Hvis dokumentet bliver forkastet, må det gennemgå en opretning, hvor de påpegede fejl rettes, og derpå undergå et nyt review. Mere normalt er det dog, at reviewet ikke ender med en sådan "karaktergivning", men blot konstaterer eventuelle fejl og mangler, og overlader beslutningen om dokumentets videre skæbne til projektgruppen.



Grunden til at man med reviewteknikken kan finde flere fejl, end man ellers ville kunne, er, at der bliver set på dokumentet med *friske øjne*. Det er som regel lettere at finde fejl i andres dokumenter end i sit eget!

Et biprodukt ved at holde reviews er *uddannelseseffekten*. Dels bliver reviewerne uddannet ved at se andres måde at gøre tingene på, dels bliver forfatteren uddannet ved at blive gjort opmærksom på problemer, der ikke var tænkt på.

Der findes flere *varianter* af reviews, som alle kan anvendes i projektarbejdet. Det drejer sig f.eks. om:

- uformelle reviews
- korrekturlæsning
- tekniske gennemgange
- kodegranskning
- inspektioner

Denne vejledning vil hovedsageligt behandle *formelle reviews*, men de andre typer vil blive gennemgået i kapitel 7.

De efterfølgende 5 kapitler handler om hver sin fase af det formelle review:

- planlægning
- formøde
- forberedelse
- reviewmøde
- efterbehandling

Resten af dette kapitel omhandler dels *reviewets komponenter* dels *reviewets faser*.

1.1 Reviewets komponenter

I dette afsnit gennemgås kort de forskellige komponenter, der indgår i et review, samt hvilken funktion de har. Det drejer sig om

- materialet
- deltagerne og
- resultatet

Materialet

For at et review kan gennemføres, skal der være noget skriftligt materiale til stede, nemlig

- et dokument, som er den specifikation eller den kildetekst, der skal reviewes
- noget *baggrundsmateriale*, som er den information, der er nødvendig for at forstå dokumentet
- en *standard for dokumentets udformning og indhold*, som giver retningslinjer for hvorledes dokumentet skal være udformet. Denne standard er dog ikke en absolut nødvendighed.
- *checklister* (hvis de findes, og hvis de ikke er en del af standarden for dokumentets udformning og indhold).

Deltagerne

De roller, der er involveret i et review, er følgende:

- *forfatteren*, som er den person (eller de personer), der har skrevet dokumentet
- *reviewerne*, som er de personer, der skal gennemgå dokumentet med henblik på at finde fejl, mangler og fortræffeligheder

- *reviewlederen*, som er den person, der sørger for de praktiske arrangementer omkring reviewet, dvs. indkaldelse, kopiering og uddeling af materiale samt mødeledelse på formøde og reviewmøde
- *referenten*, som noterer de kritikpunkter, der rejses på reviewmødet. Normalt er forfatteren selv referent
- *tilhørerne*, som typisk vil være personer fra projektgruppen, der gerne vil overvære reviewet. Også personer fra andre projekter kan overvære et review, for at blive fortrolige med teknikken, inden de måske selv skal deltagte i et review.

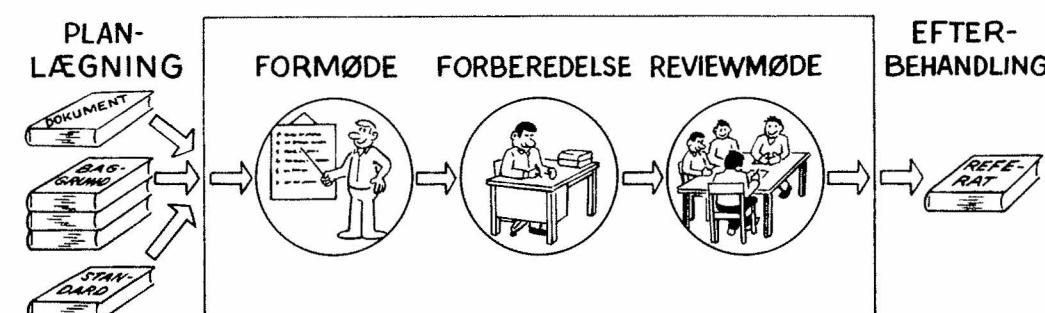
Resultatet

Det synlige resultat af et review er *referatet*, som indeholder reviewernes kritik af dokumentet. Referatet kan også indeholde reviewmødets *godkendelse eller forkastelse* af dokumentet. Dette referat skrives af referenten.

1.2 Reviewets faser

Et review kan inddeltes i følgende fem faser:

- *planlægning*, hvor deltagerne udvælges og materialet gøres klar
- *formøde*, hvor forfatteren præsenterer dokumentet for reviewerne, hvor reviewets formål præciseres og hvor forløbet fastlægges
- *forberedelse*, hvor reviewerne hver for sig kritisk gennemgår dokumentet
- *reviewmøde*, hvor reviewerne fremlægger deres kritik
- *efterbehandling*, hvor der udgives referat, og hvor dokumentet rettes med hensyn til de fejl og mangler, reviewerne har påpeget.



Figur 1. Reviewets faser

2. Planlægning

I planlægningsfasen skal følgende aktiviteter udføres:

- fastsættelse af tidspunkt for review
- udvælgelse af deltagere
- klargøring af dokument
- fremfinding af materiale
- indkaldelse

Disse aktiviteter beskrives i de følgende afsnit.

2.1 Fastsættelse af tidspunkt

Det skal både planlægges hvilken dato, reviewet skal holdes, hvor når på dagen, det skal foregå, og hvorlænge.

Dato

Reviews skal være med i planen fra starten, men det nøjagtige tidspunkt fastsættes af forfatteren. Det er nemlig vigtigt, at dokumentet ikke sendes til review *før det er færdigt*. Man opnår intet ved at reviewe et halvfærdigt dokument - højst irritation over, at der findes for mange fejl og mangler, som forfatteren sikkert allerede kendte!

Lad være med at udskyde et review, fordi der "ikke er tid lige nu". Projektet gør sig selv en bjørnetjeneste ved at arbejde videre med et dokument, der indeholder fejl.

Derimod kan forfatteren godt arbejde videre med dokumentets efterfølger, mens reviewet er i forberedelsesfasen (efterfølgeren er f.eks. kildeteksten, hvis dokumentet er modulspecifikationen). Man skal da blot huske at rette de fejl, reviewerne påpeger, også i dokumentets efterfølger.

Klokkeslet

Det er en god ide, at placere reviewet først på formiddagen, f.eks. kl. 9.30 til 11.30. Det er som regel det tidspunkt, folk er mest oplagte.



Reviewmødets varighed

Et reviewmøde bør *aldrig være længere end 2 timer!* Erfaringen viser, at efter 2 timer falder koncentrationsevnen væsentligt. Hvis

dokumentet er for stort til, at man kan nå det hele igennem på 2 timer, er det bedre at dele reviewet over flere dage.

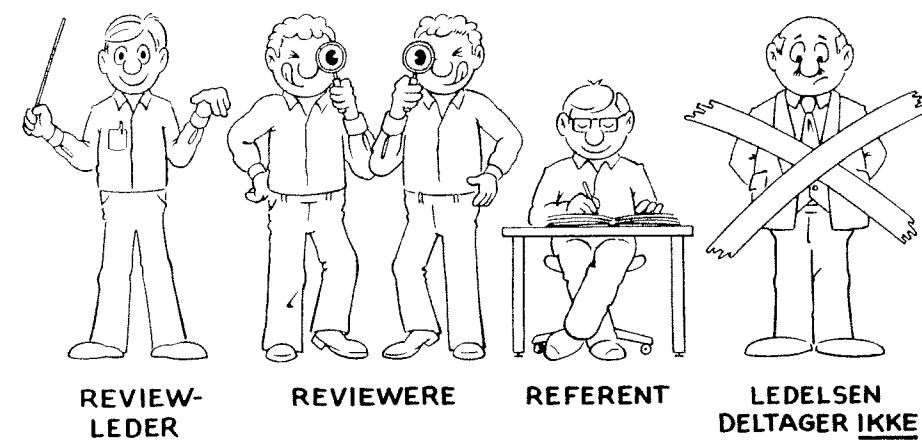
Kommer reviewerne udenbys fra, kan man naturligvis dispensere fra denne regel. Man må da holde gode pauser med passende mellemrum i stedet.

2.2 Udvælgelse af deltagere

Deltagerne bør findes så tidligt som muligt, dvs. helst allerede når reviewet planlægges i projektplanen. På denne måde kan de indpasse deltagelsen i deres egne tidsplaner.

Der må *ikke deltage personer fra ledelsen*. Dette punkt er vigtigt, idet det er *dokumentet*, der skal reviewes, ikke *forfatteren*. Deltager repræsentanter fra ledelsen, vil det næppe kunne undgås, at der fokuseres mere på forfatteren, end der ellers ville være blevet. Dette har den naturlige konsekvens, at forfatteren stiller sig i *forsvarsposition* fremfor at lægge sit dokument åbent frem. I yderste konsekvens vil ingen godvilligt gøre sit dokument til genstand for et review.

Ved review af kravspecifikationer kan repræsentanter for kunden, salgsafdelingen eller andre relevante afdelinger være gode reviewer.



Reviewets deltagere

Reviewlederen

Den første person, der skal udvælges, er *reviewlederen*. Reviewlederen udpeges af ledelsen i samarbejde med projektgruppen. Reviewlederen kan f.eks. være en anden af projektdeltagerne, men det er bedst, hvis det er en person uden for projektgruppen, da en sådan har lettere ved at være neutral.

Reviewlederen er reviewets "praktiske gris" og skal derfor være en god organisator. Reviewlederen skal også være en god mødeleder, dvs. have styrke til at afbryde begyndende skænderier og personlighed til at kunne skabe en god stemning omkring reviewet.

Reviewerne

Reviewerne findes af reviewlederen i samarbejde med ledelsen.

De personer, der skal være *reviewere*, skal

- være teknisk kompetente inden for området
- have diplomatisk sans
- kunne være i stue sammen

For at alle software-folk i firmaet (med de nævnte kvalifikationer) får lejlighed til at være reviewere, kan man f.eks. indføre en "vagtoftning", således at 4 personer har "review-vagten" i f.eks. 3 måneder. Det betyder, at reviewlederen aldrig har problemer med at finde reviewere.

En anden måde at gøre sagen an på er, at reviewerne udpeges for et helt projekt ad gangen. Det er altså de samme personer, der er reviewere af såvel kravspecifikationer som program- og procesdesign, modulspecifikationer osv. Dette indebærer den fordel, at reviewerne ikke behøver bruge tid på at sætte sig ind i dokumentets omgivelser hver gang, men kan nøjes med at gøre det første gang.

Endelig kan man finde reviewerne fra gang til gang. Dette betyder et større arbejde for reviewlederen, men til gengæld kan man "håndplukke" personer, med interesse for netop det emneområde, det pågældende dokument dækker. Hvis en anden end forfatteren skal arbejde videre med dokumentet i den næste fase, vil vedkommende være en god kandidat som reviewer, da man er mere motiveret for at finde fejl og mangler, hvis man ved, at man selv skal overtage dokumentet.

Normalt vil man have to reviewere, men i enkelte tilfælde kan man have flere. Dette kan f.eks. være tilfældet, hvis man reviewer en kravspecifikation, hvor man gerne vil have flere synspunkter repræsenteret. Det er dog ikke en god ide med mere end 3-4 reviewere. Dels falder reviewernes engagement ("De andre finder nok fejlene!"), dels bliver det svært at overholde de to timer, reviewmødet højst må vare, hvis for mange skal have taletid. Og ofte vil to grundige reviewere alligevel finde fejlene, manglerne og fortræffelighederne.

Referenten

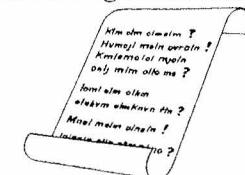
Referenten kan f.eks. være en af projektdeltagerne, men i praksis vil forfatteren selv være en god referent.

Tilhørerne

Tilhørerne må naturligvis dukke op af sig selv, men de skal vide, at reviewet holdes, så det skal annonceres ordentligt. Personer, der ikke har deltaget i et review før, bør inviteres specielt til at være tilhørere, så de kan lære teknikken.

2.3 Klargøring af dokument

Dokumentet skal være færdigt til review og i overensstemmelse med standarden for det pågældende dokument. Bemærk at det er *forfatteren*, der bestemmer, hvornår dokumentet er færdigt til review. Som nævnt tidligere, opnår man intet ved at forsøge at tvinge et review igennem af et dokument, der ikke er klart.



Som en del af denne klargøring til review kan forfatteren fremstille en liste af spørgsmål eller særligt kritiske områder i dokumentet, som reviewerne skal overveje.

Kan det lade sigøre at udskrive dokumentet med linienumre (eller kopiere det på papir med linienumre), vil det lette reference til specifikke linier på selve reviewmødet.

En typisk størrelse for et dokument, der ønskes reviewet er 20-30 sider. Dette svarer til, hvad man kan nå igennem på de to timer, reviewmødet højst må vare. Vælger man at dele reviewmødet over flere dage, kan man naturligvis godt reviewe større dokumenter.

2.4 Fremfinding af materiale

Alt det baggrundsmateriale, der er nødvendigt under læsningen af dokumentet, skal findes frem. Det kan f.eks. dreje sig om

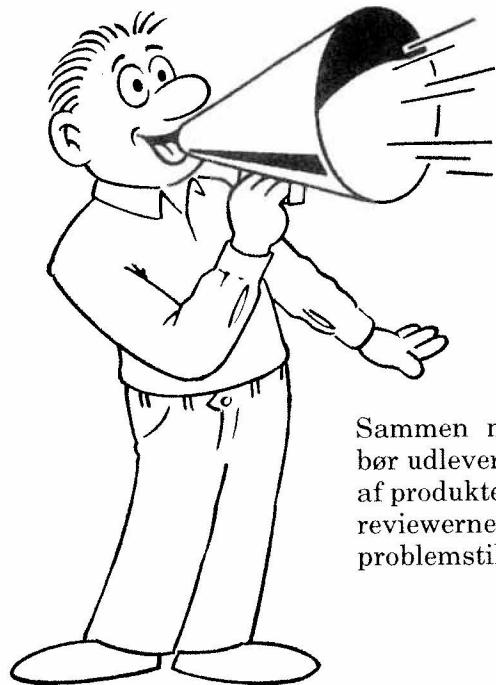
- en overordnet beskrivelse på introducerende niveau af det produkt, hvor dokumentet indgår
- forgænger dokumentet (hvis dokumentet, der skal reviewes, er en modulspecifikation, så er forgænger dokumentet lig med procesdesignet)
- skrifter, der henvises til fra dokumentet, eller som er nødvendige for forståelsen af dokumentet. Det kan f.eks. dreje sig om beskrivelser af fælles datastrukturer eller grænseflader.

Ud over dette baggrundsmateriale, skal man også sørge for, at reviewerne får den standard, dokumentet følger, hvis en sådan findes. Endelig skal man sørge for, at reviewerne får eventuelle checklister for dokumentets udformning og indhold, hvis de findes. Ofte vil disse checklister være en del af standarden for dokumentet.

Det er forfatteren, der fremfinder baggrundsmaterialet.

2.5 Indkaldelse

Der skal indkaldes til såvel formøde som reviewmøde i god tid.



Sammen med indkaldelsen til *formødet* bør udleveres den overordnede beskrivelse af produktet (hvis en sådan findes). Så har reviewerne en chance for at sætte sig ind i problemstillingen på forhånd.

De reviewere, der *ikke* har deltaget i et review før, bør også have udleveret denne *Vejledning i review* sammen med indkaldelsen til formødet.

Det er reviewlederen, der er ansvarlig for, at deltagerne bliver indkaldt, og at materialet bliver kopieret.

Af hensyn til eventuelle tilhørere, bør indkaldelsen annonceres offentligt, f.eks. på opslagstavler.

- Har man kendskab til personer, der kunne være interesserede i dokumentet, bør de have en speciel invitation til at deltage i reviewet som tilhørere.

Følgende skal kopieres til reviewerne:

- dokument (evt. i to eksemplarer, hvoraf det ene bruges til marketing af korrekturretter og det andet bruges til de andre kommentarer)
- baggrundsmateriale
- checklister
- spørgsmål til reviewerne
- evt. denne *Vejledning i review*

Det udleverede materiale kan f.eks. samles i et ringbind med fanebladsforside, så det er let for reviewerne at finde rundt i.

3. Formøde

Formålet med formødet er dels at reviewerne skal få et klart billede af, hvad der forventes af dem, dels at dokumentet skal introduceres for deltagerne. Formødet kan f.eks. forløbe efter følgende dagsorden:

Dagsorden:

1. Reviewets formål (ved reviewlederen)
 - . Hvad skal reviewes?
 - . Reviewets formål
2. Deltagernes roller (ved reviewlederen)
 - . Diskussion af reviewets teknik
 - . Dagsorden for selve reviewmødet
 - . Udlevering af spørgsmål til reviewerne
3. Overordnet gennemgang af produktet (ved en fra projektgruppen)
4. Overordnet gennemgang af dokumentet (ved forfatteren)
 - . Formål
 - . Funktioner
 - . Grænseflader (hvis relevant)
 - . Datastruktur (hvis relevant)
 - . Logisk struktur (hvis relevant)
 - . Gennemgang af det udleverede baggrundsmateriale
 - . Gennemgang af spørgsmål til reviewerne

Figur 2. Dagsorden for formøde

3.1 Reviewets formål

Reviewlederen indleder med at fortælle hvad det er, der skal reviewes, og hvad formålet med reviewet er. Man kan evt. fortælle lidt om baggrunden, hvilke reviews, der tidligere har været holdt (dvs. dokumentets forgængere) osv.

Under dette punkt udleveres også ringbindet med dokument, baggrundsmateriale og dokument-standard.

3.2 Deltagernes roller

Det er vigtigt, at reviewerne er helt klar over, hvad der forventes af dem. Hvis der derfor er "nye" reviewere med, bør der bruges noget tid på at diskutere selve reviewteknikken og specielt reviewernes "rolle-beskrivelse" (se senere i kapitlet om selve reviewmødet).

Reviewlederen bør også fortælle om *dagsordenen* for selve reviewmødet, så reviewerne kan tilrettelægge deres fremlæggelse efter det. Det kan f.eks. være, at reviewlederen finder det hensigtsmæssigt, at tage *alle* kommentarer til et *afsnit* i dokumentet på én gang. Eller det kan være, at reviewerne skal aflevere deres væsentlige kritikpunkter skriftligt af hensyn til referenten. Alt dette skal de vide på forhånd.

3.3 Overordnet gennemgang af produktet

For at give reviewerne en fornemmelse af den *sammenhæng*, dokumentet indgår i, fortæller en person fra projektgruppen (gerne forfatteren selv) om selve produktet. Dette punkt kan reviewerne evt. have forberedt sig på, hvis de har fået udleveret en overordnet introduktion til produktet sammen med indkaldelsen til formødet.

Punktet er selvfølgelig overflødig, hvis reviewerne er "faste reviewerne" på projektet, og dermed har kendskab til produktet fra tidligere reviews.

3.4 Overordnet gennemgang af dokumentet

For at lette reviewernes tilegnelse af selve dokumentet, gennemgås det af forfatteren (på overordnet plan). Man kan f.eks. fortælle om formål, struktur, kommunikation med andre moduler osv.

Forfatteren bør også gennemgå baggrundsmaterialet, for at reviewerne kan få et overblik over, hvor de kan finde hvad.

Har forfatteren forberedt nogle *spørgsmål til dokumentet*, som reviewerne særligt skal have opmærksomheden henledt på, gennemgås de på dette tidspunkt.

4. Forberedelse

I denne fase skal reviewerne hver for sig studere dokumentet med henblik på at finde logiske fejl, mangler, afgivelser fra standarden og fortræffeligheder ved dokumentet. Dette er *reviewets vigtigste fase*.

Længden af forberedelsesfasen er afhængig af flere faktorer, f.eks.

- dokumentets kompleksitet
- baggrundsmaterialets omfang og
- reviewernes erfaring

En typisk forberedelsestid vil være 3-6 minutter pr. side (se f.eks. /Skogstad86/).

Under dette arbejde kan reviewerne f.eks. benytte de checklister for "almindeligt forekommende fejl", der bør indgå i standarden for det pågældende dokument.

Det er en god ide, at dele kommentarerne i tre grupper:

- *Trykfejl*, stavefejl og andre korrekturrettelser. Disse kan f.eks. noteres med rødt direkte i dokumentet og udleveres til forfatteren ved reviewmødets start.
- *Afgivelser fra standarden* (f.eks. forkert side/afsnitsnummere-ring eller udeladte afsnit). Disse vil typisk vedrøre dokumentets form, og derfor blive gennemgået først på reviewmødet. Det er derfor lettest, hvis de er skilt ud for sig selv.
- *Logiske fejl* og mangler samt *fortræffeligheder*. Logiske fejl kan f.eks. være uopfyldelige krav i en kravspecifikation eller mulighed for baglås i et programdesign. Afsløring af denne type fejl er naturligvis en af reviewernes fornemmeste opgaver. En god hjælp til at finde *manglerne*, er de uddelte checklister. Fortræffeligheder ved dokumentet kan f.eks. være en god måde at løse et bestemt problem på, en overskuelig struktur, etc. Det er vigtigt at påpege fortræffelighederne, så de ikke forsvinder ved en eventuel rettelse af dokumentet.

5. Reviewmøde

I dette kapitel gennemgås dels hvordan selve reviewmødet forløber, dels hvilke *roller*, de forskellige deltagere har under mødet.

5.1 Reviewmødets forløb

Et reviewmøde kan struktureres på mange måder. Den måde, der gennemgås i dette afsnit, er blot en enkelt. Der er heller ingen grund til at stå stift på dagsordenen. Hvis f.eks. en deltager undervejs finder på en kommentar, som egentlig hører hjemme under et punkt, der allerede er behandlet, så bør kommentaren tages med aligevel.

Reviewmødets forløb kan inddeltes i følgende punkter:

- Korrekturfejl
- Kommentarer til dokumentets form
- Generelle kommentarer til dokumentet
- Detaljeret gennemgang af dokumentet
- Evt. konklusion (godkendelse/forkastelse)

Under hele mødet noterer referenten de punkter, der rejses af reviewerne.

Korrekturfejl

Korrekturfejl (stavefejl, trykfejl o.lign.) kan behandles hurtigt ved at reviewerne udleverer deres korrekturrettertelser skriftligt til forfatteren (f.eks. markeret med rødt i en kopi af dokumentet).

Kommentarer til dokumentets form

Disse kommentarer drejer sig om, at dokumentet afviger fra den standard, det skal følge. Dette behøver ikke nødvendigvis at være dokumentets fejl, det kan også være standardens fejl. I så fald skal standarden laves om! Det skal dog understreges, at en eventuel diskussion om standarden *ikke* skal foregå på reviewmødet.

Generelle kommentarer

Reviewerne fremkommer med deres generelle (overordnede) kommentarer til dokumentet. De kan komme med én kommentar ad gangen på skift, eller de kan skiftes til at komme med alle deres kommentarer på én gang.

Detaljeret gennemgang

Reviewerne kommer med deres detaljerede kommentarer, dvs. afvigelser fra standarden, logiske fejl, mangler samt fortræffeligheder. Som tidligere nævnt kan de skiftes til at komme med en kommentar, eller hver reviewer kan komme med alle sine kommentarer på én gang.

Dette kan f.eks. foregå side for side eller funktion for funktion.

Konklusion

Hvis reviewet skal ende med en konklusion, skal dette punkt behandles som det sidste på reviewmødet.

Reviewerne skal være klar over, at de *forpligter sig* både ved at godkende dokumentet og ved at forkaste det. Men selv om reviewerne således bliver gjort medansvarlige for dokumentets korrekthed, frøtager det naturligvis ikke forfatteren for ansvar! Evt. kan dokumentet godkendes med kommentarer.

Husk at det er lige så slemt, at godkende et dårligt dokument, som det er at forkaste et godt dokument!

5.2 Andre forløb

Den struktur, der er beskrevet oven for, behandler hele dokumentet under ét. Først de generelle kommentarer og så de detaljerede.

Man kan i stedet behandle dokumentet afsnit for afsnit, og lade reviewerne komme med først generelle kommentarer *til det pågældende afsnit* og derefter detaljerede kommentarer til det pågældende afsnit.

Man kan også gennemgå dokumentet side for side eller afsnit for afsnit, og lade reviewerne bestemme hvad de vil sige, altså uden at bede dem komme med generelle kommentarer først, men lade dem komme med kommentarerne i den rækkefølge, de selv vil.

Endelig kan man strukturere reviewmødets forløb efter de udlevede spørgsmål.

5.3 Reviewernes opgaver

I /Freedman82/ findes følgende gode råd for reviewerne:

- Vær forberedt
- Vær solidarisk
- Tal pænt

- Giv også positiv kritik
- Påpege, ikke løse
- Undgå diskussion om stil
- Kun tekniske emner

Den dybere mening med disse råd er:

Vær forberedt

Hvis du ikke har haft tid til at forberede dig grundigt, så sig det til reviewlederen *inden* reviewmødet, så mødet kan blive utsat. Lad være med at møde op og prøve at "bluffe dig igennem". I bedste fald bliver du gennemskuet, og de andre deltagere bliver sure over, at du har spildt deres tid. I værste fald lykkes det for dig, og de fejl, du skulle have fundet i dokumentet, bliver ikke opdaget.

Vær solidarisk

Lad være med at være "bedrevidende" over for forfatteren. Ideen med reviewformen er netop, at man skiftes til at reviewe hinandens dokumenter. Derfor er det måske dig, der er forfatter, næste gang. Prøv derfor at fokusere på dokumentet frem for på forfatteren.

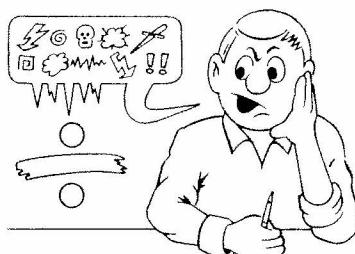
Tal påent

Husk på, at forfatteren er i en meget sårbar position. Det er ikke alle, der synes, det er rart, at få sat sine fejl og mangler til skue for andre. Tænk derfor på hvordan du kan formulere din kritik så diplomatisk som muligt. F.eks. hedder det ikke

- Det er da for dumt, at der ikke er taget højde for overløb. Enhver kan da se med et halvt øje, at programmet går i indeksfejl, når kataloget bliver fuldt.

I stedet kan man f.eks. sige:

- Jeg tror nok, der er et problem med overløb. Er det ikke rigtigt, at programmet går i indeksfejl, hvis kataloget løber fuldt?



Så vil forfatteren bladre lidt i sin specifikation, og sige "nåh jo, det har jeg da vist ikke tænkt på", og han vil være glad over at have fået udryddet en fejl, i stedet for at være sur over, at have fået at vide, at han er for dum!

Som regel bliver spørgsmål formuleret med "Hvorfor ..." opfattet mere negativt end spørgsmål, der starter med "Jeg forstår ikke helt ...".

Giv også positiv kritik

Prøv også at finde *positiv* kritik. Det lyder banalt, men negativ kritik glider som regel bedre ned, hvis man også bliver rost:

- Jeg kan godt lide den overskuelige måde dette modul er beskrevet på, men er der ikke et problem med ...

Du behøver ikke indlede *hver* negativ kritik med noget positivt.

Den positive kritik (dvs. fortræffelighederne) er også forebyggende overfor ændringer af de dele, der vitterlig er gode.

Påpege, ikke løse

Husk på, at det er din opgave som reviewer at *påpege* problemer, men du skal *ikke løse* dem. Det er forfatterens opgave, og han/hun skal have lov til at gøre det på sin egen måde. Desuden tillader reviewmødets korte tid heller ikke, at man går ind i diskussioner om alternative måder at gøre tingene på.

Hvis reviewerne har forslag til *andre* måder at løse problemerne på - og hvis forfatteren er interesseret - kan de f.eks. mødes med forfatteren *efter* reviewmødet.

Undgå diskussion om stil

Ting kan laves på mange måder, og du er måske ikke altid enig i den måde, forfatteren har valgt. Spørgsmålet er imidlertid ikke, om du kan lide forfatterens *stil*, men om det, der står, er korrekt. Prøv derfor at se bort fra, at dokumentet måske ikke lige er lavet, som du ville have lavet det. Koncentrer dig i stedet om, hvorvidt det opfylder specifikationerne og overholder standarden.

Standarden kan dog også omhandle stilen, og i så fald er det et emne, der kan diskuteres på reviewet!

Bemærk at hvis et dokument er svært at vedligeholde, så er det *ikke* et spørgsmål om stil.

Kun tekniske emner

Reviewets opgave er at finde *tekniske* problemer i dokumentet. Derimod er *betingelserne*, hvorunder dokumentet er frembragt (f.eks. tidsplaner og bemanding), irrelevante. Det er *dokumentet*, der skal reviewes.

5.4 Reviewlederens opgaver

Det er reviewlederens ansvar, at

- reviewet bliver struktureret bedst muligt
- reviewets praktiske afvikling forløber bedst muligt
- ingen brænder inde med kommentarer
- diskussionen ikke løber af sporet

Det er derimod *ikke* reviewlederens ansvar, at

- alle kommer til tiden
- alle er forberedte

Det er *alle deltageres* ansvar! Men naturligvis kan reviewlederen godt være "indpisker" og minde deltagerne om mødet lidt i forvejen, hvis erfaringen viser, at det er nødvendigt!

I et review med afsluttende godkendelse/forkastelse er det derimod reviewlederens *pligt*, at være opmærksom på, om alle deltagere er forberedte, eller om nogen forsøger at "bluffe". Vær f.eks. opmærksom på, om én hele tiden snakker de andre efter munden ("ja, det mener jeg også!"), eller hele tiden kommer med generelle kommentarer. Vær også opmærksom på, om én "læser foran" og finder på kommentarer undervejs. I denne situation må reviewlederen afbryde reviewet, og notere i referatet, at reviewet ikke har kunnet gennemføres af den og den grund. Der må derefter fastsættes et nyt tidspunkt for reviewet.

Det er imidlertid *kun* i denne situation, reviewlederen skal være vagthund! I et almindeligt review uden karaktergivning, kan det ødelægge en ellers god stemning, hvis alle føler sig overvågede.

5.5 Referentens opgaver

Referentens opgave er klar og éntydig, nemlig at

- referere reviewernes kritikpunkter så objektivt som muligt
- få referatet udgivet så hurtigt som muligt

Det sidste er ikke det mindst vigtige!

I nogle situationer kan det være en god ide, at lave et "offentligt referat", dvs. referenten skriver referatet på over-heads eller på flip-over, så alle deltagere kan se, hvad der bliver skrevet i referatet mens det bliver skrevet. På denne måde undgår man, at referenten misforstår eller overhører væsentlige punkter.

Referenten kan også læse svære passager højt.

En anden måde at sikre, at referenten får det hele med, er, at reviewerne afleverer deres kritikpunkter skriftligt til referenten, som så kan sammenskrive dem.

Selv om forfatteren selv er referent, fritager det ikke for at lave et godt og fyldigt referat.

5.6 Tilhørernes opgaver

Tilhørernes fornemmeste opgave er, at *holde mund* og i det hele taget forstyrre reviewet så lidt som muligt.

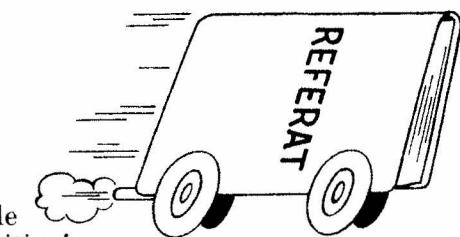
Mener en tilhører at have et *væsentligt* bidrag til diskussionen, kan man naturligvis godt henvende sig til reviewlederen og bede om at måtte komme med en kommentar. Men det kan virke meget irriterende på de andre deltagere, hvis tilhørerne hele tiden blander sig.

6. Efterbehandling

6.1 Referat

Som nævnt er det vigtigt, at referatet bliver udgivet så hurtigt som muligt.

Referatet bør naturligvis indeholde alle reviewernes kritikpunkter *også de positive!*



Er reviewet afsluttet med en konklusion, skal denne også med i referatet, gerne i indledningen.

Referatet *distribueres* til deltagerne i reviewet (så reviewerne kan få lejlighed til at kontrollere, at de ikke er blevet misforstået). I et review med godkendelsesfunktion, skal referatet også distribueres til ledelsen.

6.2 Opfølgning

De fejl og mangler, reviewerne har påpeget, skal naturligvis overvejes og eventuelt rettes. Det normale er, at forfatteren blot retter fejlene, og så sker der ikke mere.

Der kan imidlertid være rejst så alvorlige kritikpunkter (eller så dybtgående rettelser), at dokumentet skal gennem et nyt review.

Ved review af kravspecifikationer er det ofte ledelsen, der beslutter dette, men normalt er det projektgruppen eller forfatteren selv, der beslutter, at et fornyet review er nødvendigt.

6.3 Registrering af tidsforbrug

Til brug for projektplanlægning af fremtidige reviews er det vigtigt at vide, hvor lang tid, deltagerne har brugt på reviewet.

Reviewlederen bør derfor indsamle deltagernes tidsforbrug. Det kan eventuelt medtages i referatet.

Et typisk tidsforbrug i et review, med en reviewleder, to reviewerne og en forfatter, er (målt i persontimer):

$$20 + 0.16 * \text{AntalSider}$$

Idet der her er regnet med:

Planlægning: 4 persontimer

Formøde: 5 persontimer

Reviewmøde: 8 persontimer

Opfølgning: 3 persontimer

Og forberedelsen er sat til 5 minutter pr. reviewer pr. side.

7. Variationer af reviewteknikken

Der findes en række variationer af reviewteknikken, som man kan bruge alt efter situation og temperament.

7.1 Uformelle reviews

Den eneste væsentlige forskel på et formelt og et uformelt review er, at det *uformelle* review holdes inden for projektgruppen, og altså ikke involverer personer udefra.

I det uformelle review er det også tilladt at komme med løsningsforslag. Hvor det formelle review altid holdes på et *færdigt* dokument, der altså formodes at være fejlfrit, holdes det uformelle review ofte på dokumenter, der ikke er helt færdige.

7.2 Korrekturlæsning

Dette er ikke et egentligt "review". Forfatteren uddeler dokumentet til en eller flere personer, som læser det igennem og kommenterer det. Kommentarerne kan enten afleveres skriftligt, men bedre er det at supplere med en mundtlig uddybning.

Denne teknik er ikke helt så effektiv, som reviewet, men kan bruges, hvis det er svært at gennemføre et rigtigt review.

7.3 Tekniske gennemgange

Et review holdes altid *efter* dokumentet er færdigt, hvorimod en *teknisk gennemgang* (walk-through) er en metode, forfatteren kan benytte, *under fremstillingen* af dokumentet. Naturligvis kan en teknisk gennemgang også træde i stedet for et review, og altså holdes efter, dokumentet er færdigt.

Tekniske gennemgange holdes typisk inden for projektgruppen, og der må gerne fremsættes løsnings-/ændringsforslag.

Den tekniske gennemgang foregår ved at forfatteren gennemgår sit dokument (eller typisk en mindre del af det) på tavlen med et par andre projektdeltagere som tilhørere.

Tekniske gennemgange kræver heller ikke forberedelse af deltagerne, da materialet bliver gennemgået på mødet.

7.4 Kodegranskning

Review af kildetekster kaldes ofte kodegranskning. Kodegranskning er en slags "skrivebordstest" af den pågældende kildetekst, hvor revieweren ("granskeren") gennemspiller koden med tænkt input.

7.5 Inspektioner

En inspektion er en endnu mere formel udgave af reviewteknikken, som bl.a. omfatter klassificering og optælling af fejl, oplæsning af hele dokumentet på reviewmødet, speciel uddannelse af reviewlederen osv.

Denne teknik, som blev "opfundet" af Michael E. Fagan i 1976 er beskrevet i /Fagan76/, /Fagan86/ og /Skogstad86/.

8. Vejledningens hovedpunkter

Denne vejledning har beskrevet, hvorledes man gennemfører et review.

Reviewet er blevet opdelt i faserne planlægning, formøde, forbere-delse, reviewmøde og efterbehandling, og det er blevet beskrevet, hvad der udføres i de forskellige faser.

Under planlægningen er det specielt vigtigt, at dokumentet er klart til review, og at der vælges kompetente reviewere.

Under reviewmødet er det vigtigt, at reviewerne fremlægger deres kommentarer til dokumentet på en høflig måde, og at mødet forløber i en positiv stemning.

Litteraturliste

/Fagan76/

Michael E. Fagan

Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, no. 3 1976.

Den første omtale af inspektionsteknikken. Artiklen indeholder en praktisk orienteret beskrivelse af, hvordan inspektioner holdes.

/Fagan86/

Michael E. Fagan

Advances in Software Inspections. IEEE Transactions on Software Engineering, vol. SE-12, no. 7, july 1986.

Denne artikel beskriver Fagans erfaringer med inspektionsteknikken gennem de forløbne 10 år.

/Freedman82/

Daniel P. Freedman and Gerald M. Weinberg

Handbook of Walkthroughs, Inspections, and Technical Reviews. Little, Brown and Company. Boston 1982.

Den bedste bog om emnet! Indeholder en grundig gennemgang af såvel reviewteknikken, som de beslægtede teknikker. Udført som "Spørgsmål/svar", med besvarelse af de spørgsmål, forfatterne har erfaring for, der bliver stillet omkring emnet. Meget letlæst.

/Skogstad86/

Søren Skogstad Nielsen

Inspektion i praksis. En metode til fejlfinding i software og andet skriftligt materiale. Teknologisk Institut, Automatiserings-teknik, 1986.

Den første danske undersøgelse over hvordan inspektioner fungerer, og hvilke resultater, der kan opnås. Resultaterne baserer sig på målinger udført på en snes gennemførte inspektioner. En meget instruktiv indføring i inspektionsteknikken.

Craig Larman

Applying UML and Patterns

Prentice Hall PTR, 3.ed

Chapter 9, Domain Models, pp 131 – 159

8.3 Process: Planning the Next Iteration

Planning and project management are important but large topics. A few ideas are briefly presented here, and there are some more tips starting on p. 675.

Organize requirements and iterations by risk, coverage, and criticality.

- **Risk** includes both technical complexity and other factors, such as uncertainty of effort or usability.
- **Coverage** implies that all major parts of the system are at least touched on in early iterations—perhaps a “wide and shallow” implementation across many components.
- **Criticality** refers to functions the client considers of high business value.

These criteria are used to rank work across iterations. Use cases or use case scenarios are ranked for implementation—early iterations implement high ranking scenarios. In addition, some requirements are expressed as high-level features unrelated to a particular use case, such as a logging service. These are also ranked.

The ranking is done before iteration-1, but then again before iteration-2, and so forth, as new requirements and new insights influence the order. That is, the plan of iterations is *adaptive*, rather than speculatively frozen at the beginning of the project. Usually based on some collaborative ranking technique, a grouping of requirements will emerge. For example:

Rank	Requirement (Use Case or Feature)	Comment
High	Process Sale Logging ...	Scores high on all rankings. Pervasive. Hard to add late. ...
Medium	Maintain Users ...	Affects security subdomain. ...
Low

Based on this ranking, we see that some key architecturally significant scenarios of the *Process Sale* use case should be tackled in early iterations. This list is not exhaustive; other requirements will also be tackled. In addition, an implicit or explicit *Start Up* use case will be worked on in each iteration, to meet its initialization needs.

DOMAIN MODELS

It's all very well in practice, but it will never work in theory.

—anonymous management maxim

Objectives

- Identify conceptual classes related to the current iteration.
- Create an initial domain model.
- Model appropriate attributes and associations.

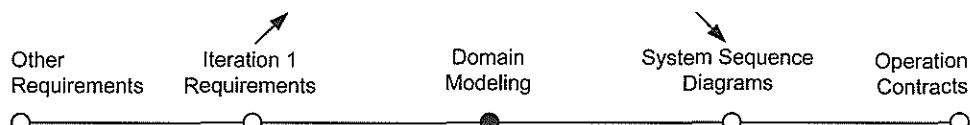
Introduction

A domain model is the most important—and classic—model in OO *analysis*.¹ It illustrates noteworthy concepts in a domain. It can act as a source of inspiration for designing some software objects and will be an input to several artifacts explored in the case studies. This chapter also shows the value of OOA/D knowledge over UML notation; the basic notation is trivial, but there are subtle modeling guidelines for a useful model—expertise can take weeks or months. This chapter explores basic skills in creating domain models.

more advanced
domain modeling
p. 507

What's Next?

Having scoped the work for iteration-1, this chapter explores a partial domain model. The next examines the specific operations upon the system that are implied in the use case scenarios under design for this iteration.



1. Use cases are an important requirements analysis artifact, but are not *object*-oriented. They emphasize an activity view.

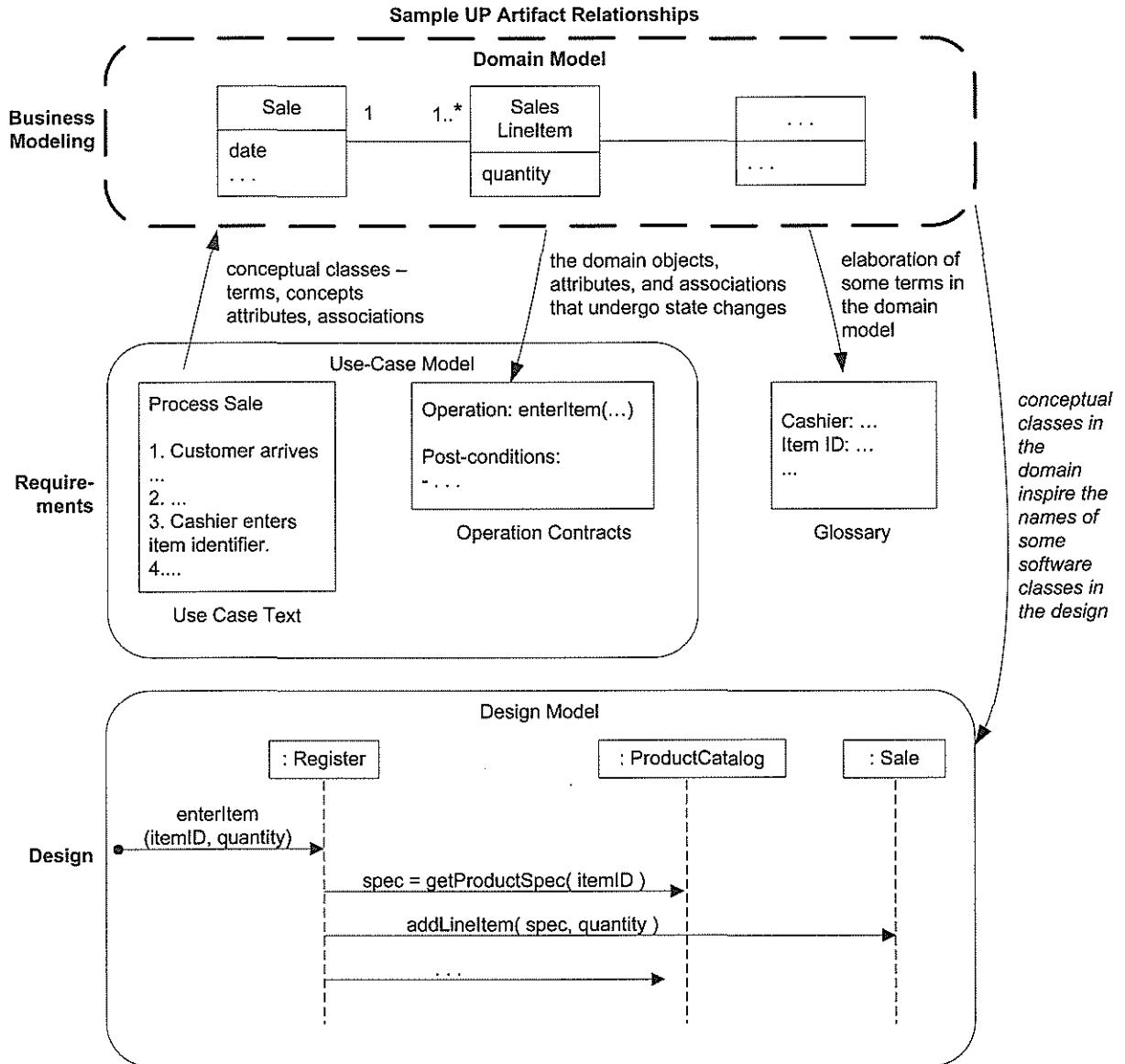


Figure 9.1 Sample UP artifact influence.

domain layer
p. 136

As with all things in an agile modeling and UP spirit, a domain model is optional. UP artifact influence emphasizing a domain model is shown in Figure 9.1. *Bounded* by the use case scenarios under development for the current iteration, the domain model can be evolved to show related noteworthy concepts. The related use case concepts and insight of experts will be input to its creation. The model can in turn influence operation contracts, a glossary, and the Design Model, especially the software objects in the **domain layer** of the Design Model.

9.1

Example

Figure 9.2 shows a partial domain model drawn with UML **class diagram** notation. It illustrates that the **conceptual classes** of *Payment* and *Sale* are significant in this domain, that a *Payment* is related to a *Sale* in a way that is meaningful to note, and that a *Sale* has a date and time, information attributes we care about.

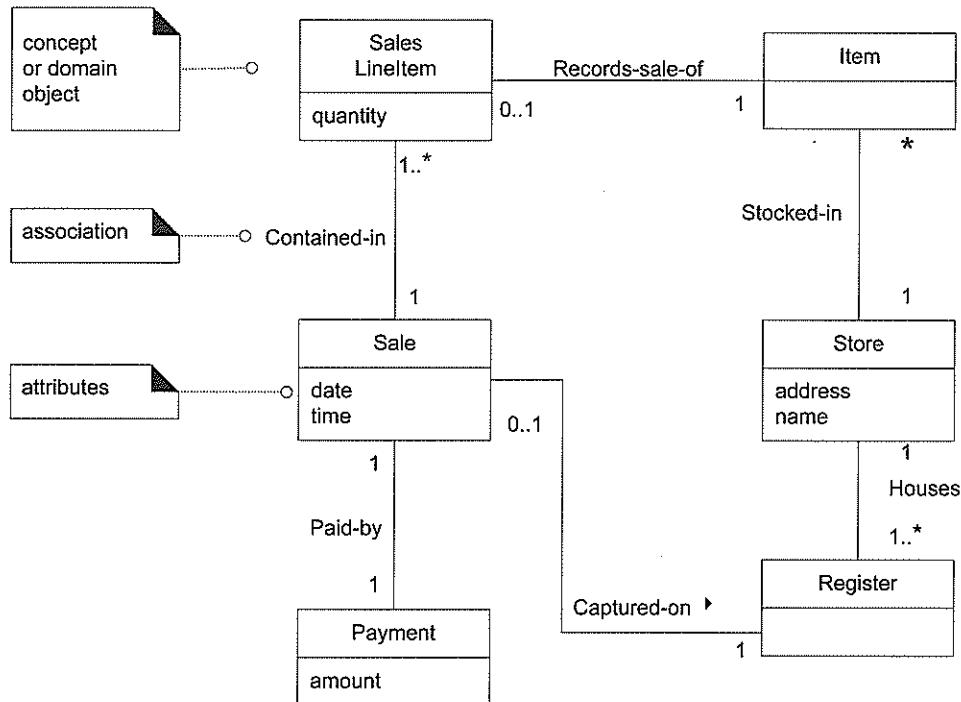


Figure 9.2 Partial domain model—a visual dictionary.

conceptual perspective p. 12

Applying the UML class diagram notation for a domain model yields a *conceptual perspective* model.

Identifying a rich set of conceptual classes is at the heart of OO analysis. If it is done with skill and *short* time investment (say, no more than a few hours in each early iteration), it usually pays off during design, when it supports better understanding and communication.

Guideline

Avoid a waterfall-mindset big-modeling effort to make a thorough or “correct” domain model—it won’t ever be either, and such over-modeling efforts lead to *analysis paralysis*, with little or no return on the investment.

9.2 What is a Domain Model?

The quintessential *object*-oriented analysis step is the decomposition of a domain into noteworthy concepts or objects.

A **domain model** is a *visual* representation of conceptual classes or real-situation objects in a domain [MO95, Fowler96]. Domain models have also been called **conceptual models** (the term used in the first edition of this book), **domain object models**, and **analysis object models**.²

Definition

In the UP, the term “Domain Model” means a representation of real-situation conceptual classes, not of software objects. The term does *not* mean a set of diagrams describing software classes, the domain layer of a software architecture, or software objects with responsibilities.

The UP defines the Domain Model³ as one of the artifacts that may be created in the Business Modeling discipline. More precisely, the UP Domain Model is a specialization of the UP **Business Object Model** (BOM) “focusing on explaining ‘things’ and products important to a business domain” [RUP]. That is, a Domain Model focuses on one domain, such as POS related things. The more broad BOM, not covered in this introductory text and not something I encourage creating (because it can lead to too much up-front modeling), is an expanded, often very large and difficult to create, multi-domain model that covers the *entire* business and all its sub-domains.

Applying UML notation, a domain model is illustrated with a set of **class diagrams** in which no operations (method signatures) are defined. It provides a *conceptual perspective*. It may show:

- domain objects or conceptual classes
- associations between conceptual classes
- attributes of conceptual classes

Definition: Why Call a Domain Model a “Visual Dictionary”?

Please reflect on Figure 9.2 for a moment. See how it visualizes and relates words or concepts in the domain. It also shows an *abstraction* of the conceptual

-
2. They are also related to conceptual entity relationship models, which are capable of showing purely conceptual views of domains, but that have been widely re-interpreted as data models for database design. Domain models are not data models.
 3. Capitalization of “Domain Model” or terms is used to emphasize it as an official model name defined in the UP, versus the general well-known concept of “domain models.”

WHAT IS A DOMAIN MODEL?

classes, because there are many other things one could communicate about registers, sales, and so forth.

The information it illustrates (using UML notation) could alternatively have been expressed in plain text (in the UP Glossary). But it's easy to understand the terms and especially their relationships in a visual language, since our brains are good at understanding visual elements and line connections.

Therefore, the domain model is a *visual dictionary* of the noteworthy abstractions, domain vocabulary, and information content of the domain.

Definition: Is a Domain Model a Picture of Software Business Objects?

A UP Domain Model, as shown in Figure 9.3, is a visualization of things in a real-situation domain of interest, *not* of software objects such as Java or C# classes, or software objects with responsibilities (see Figure 9.4). Therefore, the following elements are not suitable in a domain model:

- Software artifacts, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.
- Responsibilities or methods.⁴

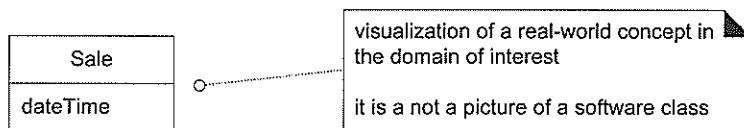


Figure 9.3 A domain model shows real-situation conceptual classes, not software classes.

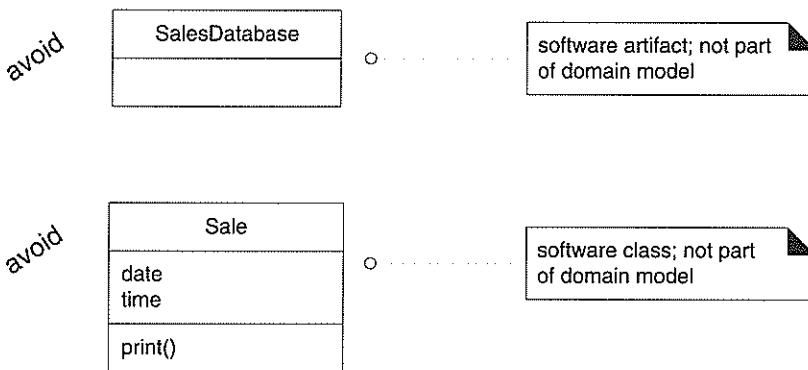


Figure 9.4 A domain model does not show software artifacts or classes.

4. In object modeling, we usually speak of responsibilities related to software objects. And methods are purely a software concept. But, the domain model describes real-situation concepts, not software objects. Considering object responsibilities during *design* work is very important; it is just not part of this model.

Definition: What are Two Traditional Meanings of “Domain Model”?

In the UP and thus this chapter, “Domain Model” is a conceptual perspective of objects in a real situation of the world, not a software perspective. But the term is overloaded; it also has been used (especially in the Smalltalk community where I did most of my early OO development work in the 1980s) to mean “the domain layer of software objects.” That is, the layer of software objects below the presentation or UI layer that is composed of **domain objects**—software objects that represent things in the problem domain space with related “business logic” or “domain logic” methods. For example, a *Board* software class with a *getSquare* method.

Which definition is correct? Well, all of them! The term has long established uses in different communities to mean different things.

I’ve seen lots of confusion generated by people using the term in different ways, without explaining which meaning they intend, and without recognizing that others may be using it differently.

In this book, I’ll usually write **domain layer** to indicate the second software-oriented meaning of domain model, as that’s quite common.

Definition: What are Conceptual Classes?

The domain model illustrates conceptual classes or vocabulary in the domain. Informally, a **conceptual class** is an idea, thing, or object. More formally, a conceptual class may be considered in terms of its symbol, intension, and extension [MO95] (see Figure 9.5).

- **Symbol**—words or images representing a conceptual class.
- **Intension**—the definition of a conceptual class.
- **Extension**—the set of examples to which the conceptual class applies.

For example, consider the conceptual class for the event of a purchase transaction. I may choose to name it by the (English) symbol *Sale*. The intension of a *Sale* may state that it “represents the event of a purchase transaction, and has a date and time.” The extension of *Sale* is all the examples of sales; in other words, the set of all sale instances in the universe.

Definition: Are Domain and Data Models the Same Thing?

A domain model is not a **data model** (which by definition shows persistent data to be stored somewhere), so do not exclude a class simply because the requirements don’t indicate any obvious need to remember information about it (a criterion common in data modeling for relational database design, but not relevant to domain modeling) or because the conceptual class has no attributes. For example, it’s valid to have attributeless conceptual classes, or conceptual classes that have a purely behavioral role in the domain instead of an information role.

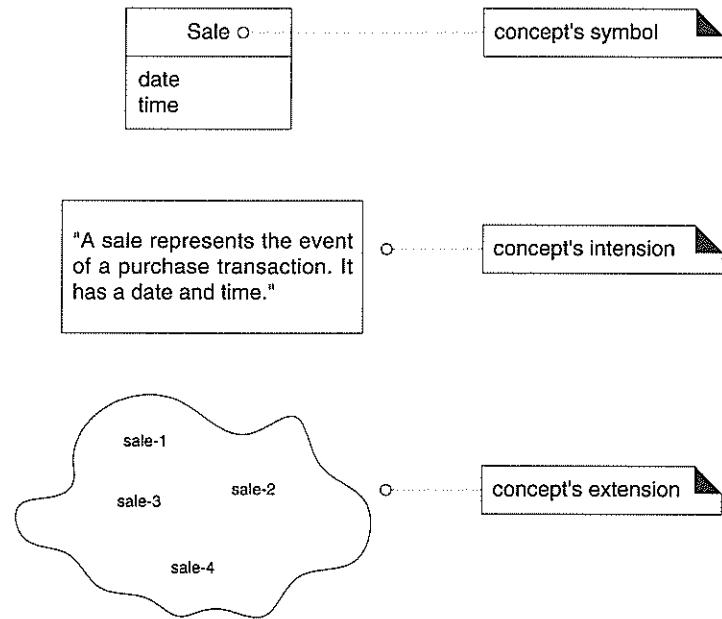


Figure 9.5 A conceptual class has a symbol, intension, and extension.

9.3 Motivation: Why Create a Domain Model?

I'll share a story that I've experienced many times in OO consulting and coaching. In the early 1990s I was working with a group developing a funeral services business system in Smalltalk, in Vancouver (you should see the domain model!). Now, I knew almost nothing about this business, so one reason to create a domain model was so that I could start to understand their key concepts and vocabulary.

*domain layer
p. 206*

We also wanted to create a **domain layer** of Smalltalk objects representing business objects and logic. So, we spent perhaps one hour sketching a UML-ish (actually OMT-ish, whose notation inspired UML) domain model, not worrying about software, but simply identifying the key terms. Then, those terms we sketched in the domain model, such as *Service* (like flowers in the funeral room, or playing “You Can’t Always Get What You Want”), were also used as the names of key software classes in our domain layer implemented in Smalltalk.

This similarity of naming between the domain model and the domain layer (a real “service” and a Smalltalk *Service*) supported a lower gap between the software representation and our mental model of the domain.

Motivation: Lower Representational Gap with OO Modeling

This is a key idea in OO: Use software class names in the domain layer inspired from names in the domain model, with objects having domain-familiar information and responsibilities. Figure 9.6 illustrates the idea. This supports a **low representational gap** between our mental and software models. And that's not just a philosophical nicety—it has a practical time-and-money impact. For example, here's a source-code payroll program written in 1953:

```
100001010100011110101010100010101010101111010101...
```

As computer science people, we know it runs, but the gap between this software representation and our mental model of the payroll domain is huge; that profoundly affects comprehension (and modification) of the software. OO modeling can lower that gap.

Of course, object technology is also of value because it can support the design of elegant, loosely coupled systems that scale and extend easily, as will be explored in the remainder of the book. A lowered representational gap is useful, but arguably secondary to the advantage objects have in supporting ease of change and extension, and managing and hiding complexity.

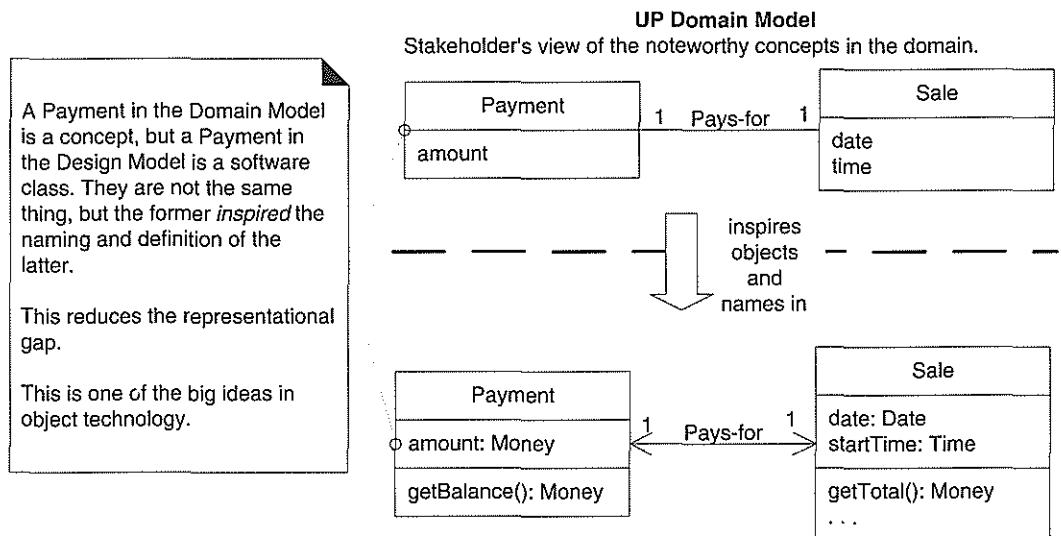


Figure 9.6 Lower representational gap with OO modeling.

9.4 Guideline: How to Create a Domain Model?

Bounded by the current iteration requirements under design:

1. Find the conceptual classes (see a following guideline).
2. Draw them as classes in a UML class diagram.
3. Add associations and attributes. See p. 149 and p. 158.

9.5 Guideline: How to Find Conceptual Classes?

Since a domain model shows conceptual classes, a central question is: How do I find them?

What are Three Strategies to Find Conceptual Classes?

1. Reuse or modify existing models. This is the first, best, and usually easiest approach, and where I will start if I can. There are published, well-crafted domain models and data models (which can be modified into domain models) for many common domains, such as inventory, finance, health, and so forth. Example books that I'll turn to include *Analysis Patterns* by Martin Fowler, *Data Model Patterns* by David Hay, and the *Data Model Resource Book* (volumes 1 and 2) by Len Silverston.
2. Use a category list.
3. Identify noun phrases.

Reusing existing models is excellent, but outside our scope. The second method, using a category list, is also useful.

Method 2: Use a Category List

We can kick-start the creation of a domain model by making a list of candidate conceptual classes. Table 9.1 contains many common categories that are usually worth considering, with an emphasis on business information system needs. The guidelines also suggest some priorities in the analysis. Examples are drawn from the 1) POS, 2) Monopoly, and 3) airline reservation domains.

Conceptual Class Category	Examples
business transactions <i>Guideline:</i> These are critical (they involve money), so start with transactions.	<i>Sale, Payment</i> <i>Reservation</i>
transaction line items <i>Guideline:</i> Transactions often come with related line items, so consider these next.	<i>SalesLineItem</i>
product or service related to a transaction or transaction line item <i>Guideline:</i> Transactions are for something (a product or service). Consider these next.	<i>Item</i> <i>Flight, Seat, Meal</i>
where is the transaction recorded? <i>Guideline:</i> Important.	<i>Register, Ledger</i> <i>FlightManifest</i>
roles of people or organizations related to the transaction; actors in the use case <i>Guideline:</i> We usually need to know about the parties involved in a transaction.	<i>Cashier, Customer, Store</i> <i>MonopolyPlayer</i> <i>Passenger, Airline</i>
place of transaction; place of service	<i>Store</i> <i>Airport, Plane, Seat</i>
noteworthy events, often with a time or place we need to remember	<i>Sale, Payment</i> <i>MonopolyGame</i> <i>Flight</i>
physical objects <i>Guideline:</i> This is especially relevant when creating device-control software, or simulations.	<i>Item, Register</i> <i>Board, Piece, Die</i> <i>Airplane</i>
descriptions of things <i>Guideline:</i> See p. 147 for discussion.	<i>ProductDescription</i> <i>FlightDescription</i>

GUIDELINE: HOW TO FIND CONCEPTUAL CLASSES?

Conceptual Class Category	Examples
catalogs	<i>ProductCatalog</i> <i>FlightCatalog</i>
containers of things (physical or information)	<i>Store, Bin</i> <i>Board</i> <i>Airplane</i>
things in a container	<i>Item</i> <i>Square (in a Board)</i> <i>Passenger</i>
other collaborating systems	<i>CreditAuthorizationSystem</i> <i>AirTrafficControl</i>
records of finance, work, contracts, legal matters	<i>Receipt, Ledger</i> <i>MaintenanceLog</i>
financial instruments	<i>Cash, Check, LineOfCredit</i> <i>TicketCredit</i>
schedules, manuals, documents that are regularly referred to in order to perform work	<i>DailyPriceChangeList</i> <i>RepairSchedule</i>

Table 9.1 Conceptual Class Category List.

Method 3: Finding Conceptual Classes with Noun Phrase Identification

Another useful technique (because of its simplicity) suggested in [Abbot83] is **linguistic analysis**: Identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.⁵

Guideline

Care must be applied with this method; a mechanical noun-to-class mapping isn't possible, and words in natural languages are ambiguous.

5. Linguistic analysis has become more sophisticated; it also goes by the name **natural language modeling**. See [Moreno97] for example.

Nevertheless, linguistic analysis is another source of inspiration. The fully dressed use cases are an excellent description to draw from for this analysis. For example, the current scenario of the *Process Sale* use case can be used.

Main Success Scenario (or Basic Flow):

1. Customer arrives at a POS checkout with goods and/or services to purchase.
 2. Cashier starts a new sale.
 3. Cashier enters item identifier.
 4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
- Cashier repeats steps 2-3 until indicates done.
5. System presents total with taxes calculated.
 6. Cashier tells Customer the total, and asks for payment.
 7. Customer pays and System handles payment.
 8. System logs the completed sale and sends sale and payment information to the external Accounting (for accounting and commissions) and Inventory systems (to update inventory).
 9. System presents receipt.
 10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

...

7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

The domain model is a visualization of noteworthy domain concepts and vocabulary. Where are those terms found? Some are in the use cases. Others are in other documents, or the minds of experts. In any event, use cases are one rich source to mine for noun phrase identification.

Some of these noun phrases are candidate conceptual classes, some may refer to conceptual classes that are ignored in this iteration (for example, “Accounting” and “commissions”), and some may be simply attributes of conceptual classes. See p. 160 for advice on distinguishing between the two.

A weakness of this approach is the imprecision of natural language; different noun phrases may represent the same conceptual class or attribute, among other ambiguities. Nevertheless, it is recommended in combination with the *Conceptual Class Category List* technique.

9.6 Example: Find and Draw Conceptual Classes

*iteration-1
requirements
p. 124*

Case Study: POS Domain

From the category list and noun phrase analysis, a list is generated of candidate conceptual classes for the domain. Since this is a business information system, I'll focus first on the category list guidelines that emphasize business transactions and their relationship with other things. The list is constrained to the requirements and simplifications currently under consideration for iteration-1, the basic cash-only scenario of *Process Sale*.

<i>Sale</i>	<i>Cashier</i>
<i>CashPayment</i>	<i>Customer</i>
<i>SalesLineItem</i>	<i>Store</i>
<i>Item</i>	<i>ProductDescription</i>
<i>Register</i>	<i>ProductCatalog</i>
<i>Ledger</i>	

There is no such thing as a “correct” list. It is a somewhat arbitrary collection of abstractions and domain vocabulary that the modelers consider noteworthy. Nevertheless, by following the identification strategies, different modelers will produce similar lists.

In practice, I don't create a text list first, but immediately draw a UML class diagram of the conceptual classes as we uncover them. See Figure 9.7.

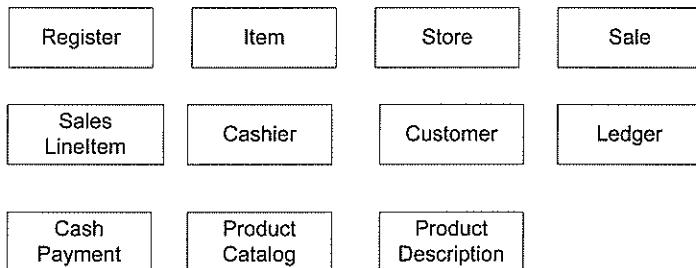


Figure 9.7 Initial POS domain model.

Adding the associations and attributes is covered in later sections.

Case Study: Monopoly Domain

*iteration-1
requirements
p. 124*

From the Category List and noun phrase analysis, I generate a list of candidate conceptual classes for the iteration-1 simplified scenario of *Play a Monopoly Game* (see Figure 9.8). Since this is a simulation, I emphasize the noteworthy tangible, physical objects in the domain.

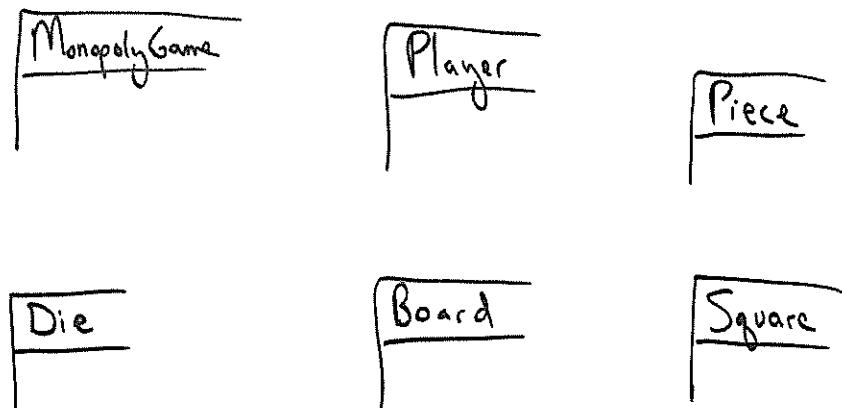


Figure 9.8 Initial Monopoly domain model.

9.7 Guideline: Agile Modeling—Sketching a Class Diagram

Notice the sketching style in the UML class diagram of Figure 9.8—keeping the bottom and right sides of the class boxes open. This makes it easier to grow the classes as we discover new elements. And although I've grouped the class boxes for compactness in this book diagram, on a whiteboard I'll spread them out.

9.8 Guideline: Agile Modeling—Maintain the Model in a Tool?

It's normal to miss significant conceptual classes during early domain modeling, and to discover them later during design sketching or programming. If you are taking an agile modeling approach, the purpose of creating a domain model is to quickly understand and communicate a rough approximation of the key concepts. Perfection is not the goal, and agile models are usually discarded shortly after creation (although if you've used a whiteboard, I recommend taking a digital snapshot). From this viewpoint, there is no motivation to maintain or update the model. But that doesn't mean it's wrong to update the model.

If someone wants the model maintained and updated with new discoveries, that's a good reason to redraw the whiteboard sketch within a UML CASE tool, or to originally do the drawing with a tool and a computer projector (for others to

see the diagram easily). But, ask yourself: Who is going to use the updated model, and why? If there isn’t a practical reason, don’t bother. Often, the evolving *domain layer* of the software hints at most of the noteworthy terms, and a long-life OO analysis domain model doesn’t add value.

9.9

Guideline: Report Objects—Include ‘Receipt’ in the Model?

Receipt is a noteworthy term in the POS domain. But perhaps it’s only a *report* of a sale and payment, and thus duplicate information. Should it be in the domain model?

Here are some factors to consider:

- In general, showing a report of other information in a domain model is not useful since all its information is derived or duplicated from other sources. This is a reason to exclude it.
- On the other hand, it has a special role in terms of the business rules: It usually confers the right to the bearer of the (paper) receipt to return bought items. This is a reason to show it in the model.

Since item returns are not being considered in this iteration, *Receipt* will be excluded. During the iteration that tackles the *Handle Returns* use case, we would be justified to include it.

9.10

Guideline: Think Like a Mapmaker; Use Domain Terms

The mapmaker strategy applies to both maps and domain models.

Guideline

Make a domain model in the spirit of how a cartographer or mapmaker works:

- Use the existing names in the territory. For example, if developing a model for a library, name the customer a “Borrower” or “Patron”—the terms used by the library staff.
- Exclude irrelevant or out-of-scope features. For example, in the Monopoly domain model for iteration-1, cards (such as the “Get out of Jail Free” card) are not used, so don’t show a *Card* in the model this iteration.
- Do not add things that are not there.

The principle is similar to the *Use the Domain Vocabulary* strategy [Coad95].

9.11 Guideline: How to Model the *Unreal* World?

Some software systems are for domains that find very little analogy in natural or business domains; software for telecommunications is an example. Yet it is still possible to create a domain model in these domains. It requires a high degree of abstraction, stepping back from familiar non-OO designs, and listening carefully to the core vocabulary and concepts that domain experts use.

For example, here are candidate conceptual classes related to the domain of a telecommunication switch: *Message*, *Connection*, *Port*, *Dialog*, *Route*, *Protocol*.

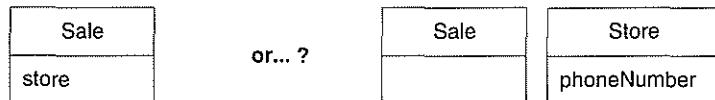
9.12 Guideline: A Common Mistake with Attributes vs. Classes

Perhaps the most common mistake when creating a domain model is to represent something as an attribute when it should have been a conceptual class. A rule of thumb to help prevent this mistake is:

Guideline

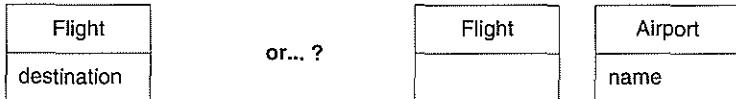
If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

As an example, should *store* be an attribute of *Sale*, or a separate conceptual class *Store*?



In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something that occupies space. Therefore, *Store* should be a conceptual class.

As another example, consider the domain of airline reservations. Should *destination* be an attribute of *Flight*, or a separate conceptual class *Airport*?



In the real world, a destination airport is not considered a number or text—it is a massive thing that occupies space. Therefore, *Airport* should be a concept.

9.13 Guideline: When to Model with 'Description' Classes?

A **description class** contains information that describes something else. For example, a *ProductDescription* that records the price, picture, and text description of an *Item*. This was first named the *Item-Descriptor* pattern in [Coad92].

Motivation: Why Use 'Description' Classes?

The following discussion may at first seem related to a rare, highly specialized issue. However, it turns out that the need for description classes is common in many domain models.

Assume the following:

- An *Item* instance represents a physical item in a store; as such, it may even have a serial number.
- An *Item* has a description, price, and itemID, which are not recorded anywhere else.
- Everyone working in the store has amnesia.
- Every time a real physical item is sold, a corresponding software instance of *Item* is deleted from "software land."

With these assumptions, what happens in the following scenario?

There is strong demand for the popular new vegetarian burger—ObjectBurger. The store sells out, implying that all *Item* instances of ObjectBurgers are deleted from computer memory.

Now, here is one problem: If someone asks, "How much do ObjectBurgers cost?", no one can answer, because the memory of their price was attached to inventoried instances, which were deleted as they were sold.

Here are some related problems: The model, if implemented in software similar to the domain model, has duplicate data, is space-inefficient, and error-prone (due to replicated information) because the description, price, and itemID are duplicated for every *Item* instance of the same product.

The preceding problem illustrates the need for objects that are *descriptions* (sometimes called *specifications*) of other things. To solve the *Item* problem, what is needed is a *ProductDescription* class that records information about items. A *ProductDescription* does not represent an *Item*, it represents a description of information *about* items. See Figure 9.9.

A particular *Item* may have a serial number; it represents a physical instance. A *ProductDescription* wouldn't have a serial number.

Switching from a conceptual to a software perspective, note that even if all inventoried items are sold and their corresponding *Item* software instances are deleted, the *ProductDescription* still remains.

The need for description classes is common in sales, product, and service domains. It is also common in manufacturing, which requires a *description* of a manufactured thing that is distinct from the thing itself.

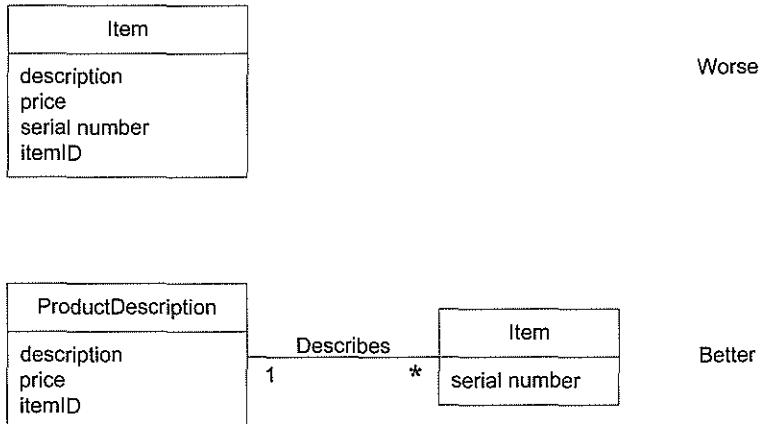


Figure 9.9 Descriptions about other things. The * means a multiplicity of “many.” It indicates that one *ProductDescription* may describe many (*) *Items*.

Guideline: When Are Description Classes Useful?

Guideline

Add a description class (for example, *ProductDescription*) when:

- There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
- Deleting instances of things they describe (for example, *Item*) results in a loss of information that needs to be maintained, but was incorrectly associated with the deleted thing.
- It reduces redundant or duplicated information.

Example: Descriptions in the Airline Domain

As another example, consider an airline company that suffers a fatal crash of one of its planes. Assume that all the flights are cancelled for six months pending completion of an investigation. Also assume that when flights are cancelled, their corresponding *Flight* software objects are deleted from computer memory. Therefore, after the crash, all *Flight* software objects are deleted.

If the only record of what airport a flight goes to is in the *Flight* software instances, which represent specific flights for a particular date and time, then there is no longer a record of what flight routes the airline has.

The problem can be solved, both from a purely conceptual perspective in a domain model and from a software perspective in the software designs, with a *FlightDescription* that describes a flight and its route, even when a particular flight is not scheduled (see Figure 9.10).

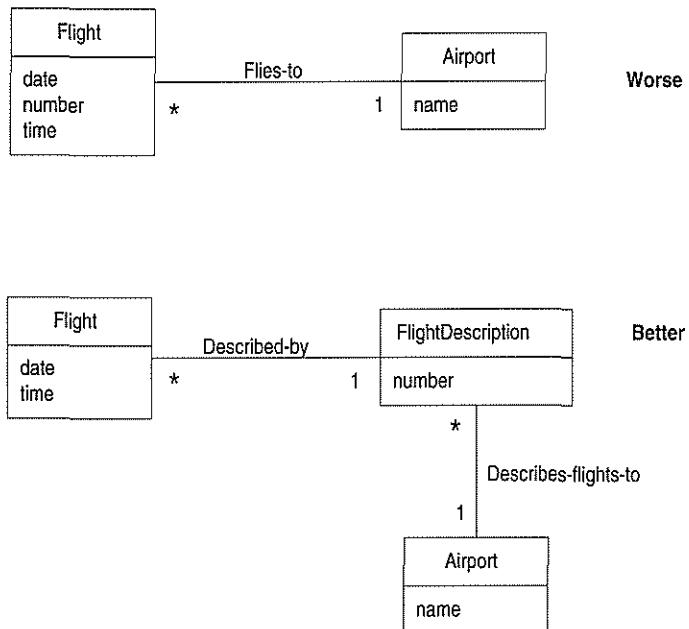


Figure 9.10 Descriptions about other things.

Note that the prior example is about a service (a flight) rather than a good (such as a veggieburger). Descriptions of services or service plans are commonly needed.

As another example, a mobile phone company sells packages such as “bronze,” “gold,” and so forth. It is necessary to have the concept of a description of the package (a kind of service plan describing rates per minute, wireless Internet content, the cost, and so forth) separate from the concept of an actual sold package (such as “gold package sold to Craig Larman on Jan. 1, 2047 at \$55 per month”). Marketing needs to define and record this service plan or *MobileCommunicationsPackageDescription* before any are sold.

9.14 Associations

It's useful to find and show associations that are needed to satisfy the information requirements of the current scenarios under development, and which aid in

understanding the domain.

An **association** is a relationship between classes (more precisely, instances of those classes) that indicates some meaningful and interesting connection (see Figure 9.11).

In the UML, associations are defined as “the semantic relationship between two or more classifiers that involve connections among their instances.”

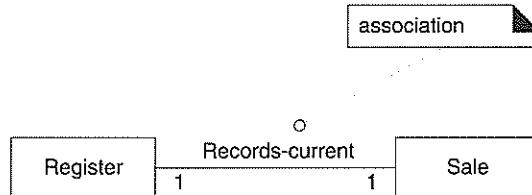


Figure 9.11 Associations.

Guideline: When to Show an Association?

Associations worth noting usually imply knowledge of a relationship that needs to be preserved for some duration—it could be milliseconds or years, depending on context. *In other words, between what objects do we need some memory of a relationship?*

For example, do we need to *remember* what *SalesLineItem* instances are associated with a *Sale* instance? Definitely, otherwise it would not be possible to reconstruct a sale, print a receipt, or calculate a sale total.

And we need to remember completed *Sales* in a *Ledger*, for accounting and legal purposes.

Because the domain model is a conceptual perspective, these statements about the need to remember refer to a need in a real situation of the world, not a software need, although during implementation many of the same needs will arise.

In the monopoly domain, we need to remember what *Square* a *Piece* (or *Player*) is on—the game doesn’t work if that isn’t remembered. Likewise, we need to remember what *Piece* is owned by a particular *Player*. We need to remember what *Squares* are part of a particular *Board*.

But on the other hand, there is no need to remember that the *Die* (or the plural, “dice”) total indicates the *Square* to move to. It’s true, but we don’t need to have an ongoing memory of that fact, after the move has been made. Likewise, a *Cashier* may look up *ProductDescriptions*, but there is no need to remember the fact of a particular *Cashier* looking up particular *ProductDescriptions*.

Guideline

Consider including the following associations in a domain model:

- Associations for which knowledge of the relationship needs to be preserved for some duration (“need-to-remember” associations).
- Associations derived from the Common Associations List.

Guideline: Why Should We Avoid Adding Many Associations?

We need to avoid adding too many associations to a domain model. Digging back into our discrete mathematics studies, you may recall that in a graph with n nodes, there can be $(n \cdot (n-1))/2$ associations to other nodes—a potentially very large number. A domain model with 20 classes could have 190 associations lines! Many lines on the diagram will obscure it with “visual noise.” Therefore, be parsimonious about adding association lines. Use the criterion guidelines suggested in this chapter, and focus on “need-to-remember” associations.

Perspectives: Will the Associations Be Implemented In Software?

During domain modeling, an association is *not* a statement about data flows, database foreign key relationships, instance variables, or object connections in a software solution; it is a statement that a relationship is meaningful in a purely conceptual perspective—in the real domain.

That said, many of these relationships *will* be implemented in software as paths of navigation and visibility (both in the Design Model and Data Model). But the domain model is not a data model; associations are added to highlight our rough understanding of noteworthy relationships, not to document object or data structures.

Applying UML: Association Notation

An association is represented as a line between classes with a capitalized association name. See Figure 9.12.

The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.

The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible. This traversal is purely abstract; it is *not* a statement about connections between software entities.

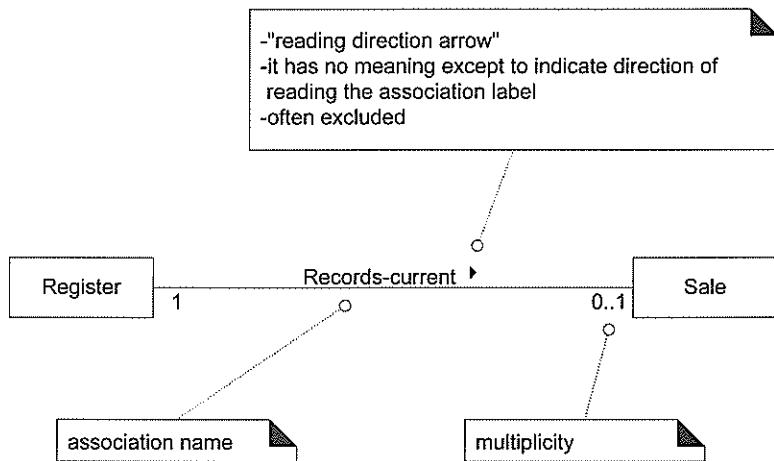


Figure 9.12 The UML notation for associations.

An optional “reading direction arrow” indicates the direction to read the association name; it does not indicate direction of visibility or navigation. If the arrow is not present, the convention is to read the association from left to right or top to bottom, although the UML does not make this a rule (see Figure 9.12).

Caution

The reading direction arrow has no meaning in terms of the model; it is only an aid to the reader of the diagram.

Guideline: How to Name an Association in UML?

Guideline

Name an association based on a *ClassName-VerbPhrase-ClassName* format where the verb phrase creates a sequence that is readable and meaningful.

Simple association names such as “Has” or “Uses” are usually poor, as they seldom enhance our understanding of the domain.

For example,

- *Sale Paid-by CashPayment*
 - bad example (doesn’t enhance meaning): *Sale Uses CashPayment*
- *Player Is-on Square*
 - bad example (doesn’t enhance meaning): *Player Has Square*

Association names should start with a capital letter, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter. Two common and equally legal formats for a compound association name are:

- *Records-current*
- *RecordsCurrent*

Applying UML: Roles

Each end of an association is called a **role**. Roles may optionally have:

- multiplicity expression
- name
- navigability

Multiplicity is examined next.

Applying UML: Multiplicity

Multiplicity defines how many instances of a class *A* can be associated with one instance of a class *B* (see Figure 9.13).

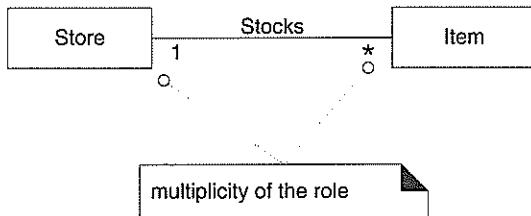


Figure 9.13 Multiplicity on an association.

For example, a single instance of a *Store* can be associated with “many” (zero or more, indicated by the *) *Item* instances.

Some examples of multiplicity expressions are shown in Figure 9.14.

The multiplicity value communicates how many instances can be validly associated with another, at a particular moment, rather than over a span of time. For example, it is possible that a used car could be repeatedly sold back to used car dealers over time. But at any particular moment, the car is only *Stocked-by one* dealer. The car is not *Stocked-by many* dealers at any particular moment. Similarly, in countries with monogamy laws, a person can be *Married-to* only *one* other person at any particular moment, even though over a span of time, that same person may be married to *many* persons.

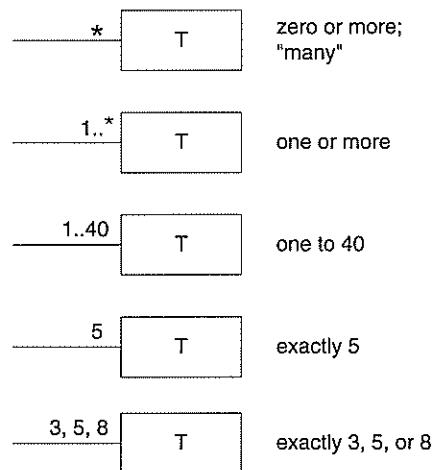


Figure 9.14 Multiplicity values.

The multiplicity value is dependent on our interest as a modeler and software developer, because it communicates a domain constraint that will be (or could be) reflected in software. See Figure 9.15 for an example and explanation.



Multiplicity should "1" or "0..1"?

The answer depends on our interest in using the model. Typically and practically, the multiplicity communicates a domain constraint that we care about being able to check in software, if this relationship was implemented or reflected in software objects or a database. For example, a particular item may become sold or discarded, and thus no longer stocked in the store. From this viewpoint, "0..1" is logical, but ...

Do we care about that viewpoint? If this relationship was implemented in software, we would probably want to ensure that an *Item* software instance would always be related to 1 particular *Store* instance, otherwise it indicates a fault or corruption in the software elements or data.

This partial domain model does not represent software objects, but the multiplicities record constraints whose practical value is usually related to our interest in building software or databases (that reflect our real-world domain) with validity checks. From this viewpoint, "1" may be the desired value.

Figure 9.15 Multiplicity is context dependent.

Rumbaugh gives another example of *Person* and *Company* in the *Works-for* association [Rumbaugh91]. Indicating if a *Person* instance works for one or many *Company* instances is dependent on the context of the model; the tax depart-

ment is interested in *many*; a union probably only *one*. The choice usually depends on why we are building the software.

Applying UML: Multiple Associations Between Two Classes

Two classes may have multiple associations between them in a UML class diagram; this is not uncommon. There is no outstanding example in the POS or Monopoly case study, but an example from the domain of the airline is the relationships between a *Flight* (or perhaps more precisely, a *FlightLeg*) and an *Airport* (see Figure 9.16); the flying-to and flying-from associations are distinctly different relationships, which should be shown separately.

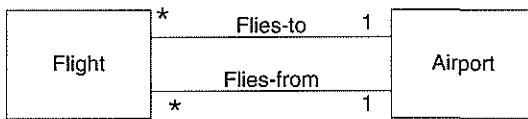


Figure 9.16 Multiple associations.

Guideline: How to Find Associations with a Common Associations List

Start the addition of associations by using the list in Table 9.2. It contains common categories that are worth considering, especially for business information systems. Examples are drawn from the 1) POS, 2) Monopoly, and 3) airline reservation domains.

Category	Examples
A is a transaction related to another transaction B	<i>CashPayment—Sale</i> <i>Cancellation—Reservation</i>
A is a line item of a transaction B	<i>SalesLineItem—Sale</i>
A is a product or service for a transaction (or line item) B	<i>Item—SalesLineItem (or Sale)</i> <i>Flight—Reservation</i>
A is a role related to a transaction B	<i>Customer—Payment</i> <i>Passenger—Ticket</i>
A is a physical or logical part of B	<i>Drawer—Register</i> <i>Square—Board</i> <i>Seat—Airplane</i>

Category	Examples
A is physically or logically contained in/on B	Register—Store, Item—Shelf Square—Board Passenger—Airplane
A is a description for B	ProductDescription—Item FlightDescription—Flight
A is known/logged/recorded/reported/captured in B	Sale—Register Piece—Square Reservation—FlightManifest
A is a member of B	Cashier—Store Player—MonopolyGame Pilot—Airline
A is an organizational subunit of B	Department—Store Maintenance—Airline
A uses or manages or owns B	Cashier—Register Player—Piece Pilot—Airplane
A is next to B	SalesLineItem—SalesLineItem Square—Square City—City

Table 9.2 Common Associations List.

9.15 Example: Associations in the Domain Models

Case Study: NextGen POS

The domain model in Figure 9.17 shows a set of conceptual classes and associations that are candidates for our POS domain model. The associations are primarily derived from the “need-to-remember” criteria of this iteration requirements, and the Common Association List. Reading the list and mapping the examples to the diagram should explain the choices. For example:

- **Transactions related to another transaction**—Sale Paid-by CashPayment.
- **Line items of a transaction**—Sale Contains SalesLineItem.

EXAMPLE: ASSOCIATIONS IN THE DOMAIN MODELS

- **Product for a transaction (or line item)—*SalesLineItem Records-sale-of Item*.**

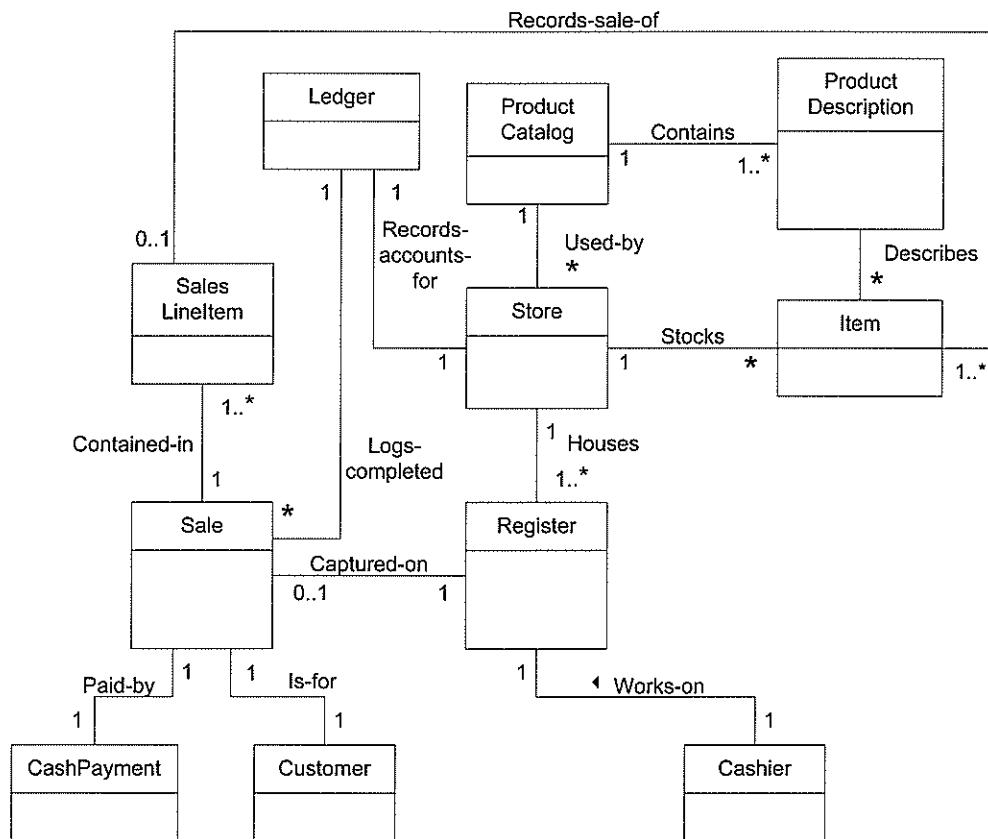


Figure 9.17 NextGen POS partial domain model.

Case Study: Monopoly

See Figure 9.18. Again, the associations are primarily derived from the “need-to-remember” criteria of this iteration requirements, and the Common Association List. For example:

- **A is contained in or on B—Board Contains Square.**
- **A owns B—Players Owns Piece.**
- **A is known in/on B—Piece Is-on Square.**
- **A is member of B—Player Member-of (or Plays) MonopolyGame.**

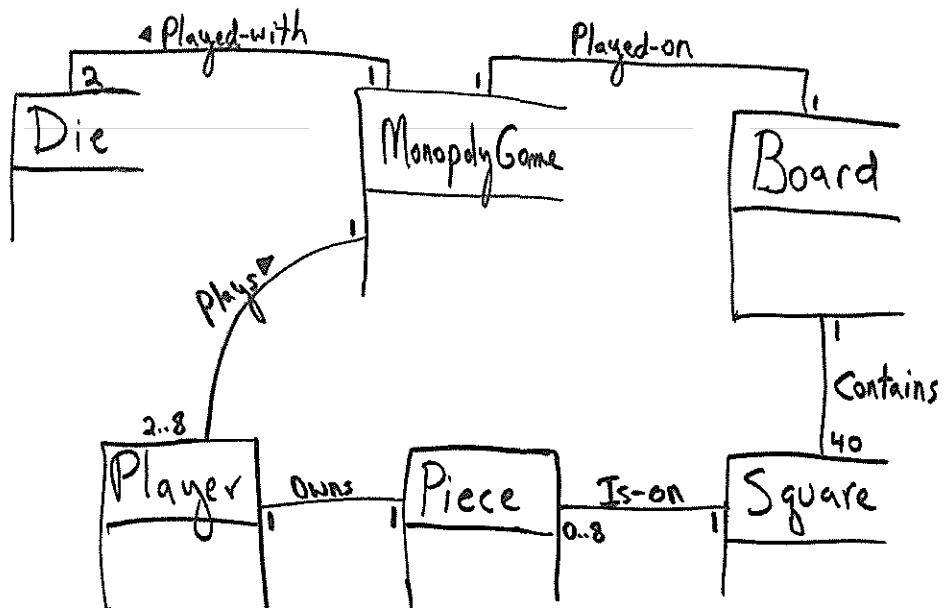


Figure 9.18 Monopoly partial domain model.

9.16 Attributes

It is useful to identify those attributes of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development. An **attribute** is a logical data value of an object.

Guideline: When to Show Attributes?

Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information.

For example, a receipt (which reports the information of a sale) in the *Process Sale* use case normally includes a date and time, the store name and address, and the cashier ID, among many other things.

Therefore,

- *Sale* needs a *dateTime* attribute.
- *Store* needs a *name* and *address*.
- *Cashier* needs an *ID*.

Applying UML: Attribute Notation

Attributes are shown in the second compartment of the class box (see Figure 9.19). Their type and other information may optionally be shown.

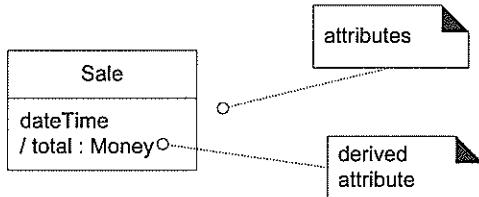


Figure 9.19 Class and attributes.

More Notation

detailed UML class diagram notation p. 249, and also on the back inside cover of the book

The full syntax for an attribute in the UML is:

visibility name : type multiplicity = default {property-string}

Some common examples are shown in Figure 9.20.

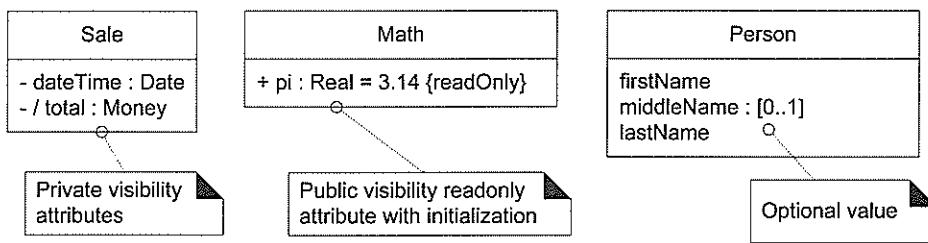


Figure 9.20 Attribute notation in UML.

As a convention, most modelers will assume attributes have private visibility (-) unless shown otherwise, so I don't usually draw an explicit visibility symbol.

{*readOnly*} is probably the most common property string for attributes.

Multiplicity can be used to indicate the optional presence of a value, or the number of objects that can fill a (collection) attribute. For example, many domains require that a first and last name be known for a person, but that a middle name is optional. The expression *middleName : [0..1]* indicates an optional value—0 or 1 values are present.

Guideline: Where to Record Attribute Requirements?

Notice that, subtly, *middleName : [0..1]* is a requirement or domain rule, embedded in the domain model. Although this is just a conceptual-perspective domain model, it probably implies that the software perspective should allow a missing

James K. Peckol

Embedded Systems Design, A Contemporary Design Tool

Wiley, ISBN 978-0-471-72180-2.

Chapter 9, System Design, pp 366 - 368, pp 376 - 391

The time and frequency measurements will be implemented to provide three user selectable resolution ranges: high frequency range/shorter duration signals, a second for midrange frequency/midrange duration signals, and a third for low frequency/longer duration signals. The events measurement capability will support two selectable counting durations, shorter and longer.

For frequency, period, and events measurements, the user will be able to select either a positive or negative edge trigger. For interval measurements, the user will be able to select the polarity of the start and stop signals independently.

Operating Specifications

The system shall operate in a standard commercial / industrial environment

Temperature Range 0–85°C

Humidity up to 90% RH noncondensing

Power 120–240 VAC 50 Hz, 60 Hz, 400 Hz, 15 VDC

The system shall operate for a minimum of 8 hours on a fully charged battery

The system time base shall meet the following specifications.

Temperature stability 0–50 °C

$< 6 \times 10^{-6}$

Aging Rate

90 day $< 3 \times 10^{-8}$

6 month $< 6 \times 10^{-7}$

1 year $< 25 \times 10^{-6}$

Reliability and Safety Specification

The counter shall comply with the appropriate standards

Safety: UL-3111-1, IEC-1010, CSA 1010.1

EMC: CISPR-11, IEC 801-2, -3, -4, EN50082-1

MTBF: Minimum of 10,000 hours

9.7 THE SYSTEM DESIGN SPECIFICATION

System Design Specification, System Requirements Specification

The *System Design Specification* is based on the *System Requirements Specification* and specifies the *how* of the design, not the *what*. The specification is written in the designer's language and from the designer's point of view. It serves as a bridge between the customer and the designer, as we see in Figure 9.19.

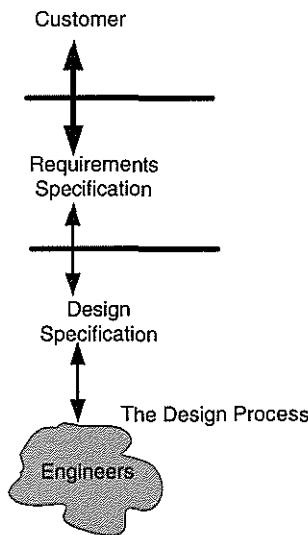


Figure 9.19 The Customer, the Requirements, the Design, and the Engineer

Requirements Specification Design Specification

Whereas the *Requirements Specification* provides a view from the outside of the system looking in, the *Design Specification* provides a view from the inside looking out as well. Notice also that the *Design Specification* has two masters:

Requirements Specification *Design Specification*

- It must specify the system's public interface from inside the system.
- It must specify *how* the requirements defined for and by the public interface are to be met by the internal functions of the system.

We have seen that the *Requirements Specification* is written in less formal terms with the intent of capturing the customer's view of the product. The *Design Specification* must formalize those requirements in precise, unambiguous language. Putting the inevitable changes that occur during the lifetime of any project aside for the moment, we find that the design specification should be sufficiently clear, robust, and complete that a group of engineers could develop the product without ever talking to the author of the specification.

Design Note

A good litmus test of the viability of a design specification is the question, "If I send this to my colleague (who is working for one of our subcontractors), will he or she understand this?" If the answer is no, the specification should be reexamined.

9.7.1 The System

As part of formalizing and quantifying the system's requirements, one must attach concrete numbers, tolerances, and constraints to all of the system's input and output signals. All timing relationships must be defined. The system's functional and operational behaviors are described in detail.

9.7.2 Quantifying the System

The quantification of the system's characteristics begins with the inputs and outputs, based on the specified requirements. The necessary technical details are added to enable the engineer to accurately and faithfully execute the actual design.

- *System Inputs and Outputs*

For each I/O variable, the following are specified.

- The name of the signal
- The use of the signal as an input or output
- The nature of the signal as an event, data, state variable, and so on.

Starting with the requirements specification, we provide detailed descriptions as necessary and incorporate any additional technical or technological constraints that may be needed.

- The complete specification of the signal, including nominal value, range, level tolerances, timing, and timing tolerances
- The interrelationships with other signals, including any constraints on those relationships

- *Responsibilities—Activities*

- Functional and Operational Specifications

The functional and operational specifications that will quantify the dynamic behavior of the system are now formulated. The functional requirements specification identifies the major functions that the system must perform from a high-level view. The operational specification endeavors to capture specific details of how those functions behave within the context of the operating environment.

The manner in which a particular function must operate, the conditions imposed on the operation, and the range of that operation are now captured. The specification must consider concrete numbers—precisions and tolerances.

All variables in the functional specification, all operating conditions, and ordinary and extraordinary operating modes must be quantified. The specification may include domain-specific knowledge that is proprietary or heuristically known to the customer. Such knowledge can be very important to the design.

In stating the specific design requirements for the system, one can use table equations or algorithms, formal design language, or pseudo code, flow diagram or detailed UML diagrams such as state charts, sequence diagrams, and timing lines. Schematics, codes, or parts lists are not included, except in limited circumstances.

- Technological (and Other) Specifications

The technological portion includes all detailed and concrete specifications that are relevant to the design of the system hardware and software. Five areas that should be considered can easily be identified.

1. Geographical constraints

Distributed applications can span a single room, can expand to include a complete factory, or can encompass several countries. Consequently, one must address both the technical items such as interconnection topologies, communications methods, restrictions on usage, and environmental contamination as well as nontechnical matters such as costs associated with the physical medium and its installation.

2. Characterization of and constraints on interface signals

The assumption is made that signals between the system and the external world are electrical, optical, or wireless or that they can be converted into or from such a form. The necessary physical characterization of each is obviously going to depend on the type of signal. That is, an electrical signal is specified differently from an optical signal.

Since many of the interface signals may be driven by the external environment, potentially they are beyond the designer's control. Therefore, it is important to gain as much information about them as possible.

3. User interface requirements

If the system interfaces to such external world devices as medical or instrumentation equipment, how information is presented and whether any relevant and associated protocols exist must be considered. There may also be standards that govern how such information must be presented.

Consider the significant risk that would arise if each avionics vendor presented critical flight information and controls to the aircraft pilot in a different way. The near disaster at Three Mile Island in 1979 arose, in part, because of the confusion caused by too much information.

4. Temporal constraints

The system may have to perform under hard or soft real-time constraints. Such constraints may specify delays on signals originating from external entities' responses to system outputs by external entities, and/or internal system delays.

5. Electrical Infrastructure considerations

There must be a specification for the electrical characteristics of any electrical infrastructure. Included in this portion of the specification are power consumption

tion, necessary power supplies, tolerances and capacities of such supplies, tolerance to degraded power, and power management schemes.

- *Safety and Reliability*

In formulating the design requirements for the safety and reliability of the system, the focus shifts to the detailed objectives of each and to the strategy for achieving those goals.

Safety considerations should address

- Understanding and specifying any environmental and safety issues

The reliability specification should include

- Requirements for diagnostic tests, remote maintenance, remote upgrade, and their details
- Concrete numbers for MTTF and MTBF of any built-in self-test circuitry
- Concrete numbers for MTTF and MTBF of the system itself
- Consideration of system performance under partial or full failure

Let's now bring everything together.

EXAMPLE 9.0

Designing a Counter (Cont.)

Design Specification Requirements Specification

Quantifying the specification

We will now continue with the development of the counter. The system *Design Specification* will follow, but extend, what has been captured in the *Requirements Specification*. The focus will now be on providing specific numbers, ranges, and tolerances for signals that are within the system.

Once again, we will put together any thoughts about the environment and the system prior to writing the specification.

Environment

Specifications relating to the environment have been discussed earlier. There are no changes here.

Counter

- When specifying measurement and stimulus equipment, the specifications for that equipment are generally 10 times (one order of magnitude) better than those for the signals that must be measured or generated.
- That margin is provided when specifying the range and tolerances on the counter's measurement capabilities.
- Specifications on counting events are based on the granularity of the timing of the interval during which the events are counted.
- The values to be displayed at the measurement boundaries are now defined.

The next step is to provide any additional detail that may be needed and to fully quantify the counter specifications.

System Design Specification for a Digital Counter

System Description

This specification describes and defines the basic requirements for a digital counter. The counter is to be able to measure frequency, period, time interval, and events. The system supports three measurement ranges for each signal and two for events. The counter is to be manually operated with the ability to support remote operation in future. The counter is

to be low cost and flexible so that it may be utilized in a variety of applications.

Specification of External Environment

The counter is to operate in an industrial environment in a commercial grade temperature and lighting environment. The unit will support either line power or battery operation. Specific details are included under Operating Specifications.

A specification is a precise description of the system that meets stated requirements. Ideally, a specification document should be

- Complete
- Consistent
- Comprehensible
- Traceable to the requirements
- Unambiguous
- Modifiable
- Able to be written

System Specification

The specification should be expressed in as formal a language or notation as possible, yet readable. Ideally, it should also be executable. A *System Specification* should focus precisely on the system itself. It should provide a complete description of its externally visible characteristics, that is, its public interface. External visibility clearly separates those aspects that are functionally visible to the environment in which the system operates from those aspects of the system that reflect its internal structure.

9.9 PARTITIONING AND DECOMPOSING A SYSTEM

System Design Specification

At this point in the design cycle, all of the system requirements have been identified, captured, and formalized into the *System Design Specification*. The next step is to move inside the system and begin the process of specifying and designing the functionality that gives rise to the external behavior.

Throughout all of the previous discussions, modularity and encapsulation have been repeatedly stressed. We will look first at why such an approach is recommended and then at what should be considered as the process of decomposing and ultimately partitioning the system into hardware and software modules proceeds.

9.9.1 Initial Thoughts

So, let's get to the first question, "Why do we do this?" Reuse is one important reason. With each new design, one should always look to the previous project as well as the next one. What can be used from the last project to expedite the development of this one? How can the current design be implemented to support a future feature? Can parts of this design be used in future projects?

Second, many compilers generate object code in segments, one for each module. Such actions may place size restrictions on the individual modules. Poor module builds can significantly affect memory accesses, increase cache misses, promote thrashing, and significantly reduce performance.

Third, often, work assignments are made on a module-by-module basis. Module boundaries should be defined so as to minimize interfaces among different parts of the system. Such a practice simplifies the process of subcontracting some of the work as well. Security issues also play a role when subcontracting is considered. Whether working for a toy company or on a sensitive government project, one needs to consider what information to make available to outside vendors. By properly decomposing a system, the portions that can be outsourced and those for which control over should be retained can be more easily identified.

progressively refines

Fourth, the modules should be packaged with the goal of stabilizing the module interfaces during the early part of the design.

Fifth, partitioning the system into well-defined, loosely coupled modules helps to ensure a safe and robust design. Such an approach helps to prevent a failure in one part of the system from propagating into and affecting another.

The importance of partitioning a new design should be evident; the next step is to examine the process for doing so. The process starts with the top-level system and then *progressively refines* that model into smaller and more manageable pieces that can more easily be designed and built.

Initially, the focus is on a functional view of the system rather than on specific pieces of hardware and software. It is important first to understand and to capture the behavior at a high level. The next step is to map those functions, that functionality, onto the hardware and software as necessary to satisfy the constraints identified during the initial phases of the design. Partitioning is important during the early stages of the development of the system first as an aid in attacking the complexities of a large system and later as a guide in arriving at a sound physical architecture.

As we begin to think about organizing the system into the collection of pieces that will implement the customer's requirements, one should look at the problem from both a high-level view *and* a more detailed view. It is important to remember that developing a partition is not a one-time process; it is not necessary to be perfect the first time. The partitioning process will probably need to be done several times before a satisfactory and workable decomposition is achieved.

Prior to beginning the system partition, keep some general thoughts in mind.

1. Remember that with every rule or guideline, there must always be room for exceptions.
2. Each module should solve one well-defined piece of the problem.
3. Mixing functionality across modules makes all aspects of the development and support process much more difficult. By doing so, one can easily create noodle hardware and spaghetti code. Future changes to such modules will be very difficult to implement and can easily lead to unexpected side effects and unrelated pieces of the system suddenly not working.

Although it is desirable to have well-defined modules, with simple interfaces, that solve nicely encapsulated pieces of the problem, in embedded applications sometimes one is not able to do so because of performance or economic constraints.

The system should be partitioned so that the intended functionality of each module is easy to understand.

If other parties can understand the design, then they will be able to maintain it and to extend it as necessary throughout the product's lifetime. Remember, over half of the engineers involved in embedded systems design do not do new designs; they maintain and enhance existing designs.

During development, easy to understand designs will lead to fewer surprises as the design nears completion. All interested parties should be able to follow the design and comment as the process unwinds. A design that is too complex quickly discourages early criticism. People will not take the time to learn what the system is to do. Unfortunately, such early acceptance often is replaced by later rejection and potentially major redesign efforts. Although it is important to be proud of one's work, one should always seek out other constructive ideas.

4. Partitioning should be done so that connections between modules are only introduced because of connections between pieces of problem.

One should not put a piece of functionality into a module just because there is nowhere else for it to go.

5. Partitioning should ensure that connections between modules are as independent as possible.
6. Once again, like things should be kept together. Such a practice helps to reduce errors. Partitioning is also done to help meet the economic goals of the design.

When forming partitions, the process must be considered from a number of viewpoints. Taking only a single point of view or neglecting any one can have significant long-term effects. At the end of the day, the system may meet neither the customer's expectations nor the performance specifications.

functional coupling, cohesiveness
loosely coupled, highly cohesive modules

As the decomposition process proceeds, the design should first be considered from a *functional* point of view. The outcome from the decomposition steps is a functional model and that can be used to define the system architecture. Among the many things that should be considered, two that should appear early in the process are the *coupling* and the *cohesiveness* of the modules into which the system is being decomposed. The goal is to develop *loosely coupled, highly cohesive modules*. Let's see what these mean.

9.9.2 Coupling

coupling

Coupling is a heuristic that provides an estimate of how interdependent the modules are. Tightly coupled modules will generally utilize shared data or interchange control information. As module interdependence increases, so does the complexity of managing those modules, and the more difficulty one will have in

- Debugging the design during development
- Troubleshooting the system in the event of field failures
- Maintaining the modules and system
- Modifying the design to add features or capabilities

The major goal is to make the system's modules as independent as possible and to reduce or minimize coupling.

Design Heuristic The lower the coupling, the better job that has been done during partitioning.

During the early stages of the design, think about the following to help to reduce coupling:

1. Eliminate all unessential interaction between modules.
If a particular piece of functionality or shared parameter is not part of the intended task of two modules, then eliminate it.
2. Minimize the amount of essential interaction between modules.
While this sounds the same as the previous point, it is not. If an early analysis establishes that some interaction with another module is necessary, effort should be made to reduce the complexity of that required interface. The goal is to keep things simple.

Some of the ways to help to reduce complexity include:

- a. Reduce the number of interconnections between modules and thereby reduce the number of pieces of data that must flow between modules.
- b. Try to take the most direct route to a signal or piece of data as appropriate.
In some cases the best implementation is to use a proxy as an interface to a signal or piece of data. In general, however, it is best to reduce the number of modules involved.

- c. In general, avoid using shared global variables. A better method is to pass data into a module via its parameter list or calling interface.
With embedded applications, however, at times such sharing is critical to meeting time constraints.
 - d. Avoid arcane interconnections between or among modules. A guiding principle underlying all design is to keep things simple.
 - e. Don't hard code values into a module's parameter list or calling interface unless absolutely necessary. We must do so on occasion when an interface module or port must be at a specific address location; do not make this a general practice.
3. Loosen the essential interaction between modules, if possible.
Unless the environment demands a high degree of coordination between several modules to accomplish a task or to ensure error-free communication, simply pass the module the information necessary to get the job done. Thereafter, wait for an indication that the task has completed. Execute some other part of the task.

9.9.3 Cohesion

coupling, cohesion

An idea related to *coupling* is *cohesion*. The notion of coupling addresses the partitioning of a system; cohesion addresses bringing the pieces together. Cohesion is a measure of strength of the functional relatedness of elements in a module. The goal is to create strong, highly cohesive modules whose elements are genuinely and tightly related to one another. Conversely, elements should not be strongly related to elements in another module. We want to *maximize* cohesion and *minimize* coupling.

maximize, minimize

The use of cohesion as a reliability and quality metric has been around since the mid-1960s. A number of years of refinement and integration of the ideas of many people studying various designs and design approaches led Constantine and Yordon, *Structured Design*, Prentice-Hall (1979), to formulate a cohesion scale based on an ease of maintenance metric.

Let's look at several different kinds of cohesion.

Functional Cohesion The module implements a single task, and all comprising elements contribute to the execution of that one task.

Sequential Cohesion The module implements a task as a sequential set of procedures. The output data of each procedure becomes the input data to the next. All of the comprising elements are involved in one of those procedures.

Communicational Cohesion The module implements a task that has a number of procedures working on the same set of input data such as an image processing task.

Procedural Cohesion The module implements a number of procedures that may or may not be related to a common activity. Control, rather than data, flows from one procedure to the next.

Temporal Cohesion The module implements a number of unrelated procedures or activities that are sequentially ordered in time.

Logical Cohesion The module implements a number of procedures that are possible alternative methods for accomplishing a task. A subset of those alternatives is selected by an outside user to actually execute the task.

Coincidental Cohesion The module aggregates a number of unrelated procedures. Such cohesion, or lack thereof, should not be used.

We compare the different kinds of cohesion and coupling from several different perspectives in Table 9.0. The ranking is Excellent/Easy = 5 . . . Poor/Difficult = 1.

Table 9.0 Comparison of Coupling and Types Cohesion from Different Perspectives

Cohesion	Coupling	Ease of Modification	Ease of Understanding	Ease of Maintenance
Functional	5	5	5	5
Sequential	4	4	4	3-4
Communicational	3	3	3	3
Procedural	2-3	2-3	2-3	2
Temporal	1	3	3	2
Logical	1	1	2	1
Coincidental	1	1	1	1

Cohesion and coupling analyses provide a good set of metrics by which to begin to assess the high-level architectural aspects of a design. Remember, however, that both are guidelines. The work to ensure that the design is solid and that it is thoroughly tested still needs to be done.

There are plenty of good designs that require tightly coupled modules; CDMA cell phones are a good example. One can have tightly coupled multiprocessor designs as well as designs based on message passing. The implication is not that one design is right or wrong, or better than the other; it is just how it was done to meet the requirements.

9.9.4 More Considerations

- spatial* With today's systems, a *spatial* point of view is often essential. This is an external view of the system, and it yields a distributed functional architecture. With such a view, performance and communication costs are taken into consideration. Closely associated with the spatial viewpoint is that of *resource* allocation; again, this is an external view. Such efforts result in a "resource architecture." Once again, performance, costs, and dependability are factors that must be considered.
- resource*
- hardware* Finally, one must consider the *hardware* and the *software*. Decomposition becomes a design process that leads to a hardware architecture as was discussed earlier. Now performance must be considered. As embedded developers, we are playing a direct role in the design and selection of the hardware platform as well as the software environment. Making trade-offs intelligently in these two areas can take us a long way toward developing a safe, robust, and high-quality/high-performance system.
- software*

9.10 FUNCTIONAL DESIGN

- functional* The purpose at this stage of the design is to find an appropriate internal *functional* architecture for the system. We are beginning to formulate *how* the requirements that have been identified can be implemented. The current focus is on analyzing the problem. Through such analysis, a somewhat loose understanding of the design can be transformed into a precise description. The result of such a process is a detailed textual or graphical description of the system. The end result is a complete consistent functional definition of the required tasks.

To establish an appreciation of a functional model of a system, consider an aircraft. If an aircraft is the system to be designed, the top-level functional model should probably not consist of more than three major functions: *take-off*, *fly*, and *land*. With such a view, we

make no statements about such issues as the support structure for the aircraft (wheels, skies, pontoons), the propulsion system (jet, rocket, propeller), or the method of lift (wings, conventional aircraft or blade, helicopter). Early on, these are not important; such decisions can be postponed until later. The advantage of such an approach is early flexibility—time to explore before beginning to constrain the system. A functional description simply formalizes the intended behavior of the design.

The functional description should be written to be understood by those knowledgeable in the application domain and by those who will do the hardware and software development. The specification must also be such that it can be reviewed by the many diverse and interested parties and tested against reality. If it is too complex to read and understand, no one will read it. When the completed project is delivered, it is too late to discover that the customer's view and developer's view of reality are totally different.

functional model

A first functional decomposition is carried out based on a search of essential internal variables and events in the system. The design process then consists of successive refinements or decompositions for each function (using exactly the same process) until elementary or leaf functions are obtained. Such decomposition forms a *functional model* of the system. The model expressed by the collection of such functions should be sufficient to verify the design quality and to evaluate system behavior and performance.

During modeling and verification, the system's operations and associated performance requirements can be allocated to the internal functions and the relations between such functions can be defined. Such a process also allows one to estimate the expected performance of the system.

external internal

As with the requirements specification, ideally, the functional model should be executable so as to permit verification with respect to the specification. There are tools today that will allow us to do this. One such tool is a behavioral Verilog model. UML is also beginning to make executable models a reality.

The functional model is different from the specification and also from the physical architecture that will be developed next. The specification describes the *external* behavior of the system; the functional model targets the *internal* behavior that will lead to the external. The architectural model addresses the physical hardware and software components onto which the functions are mapped.

In Figure 9.20, we illustrate a first-level decomposition of a simple input/output task. The system must receive data from and transmit data to the outside world. Associated with the task is a code conversion to ASCII.

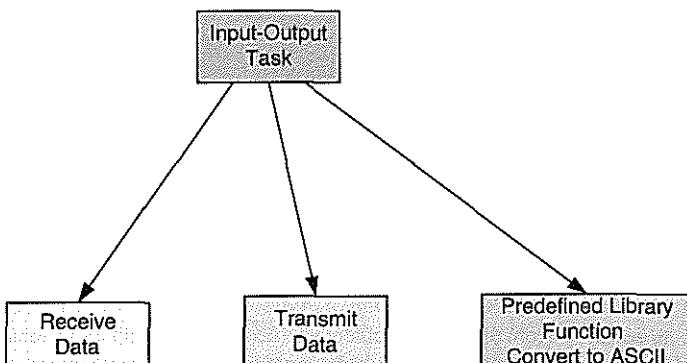


Figure 9.20 First-Level I/O Task Decomposition

Each of these functions may be further decomposed as necessary. If required, the second-level functions may also be successively refined to give the detail needed to understand and to execute the design.

The next step in the analysis is to identify the messages that flow between the user or other active external objects and the system as well as the internal signals that flow between the major functional blocks. We identify how the user will interact with the system in order to make it do what it is intended to do.

Let's now apply our understanding of partitioning to the functional design of counter system. First and foremost, we must continue to postpone the idea of working with the specific data structures, bits, bytes, microprocessors, or array logics for a while longer. Though important later in the process, at the moment, they limit exploration and can bias the functional decomposition of the system.

EXAMPLE 9.0

Designing a Counter (Cont.)

Identifying the Functions

The first diagram, in Figure 9.21, presents an aggregation of the objects in the system. That aggregation includes both the environment and the counter being designed.

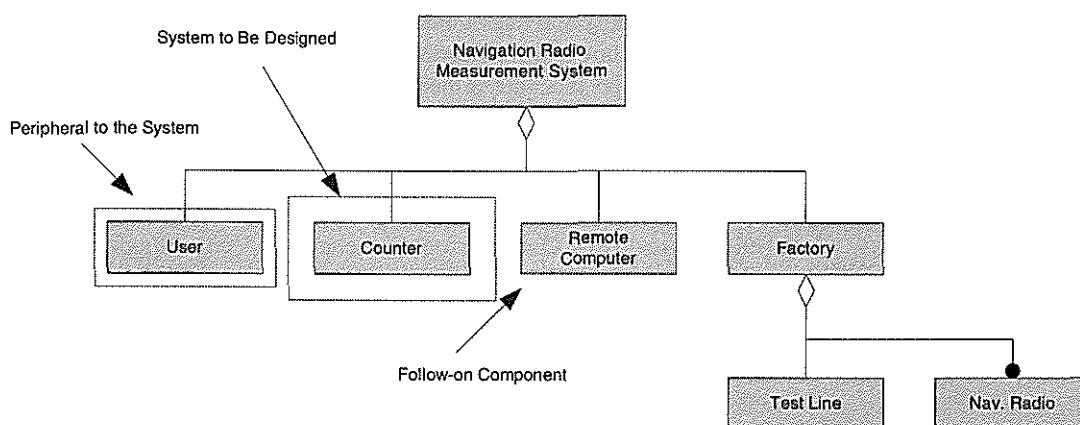


Figure 9.21 A Model of the Environment and the Counter as an Aggregation of Objects

The model of the measurement system is expressed as a collection of

- The user
- The factory
- The future remote computer
- The counter

The factory is an aggregation of test lines and numbers of navigation radios that must be tested. Note that we are using the looser term *aggregation* rather than *composition* here.

The design specification provided a high-level block diagram of the system. For this problem such a diagram provides a good starting place for the initial hierarchical decomposition of the system. Figure 9.22 elaborates on the counter component and gives one possible decomposition for that system.

The interface to the outside world is segregated into two functional blocks. The first is associated with the *presentation of information* to the user. The second is charged with *bringing in information* from the user and other tasks necessary to support the measurement. Both functional blocks are further decomposed into local operations versus remote operations.

Such a choice is made in the first case because the display is considered to be an output function and control to be an input function. In the second case, two different sets of functionality and differ-

aggregation, composition
presentation of information,
bringing in information

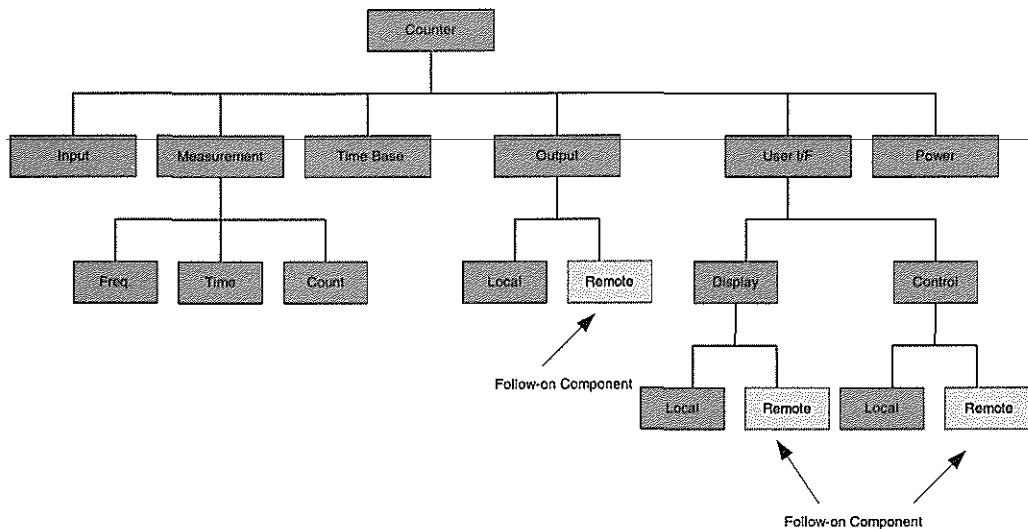


Figure 9.22 A Possible Hierarchical Decomposition of the Counter System

grammars for expressing the user's commands are anticipated. Front panel operations tend to be rather straightforward; remote operations can be a bit more involved. Certainly, these are not the only choices. The drawing in Figure 9.23 captures the interface between the counter and the surrounding environment.

The next drawing, in Figure 9.24, expresses a functional partitioning and the signal flow between the major functional blocks.

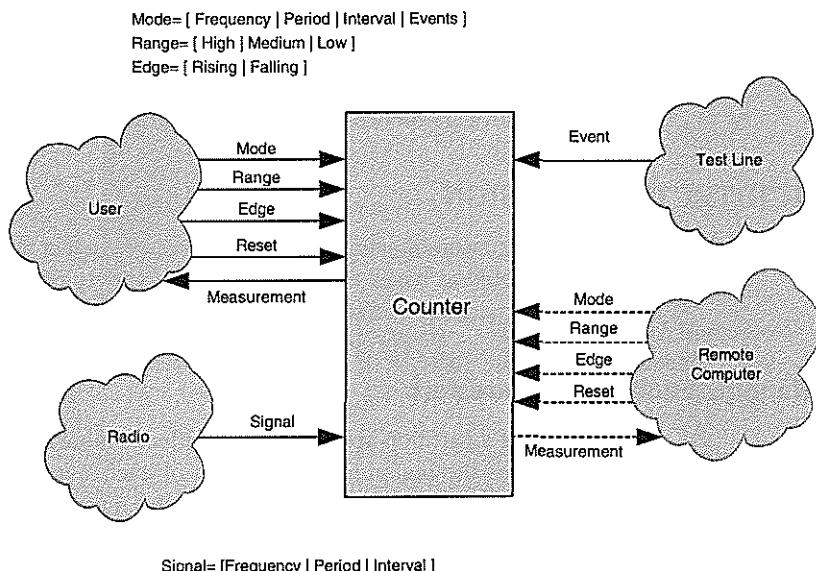


Figure 9.23 The Counter-Environment Interface

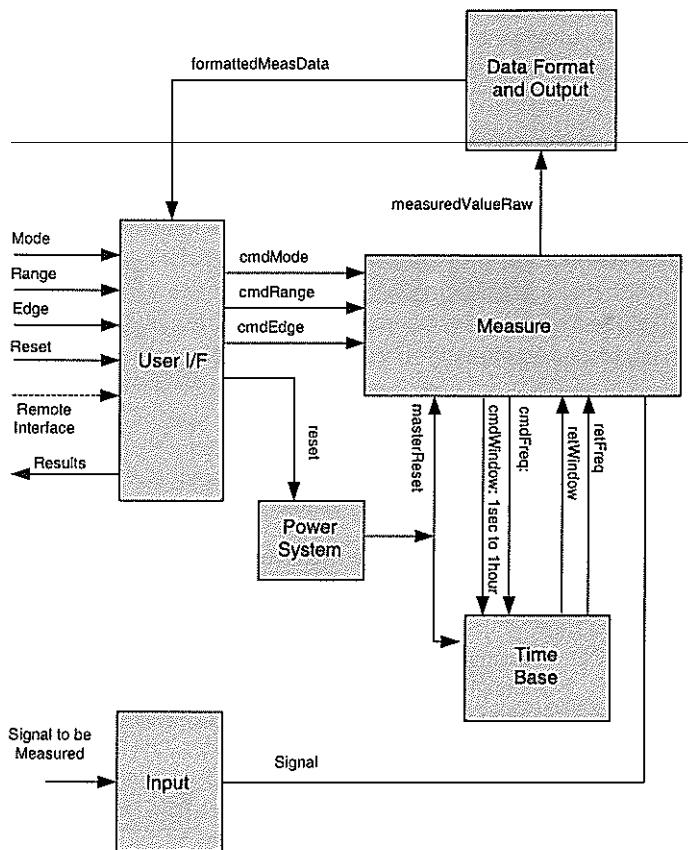


Figure 9.24 A Functional Partition of the Counter System

Next the system architecture is formulated, and functions are mapped onto the hardware and software blocks comprising that system.

9.11 ARCHITECTURAL DESIGN

In executing an architectural design, the goal is to select the most appropriate solution to the original problems based on exploration of a variety of architectures and the choice of the best-suited hardware/software partitioning and allocation of functionality.

9.11.1 Mapping Functions to Hardware

The view of a partition now changes to reflect a more detailed understanding of the system and involves the *mapping* or allocation of each functional module onto the appropriate physical hardware or software block(s). Such a mapping completely describes the hardware implementation of the system.

As noted earlier, one should always endeavor to broaden the scope of the architectural design so as not to preclude possible future enhancements. Certainly, this involves a balancing act between generality and practicality as well as simultaneously satisfying other specified requirements. Nonetheless, the plan should be for a system that evolves over its

lifetime; if this is done well, add-ons, which are inevitable in today's systems, will be much easier.

The major objective of the architectural design activity is the allocation or mapping of the different pieces of system functionality to the appropriate hardware and software blocks. Work is based on the detailed functional structure. The performance requirements are analyzed, and finally the constraints that are imposed by the available technologies as well as those that arise from the hardware and software specifications are taken into consideration.

The important constraints that must be considered include such items as

- The geographical distribution
- Physical and user interfaces
- System performance specifications
- Timing constraints and dependability requirements
- Power consumption
- Legacy components and cost

Such constraints are strong factors for deciding which portions of the system should be implemented in software and which portions should be done in hardware.

The proper allocation of the pieces of functionality is generally obvious for a significant part of the system. For those, it is easy to say, "this part must be hardware or this part must be software." The power supply, display, communications port, and the package containing the system are necessarily hardware. The operating system and associated drivers, it is generally agreed, are necessarily software.

The situation is expressed graphically as in Figure 9.25. There is a gray area between the hardware and software where the implementation approach is not precisely defined. In selecting the components that make up this area, one is making engineering decisions or trade-offs of speed, cost, size, weight, as well as many other factors.

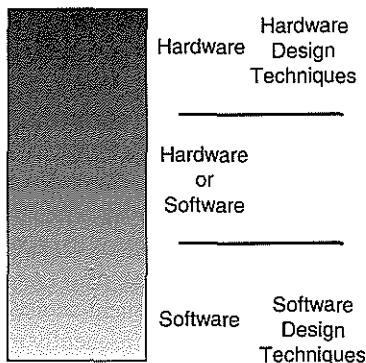


Figure 9.25 The Hardware-Software Continuum

mapping

The *mapping* onto such an architecture completely defines the hardware implementation of the system. The hardware portion of the system is specified by a physical architecture that may comprise one or more microprocessors, complex logical devices or array logics, or/and custom-integrated circuits. It is important to remember that with today's systems, these microprocessors and microcontrollers can take on a variety of personalities: (CISC) Complex Instruction Set, (RISC) Reduced Instruction Set, or (DSP) Digital Signal Processor.

For most applications, a substantial portion of the software can be easily separated from the hardware and thereby permit its concurrent development. The remaining part, that in the *co-design* boundary, is more difficult to partition and falls under what is called *co-design*.

9.11.2 Hardware and Software Specification and Design

The system specification gives a detailed quantification of the system's inputs, outputs, and functional behavior based on our original requirements. The functional decomposition is analogous to those steps taken in defining the requirements. As the architecture of the design now begins to take shape, the objective is to determine, as fully as possible, the specifications for each of the physical components in the system and the interfaces between them. The specification of the hardware for the entire system is decided by defining the hardware architecture and all its properties. The specification of the software is obtained by defining the software implementation or block diagram (using any of a variety of methods) for each software component of the architecture.

Each functional subset to be implemented in software is described by a detailed software specification that expresses the priority of each task and the spatial (data coupling) and temporal dependence relations between tasks. UML diagrams, including detailed state charts, timing diagrams, sequence diagrams, activity diagrams, and collaboration diagrams, can all be very useful at this stage in the design.

A software implementation may or may not use a real-time kernel. With an off-the-shelf real-time kernel, the development time is reduced, but not the factory cost or time-based performance specifications. For systems that do not use a real-time kernel (which represents 80 to 90% of small and medium systems), one can achieve a better optimization of the design when addressing high-speed, hard real-time constraints. Under such circumstances, the solution is being hand tailored to the specific problem rather than adapting a general-purpose solution to a specific case.

For the software design, the following must be analyzed and decided:

- Whether to use a real-time kernel
- Whether several functions can be combined in order to reduce the number of software tasks and if so, how?
- A priority for each task
- An implementation technique for each intertask relationship

When it is appropriate, a real-time kernel or the services of an operating system can be used. In general, the main objective is to reduce the complexity of the organizational part in order to reduce the size and complexity of the software and the resulting development, testing, and debugging times.

Under such circumstances, a frequent choice is the *Rate-Monotonic Scheduling* policy (this will be discussed shortly; more frequently executed tasks are assigned a higher priority). Permanent functions (those that run continuously without an activating event) and some cyclic functions without timing constraints are usually implemented within a background task.

For the implementation of intertask relationships, it is desirable to use procedure calls as much as possible, thereby simplifying the organizational part and reducing the intertask overhead. Such an implementation is only possible between functions with increasing relative priorities. Tasks triggered by hardware events are invoked through the processor interrupt or polling systems.

For each specific subpart of the system in which the partition is not obvious, a detailed specification is written; the final hardware/software partition is determined through a pro-

Rate-Monotonic Scheduling

cess of successive refinements, as was done in the earlier decomposition process. Each hardware/software partition must also include the hardware interfaces and the software drivers to support any intercomponent communication.

The result is a complete mapping of all remaining functions and functional relations onto the hardware architecture. Among the important criteria that we strive to optimize are:

- Implementation (or factory) cost
- Development time and cost
- Performance and dependability constraints
- Power consumption
- Size

EXAMPLE 9.0

Designing a Counter (Cont.)

Developing the Architecture

The next step in developing the counter begins with formulating the hardware architecture; the software architecture follows. We then map each of the functions identified earlier onto the architecture. The following diagram, in Figure 9.26, presents the hardware components.

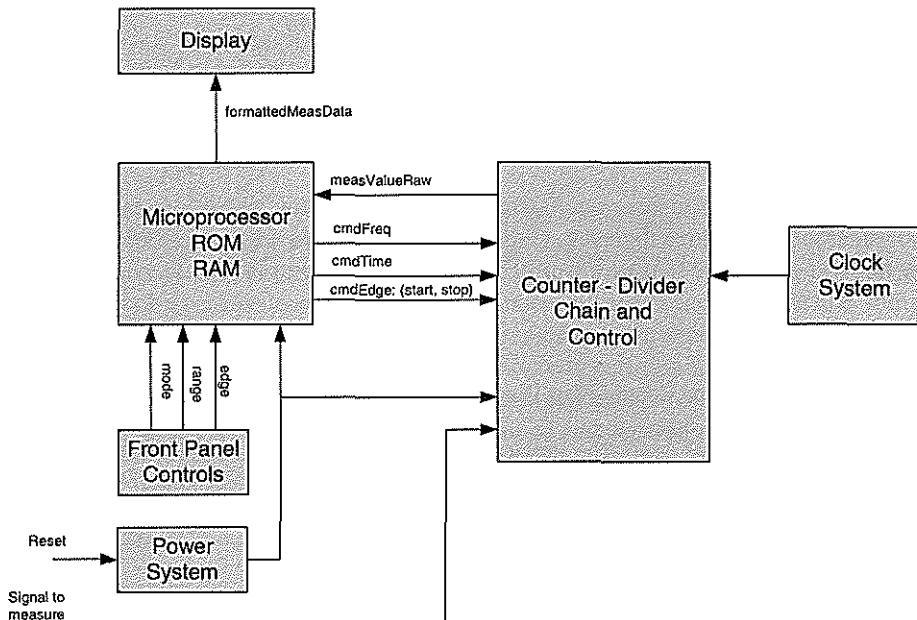


Figure 9.26 The Hardware Architecture of the Counter

In the design, the microprocessor, the display, the front panel controls, and the power system are clearly hardware. In theory, the clock system as well as the counter-divider chain and associated control could be implemented in software. However, the frequency at which the counter is intended to operate (200 MHz) biases the decision toward a hardware solution.

Figure 9.27 identifies the major software tasks, shared data, and I/O in a data and control flow diagram. The *front panel task* is continually checking (directly by polling or indirectly by interrupt) the state of the front panel for user input. A change in input is captured and passed to the *display task* (which will update the display accordingly) and to the *measurement task*. The *measurement task* issues the appropriate commands to the external *counter-divider chain control block*. At the end of each measurement, the raw data is read from the counter-divider and passed to the *output task*.

front panel task
display task
measurement task
counter-divider chain control block, output task

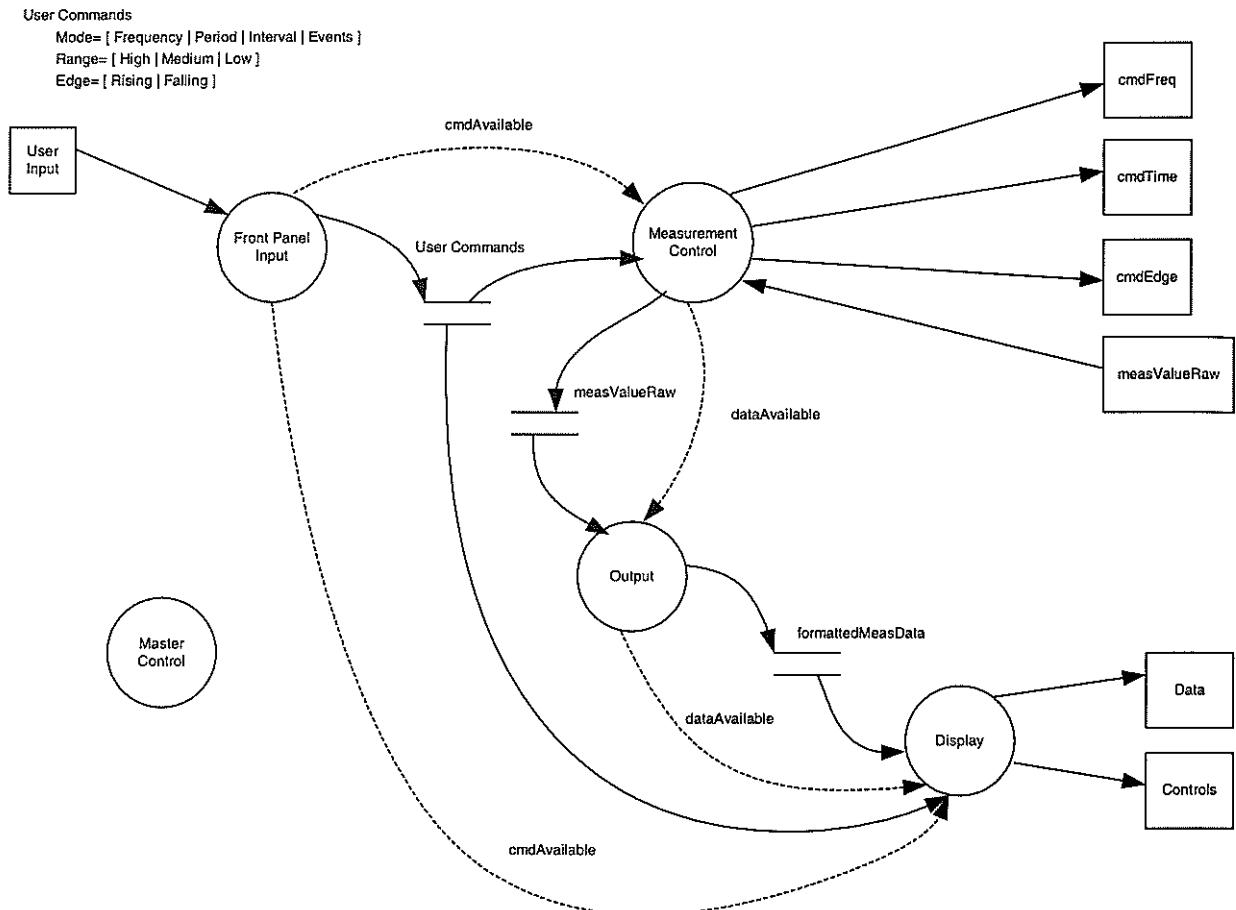


Figure 9.27 A Data and Control Flow Diagram for the Counter System

*output task, display task
master control task*

The *output task* properly formats the data and sends it to the *display task* for display on the front panel. The *master control task* manages the scheduling of all tasks and performs any necessary housekeeping or other duties as necessary.

9.12 FUNCTIONAL MODEL VERSUS ARCHITECTURAL MODEL

functional, architectural

A good question that one might ask at this stage is, “Why is it necessary to design a functional model and an architectural model?” We start by looking at any system—hardware, software, a mix—it doesn’t matter. It quickly becomes evident that the internal organization of a system is based on a collection of components and interconnections among them. An appropriate model has to include elements both at the *functional* level and at the *architectural* level to be able to represent and evaluate hardware/software system.

9.12.1 The Functional Model

interacting functional elements

The functional model describes a system through a *set of interacting functional elements*. The design proceeds at a high level without initial bias toward any specific implementation. We have the freedom to explore and to be creative. The behavior of a functional element is

best described with a hierarchical and graphical model. The functional modules will interact using one of the following three types of relations:

- shared variable* • The *shared variable relation*—which defines a data exchange without temporal dependencies
- synchronization* • The *synchronization relation*—which specifies temporal dependency
- message transfer* • The *message transfer* by port—which implies a producer/consumer kind of relationship

We will discuss each of these relations when we study processes and interprocess communication. All of them are critical in the design and development of today's embedded systems.

9.12.2 The Architectural Model

physical architecture The architectural model describes the *physical architecture* of the system based on real components such as microprocessors, arrayed logics, special-purpose processors, analog and digital components, and the many interconnections between them.

9.12.3 The Need for Both Models

mapping, functional architectural configuration These two views, when considered separately, are not sufficient to completely describe the design of contemporary systems. It is necessary to add the *mapping* between the *functional viewpoint* and the *architectural one*. Such a mapping defines a (functional) partition and the allocation of functional components to the hardware elements. This is also called *architectural configuration*.

functional model The *functional model*, located between specification model and architectural model, is suitable for representing the internal organization of a system. It explains all necessary functions and the couplings between them—expressed from the point of view of the original problem. Using such a scheme leads to a technology-independent solution. In particular with this kind of model, all or part of the description can be implemented in either software or hardware.

The functional model is the basis for a coarse-grain partitioning of the system. Such a partitioning leads naturally to the selection of which functions to implement in hardware or software. The *architectural structure* is finer grained and generally follows from the functional model; the architecture may also be imposed *a priori*.

9.13 PROTOTYPING

The prototype phase leads to an operational system prototype. A prototype implementation includes:

- Detailed design
- Debugging
- Validation
- Testing

Prototyping is naturally a bottom-up process because it consists of assembling individual parts and fleshing out more and more of the abstract functionalities. Each level of the implementation must be validated. That is, it must be checked for compliance with the specifications on the corresponding level in the top-down design.

Hardware and software implementations can be developed simultaneously and involve specialists in both domains, hopefully reducing the total implementation time. Often this

does not happen in reality. Typically, the software leads hardware. Nonetheless, a complete solution can be generated and/or synthesized for both hardware (in the form of ASICs and standard cores, etc.) and software (in the form of hardware/software interfaces). The resulting prototype can then be verified.

9.13.1 Implementation

Activities in this step are highly dependent on the technology used. Remember, the prototype is a tool for understanding and confirming system design. It is a proof of concept. A word of caution: one should not rush the analysis or design to get to the prototype. Also, one should not be afraid to throw the prototype away. For small projects, it sometimes works to try to transform the prototype into the final product. For large projects, it is usually more of a proof-of-concept that almost never can be migrated.

Those who hurry through the design and coding because a lot of testing needs to be done are going to be spending long nights getting things to work and even longer nights with unhappy customers. For some reason, customers do not seem to have much of a sense of humor when the failure of a product they have purchased has just cost them several million dollars. If you are selling to a general market, *your company* has just lost several million in R&D costs and you still do not have a product to take to market. So now, it is even worse, because you have missed an opportunity for sales revenue with a product that you cannot sell because it is poorly conceived, or it still is not ready.

9.13.2 Analyzing the System Design

We have been studying the system design process while moving from requirements to a design. Now that the first-level design is in place, it must be critically analyzed. This step provides several important checks on the design. First and foremost, it verifies that the solution meets the original requirements and specifications. At this stage in the design flow, it may also be necessary to trade off different architectural and functional aspects of the design. Such trade-offs must be made according to criteria identified in the original specification.

The first step entails a static analysis of the system. At this stage, the architecture of the system is examined. Of immediate interest is *not* how the system will behave at runtime. Rather, the major objectives are to have a system that is easy to understand, build, test, and maintain. All too often new designers (and, unfortunately, some who should know better), proclaim “but it works!!!” For systems that we are proud to put our name on, getting it to work is only one very small part of the job. Moreover, it is easy to make a one-off version of any system to work. Making one or ten million of the same design in production that will ultimately work safely and reliably is a much larger challenge.

The main goal in any design is to work ourselves out of a job. We want the design to be so reliable and so well documented that any future modifications and extensions will be effortless. The caveat, of course, is that one must also know what sufficient reliability is and when to stop documenting. Well-documented means just enough so that people can easily understand the design, but not so much that it becomes the primary deliverable.

9.13.2.1 Static Analysis

Static analysis should consider three areas:

1. Coupling

We have examined this aspect of a design already. Coupling is related to the number and complexities of the relationships that exist among the various system modules. It also gives a measure of the implications of a change. The goal is loose coupling.

*functional, behavioral***2. Cohesiveness**

Another issue that is worth stressing again is cohesiveness, which is a measure of the functional homogeneity of elements that comprise the modules. This applies to both the components and the relations. One must consider both external and internal views. External cohesion begins with the appropriate naming and meaning for elements. Internally, the structure and relationships among components is analyzed. For example, coupling through shared data is more cohesive than messages. Messages imply a temporal dependency.

3. Complexity

Two kinds of complexity are identified: *functional* and *behavioral*.

Functional complexity is characterized by:

- The number of internal functions and relational components
The goal is to keep these small. Generally, as the number of functions and relations decreases, so does the complexity of the design. Note: this does not mean to sacrifice clarity.
- Interconnections among elements comprising each module
The earlier discussion of coupling applies here as well. Keep things simple.

Behavioral complexity is characterized by:

- The number of inputs and outputs
Once again, the target is a smaller number.
- The length and ease of reading and understanding the description of the module
If several paragraphs or a page of written text (in sub 6 point font) are required to describe the function of one of the modules, that module is probably too complex. To simplify such descriptions, use tables, logical equations, or pseudo code.
- The flow control through the module and the number and structure of state variables
Have a single major thread of control through the module and keep the number of states small.

*operational***9.13.2.2 Dynamic Analysis***performance*

The objective in performing a dynamic analysis on the system is to determine how it will behave in a context that closely approximates the ultimate working environment. Dynamic analysis considers the following:

- Behavior Verification
The goal is to ensure that the behavior of the system, in its operating environment, meets the *operational* specification. That is, does it perform the functions it was intended to perform? This verification includes behavior at the boundaries of those functions. To be able to do so, of course, we need a good specification in the first place.
- Performance Analysis
Performance Analysis ensures that the system, in its operating environment, meets the *performance* specification. The focus is on specific values for inputs and outputs. We'll talk about this in a later chapter.
- Trade-off Analysis
A trade-off analysis is necessary to determine the optimal solution for the given constraints and objectives. Such an analysis, based on only a small set of performance criteria, may affect the ultimate success or failure of the product.

Frank Vahid/ Tony Givargis

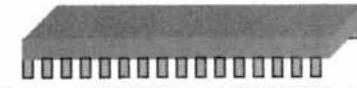
Embedded System Design, A Unified Hardware/Software Introduction

Wiley, ISBN 0-471-38678-2.

Chapter 6, Interfacing, pp 137- 153

- 5.11 A given design with cache implemented has a main memory access cost of 20 cycles on a miss and two cycles on a hit. The same design without the cache has a main memory access cost of 16 cycles. Calculate the minimum hit rate of the cache to make the cache implementation worthwhile.
- 5.12 Design your own $8K \times 32$ PSRAM using an $8K \times 32$ DRAM, by designing a refresh controller. The refresh controller should guarantee refresh of each word every 15.625 microseconds. Because the PSRAM may be busy refreshing itself when a read or write access request occurs (i.e., the enable input is set), it should have an output signal ack indicating that an access request has been completed. Make use of a timer. Design the system down to complete structure. Indicate at what frequency your clock must operate.

CHAPTER 6: *Interfacing*



- 6.1 Introduction
- 6.2 Communication Basics
- 6.3 Microprocessor Interfacing: I/O Addressing
- 6.4 Microprocessor Interfacing: Interrupts
- 6.5 Microprocessor Interfacing: Direct Memory Access
- 6.6 Arbitration
- 6.7 Multilevel Bus Architectures
- 6.8 Advanced Communication Principles
- 6.9 Serial Protocols
- 6.10 Parallel Protocols
- 6.11 Wireless Protocols
- 6.12 Summary
- 6.13 References and Further Reading
- 6.14 Exercises

6.1 Introduction

As stated in the Chapter 5, we use processors to implement processing, memory to implement storage, and buses to implement communication. The earlier chapters described processors and memory. This chapter describes implementing communication with buses, known as interfacing. Communication is the transfer of data among processors and memories. For example, a general-purpose processor reading or writing a memory is a common form of communication. A general-purpose processor reading or writing a peripheral's register is another common form.

We begin by defining some basic communication concepts. We then introduce several issues relating to the common task of interfacing to a general-purpose processor: addressing, interrupts, and direct memory access. We also describe several schemes for arbitrating among multiple processors attempting to access a single bus or memory simultaneously. We show

that many systems may include several hierarchically organized buses. We then discuss some more advanced communication principles and survey several common serial, parallel, and wireless communication protocols.

6.2 Communication Basics

Basic Terminology

We begin by introducing a very basic communication example between a processor and a memory, shown in Figure 6.1. Figure 6.1(a) shows the bus structure, or the wires connecting the processor and the memory. A line *rd'/wr* indicates whether the processor is reading or writing. An *enable* line is used by the processor to carry out the read or write. Twelve address lines *addr* indicate the memory address that the processor wishes to read or write. Eight data lines *data* are set by the processor when writing or set by the memory when the processor is reading. Figure 6.1(b) describes the read protocol over these wires: the processor sets *rd'/wr* to 0, places a valid address on *addr*, and strobes *enable*, after which the memory will place valid data on the *data* lines. Figure 6.1(c) shows a write protocol: the processor sets *rd'/wr* to 1, places a valid address on *addr*, places data on *data*, and strobes *enable*, causing the memory to store the data. This very simple example brings up several points that we now describe.

Wires may be unidirectional, meaning they transmit in only one direction, as did *rd'/wr*, *enable*, and *addr*; or they may be bidirectional, meaning they transmit in two directions, though in only one direction at a time, as did *data*. A set of wires with the same function is typically drawn as a thick line and/or as a line with a small angled line drawn through it, as was the case with *addr* and *data*.

The term *bus* can refer to a set of wires with a single function within a communication. For example, we can refer to the “address bus” and the “data bus” in the above example. The term *bus* can also refer to the entire collection of wires used for the communication (e.g., *rd'/wr*, *enable*, *addr*, and *data*) along with the communication protocol over those wires. Both uses of the term are common and are often used in conjunction with one another. For example, we may say that the processor’s bus consists of an address bus and a data bus. A *protocol* describes the rules for communicating over those wires. We deal primarily with low-level hardware protocols in this chapter, while higher-level protocols, like IP (Internet Protocol) can be built on top of these protocols, using a layered approach.

The bus connects to ports of a processor (or memory). A *port* is the actual conducting device, like metal, on the periphery of a processor, through which a signal is input to or output from the processor. A port may refer to a single wire, or to a set of wires with a single function, such as an address port consisting of twelve wires. A related term is *pin*. When a processor is packaged as its own IC, there are actual pins extending from the package, and those pins are often designed to be plugged into a socket on a printed-circuit board. Today, however, a processor commonly coexists on a single IC with other processors and memories. Such a processor does not have any actual pins on its periphery, but rather “pads” of metal in

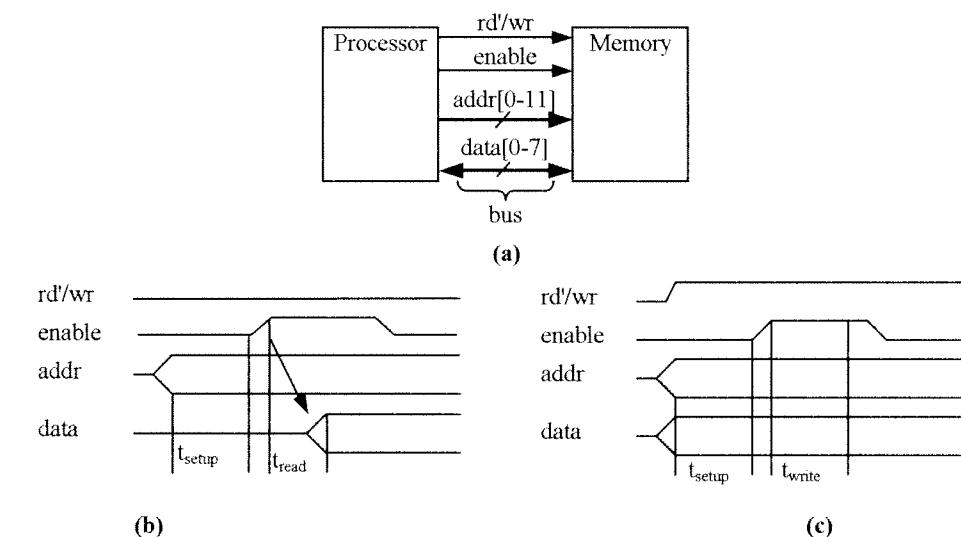


Figure 6.1: A simple bus example: (a) bus structure, (b) read protocol, (c) write protocol.

the IC. In fact, even for a processor packaged in its own IC, alternative packaging-techniques may use something other than pins for connections, such as small metallic balls. However, we can still use the term *pin* to refer to a port on a processor.

The distinction between a bus and a port is similar to the distinction between a street and a driveway — the bus is like the street, which connects various driveways. A processor’s port is like a house’s driveway, which provides access between the house and the street.

The most common method for describing a hardware protocol is a timing diagram, as was used in Figure 6.1(b) and (c). In the diagram, time proceeds to the right along the x-axis. The diagram shows that the processor must set the *rd'/wr* line low for a read to occur. The diagram also shows, using two vertical lines, that the processor must place the address on *addr* for at least *t_{setup}* time before setting the *enable* line high. The diagram shows that the high *enable* line triggers the memory to put data on the *data* wires after a time *t_{read}*. Note that a timing diagram represents control lines, like *rd'/wr* and *enable*, as either being high or low, while it represents data lines, like *addr* and *data*, either as being invalid or valid, using a single horizontal line or two horizontal lines, respectively. The actual value of data lines is not normally relevant when describing a protocol, so that value is typically not shown.

In the above protocol, the control line *enable* is active high, meaning that a 1 on the *enable* line triggers the data transfer. In many protocols, control lines are instead active low, meaning that a 0 on the line triggers the transfer. Such a control line’s name is typically written with a bar above it, a single quote after it (e.g., *enable'*), a forward slash before it (e.g., */enable*), or the letter *L* after it (e.g., *enable_L*). To be general, we will use the term *assert* to mean setting a control line to its active value, such as to 1 for an active high line, and to 0 for an active low line. We will use the term *deassert* to mean setting the control line to its inactive

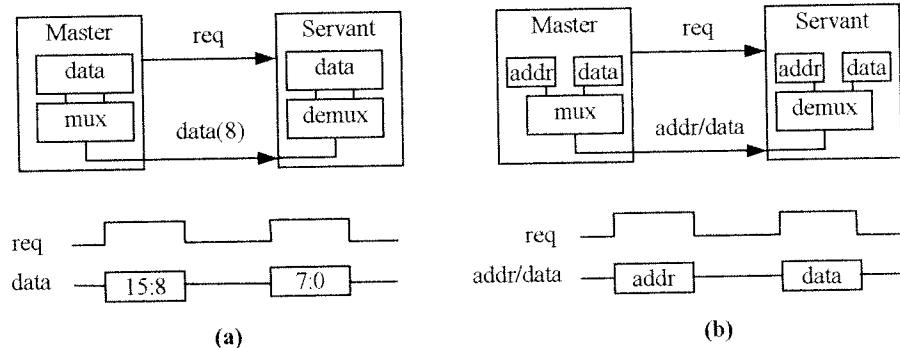


Figure 6.2: Time-multiplexed data transfer: (a) data serializing, (b) address/data muxing.

value. Notice that the rd'/wr of our earlier example merges two control signals into one line. so we accomplish a read by setting rd'/wr to 0 and a write by setting rd'/wr to 1.

A protocol typically consists of several possible subprotocols, such as a read protocol and a write protocol. Each subprotocol is known as a *transaction* or a *bus cycle*. A bus cycle may consist of several clock cycles.

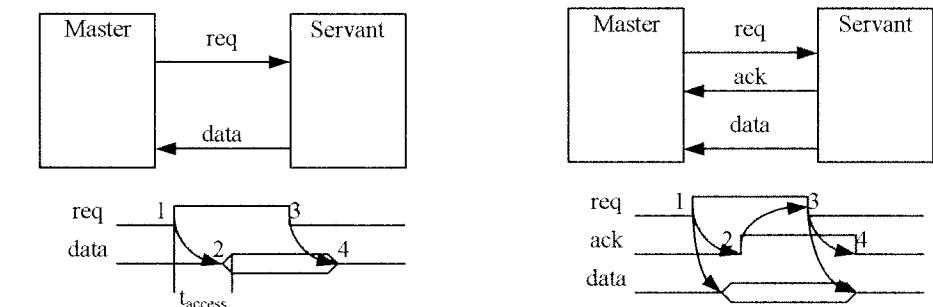
Basic Protocol Concepts

The processor-memory protocol described above was a simple one. Hardware protocols can be much more complex. However, we can understand them better by defining some basic protocol concepts. These concepts are: actors, data direction, addresses, time-multiplexing, and control methods.

An *actor* is a processor or memory involved in the data transfer. A protocol typically involves two actors: a master and a servant. A master initiates the data transfer. A servant, commonly called a slave, responds to the initiation request. In the example of Figure 6.1, the processor is the master, and the memory is the servant (i.e., the memory cannot initiate a data transfer). The servant could also be another processor. Masters are usually general-purpose processors, and servants are usually peripherals and memories.

Data direction denotes the direction that the transferred data moves between the actors. We indicate this direction by denoting each actor as either receiving or sending data. Note that actor types are independent of the direction of the data transfer. In particular, a master may either be the receiver of data, as in Figure 6.1(b), or the sender of data, as shown Figure 6.1(c).

Addresses represent a special type of data used to indicate where regular data should go to or come from. A protocol often includes both an address and regular data, as did the memory access protocol in Figure 6.1, where the address specified the location where the data should be read from or written to in the memory. An address is also necessary when a general-purpose processor communicates with multiple peripherals over a single bus; the address not only specifies a particular peripheral, but also may specify a particular register within that peripheral.



1. Master asserts *req* to receive data
2. Servant puts data on bus within time t_{access}
3. Master receives data and deasserts *req*
4. Servant ready for next request

(a)

(b)

Figure 6.3: Two protocol control methods: (a) strobe, (b) handshake. The main differences are underlined.

Another protocol concept is *time multiplexing*. To multiplex means to share a single set of wires for multiple pieces of data. In time multiplexing, the multiple pieces of data are sent one at a time over the shared wires. For example, Figure 6.2(a) shows a master sending 16 bits of data over an 8-bit bus using a strobe protocol and time-multiplexed data. The master first sends the high-order byte and then the low-order byte. The servant must receive the bytes and then demultiplex the data. This serializing of data can be done to any extent, even down to a 1-bit bus, in order to reduce the number of wires. As another example, Figure 6.2(b) shows a master sending both an address and data to a servant, such as a memory. In this case, rather than using separate sets of lines for address and data, as was done in Figure 6.1, we can time multiplex the address and data over a shared set of lines *addr/data*.

Control methods are schemes for initiating and ending the transfer. Two of the most common methods are strobe and handshake. In a *strobe* protocol, the master uses one control line, often called the *request* line, to initiate the data transfer, and the transfer is considered to be complete after some fixed time interval after the initiation. For example, Figure 6.3(a) shows a strobe protocol with a master wanting to receive data from a servant. The master first asserts the request line to initiate a transfer. The servant then has time t_{access} , to put the data on the data bus. After this time, the master reads the data bus, believing the data to be valid. The master then deasserts the request line, so that the servant can stop putting the data on the data bus, and both actors are then ready for the next transfer. An analogy is a demanding boss who tells an employee “I want that report (the data) on my desk (the data bus) in one hour (t_{access}),” and merely expects the report to be on the desk in one hour.

The second common control method is a *handshake* protocol, in which the master uses a request line to initiate the transfer, and the servant uses an *acknowledge* line to inform the master when the data is ready. For example, Figure 6.3(b) shows a handshake protocol with a

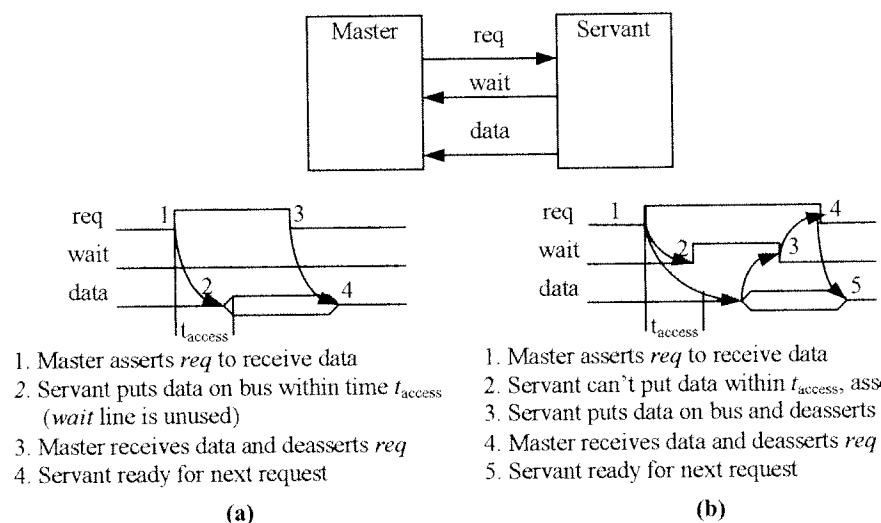


Figure 6.4: A strobe/handshake compromise: (a) fast-response, (b) slow-response. The differences are underlined.

receiving master. The master first asserts the request line to initiate the transfer. The servant takes as much time as necessary to put the data on the data bus, and then asserts the acknowledge line to inform the master that the data is valid. The master reads the data bus and then deasserts the request line so that the servant can stop putting data on the data bus. The servant deasserts the acknowledge line, and both actors are then ready for the next transfer. In our boss-employee analogy, a handshake protocol corresponds to a more tolerant boss who tells an employee “I want that report on my desk soon; let me know when it’s ready.” A handshake protocol can adjust to a servant, or servants, with varying response times, unlike a strobe protocol. However, when response time is known, a handshake protocol may be slower than a strobe protocol, since it requires the master to detect the acknowledgment before getting the data, possibly requiring an extra clock cycle if the master is synchronizing the bus control signals. A handshake also requires an extra line for acknowledge.

To achieve both the speed of a strobe protocol and the varying response time tolerance of a handshake protocol, a compromise protocol is often used, as illustrated in Figure 6.4. In this case, when the servant can put the data on the bus within time t_{access} , the protocol is identical to a strobe protocol, as shown in Figure 6.4(a). However, if the servant cannot put the data on the bus in time, it instead tells the master to wait longer, by asserting a line we’ve labeled *wait*. When the servant has finally put the data on the bus, it deasserts the *wait* line, thus informing the master that the data is ready. The master receives the data and deasserts the *request* line. Thus, the handshake only occurs if it is necessary. In our boss-employee analogy, the boss tells the employee “I want that report on my desk in an hour; if you can’t finish by then, let me know that and then let me know when it’s ready.”

Perhaps the most common communication situation in embedded systems is the input and output (I/O) of data to and from a general-purpose processor, as it communicates with its

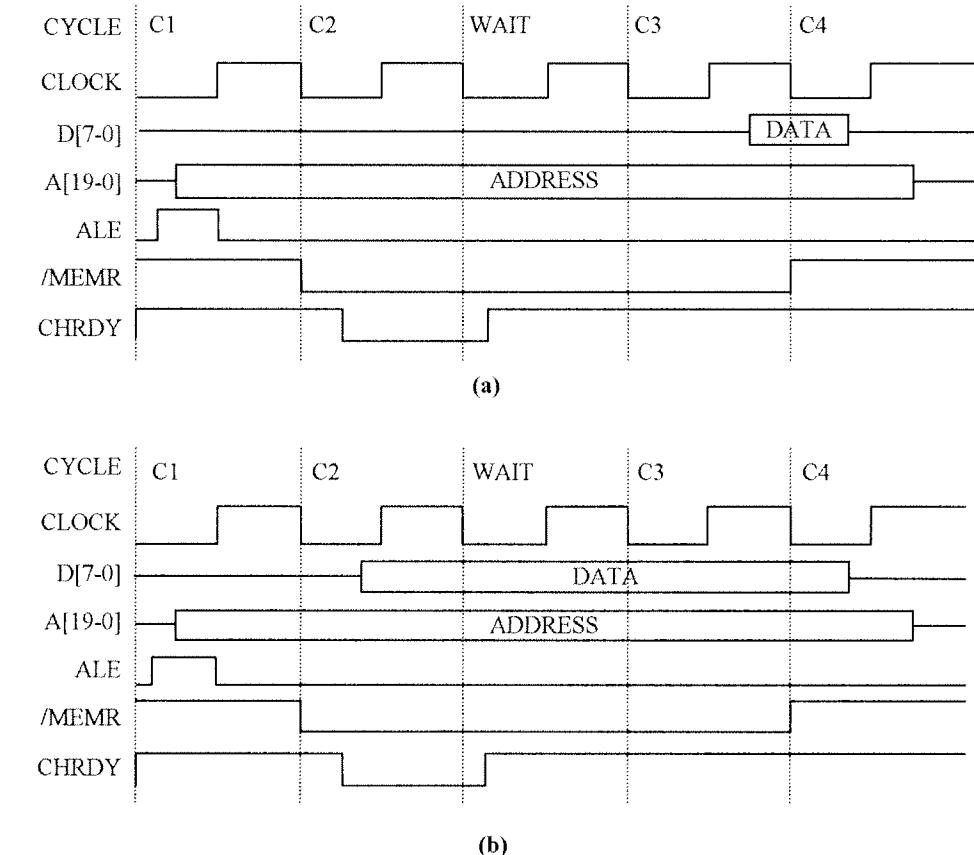


Figure 6.5: ISA bus protocol: (a) read bus timing, (b) write bus timing.

peripherals and memories. I/O is relative to the processor: input means data comes into the processor, while output means data goes out of the processor. In the next three sections, we will discuss three microprocessor-interfacing issues: addressing, interrupts, and direct memory access. We’ll use the term *microprocessor* to refer to a general-purpose processor.

Example: The ISA Bus Protocol — Memory Access

The Industry Standard Architecture (ISA) bus protocol is common in systems using an 80x86 microprocessor. Figure 6.5(a) illustrates the bus timing for performing a memory read operation, referred to as a *memory read cycle*. During a memory read cycle, the microprocessor drives the bus signals to read a byte of data from memory. Note that in Figure 6.5(a), several other control signals that are inactive during a memory read cycle are not

included in the timing diagram. The operation works as follows. In clock cycle C1, the microprocessor puts a 20-bit memory address on the address lines *A* and asserts the address latch enable signal *ALE*. During clock cycles C2 and C3, the processor asserts the memory read signal *MEMR* to request a read operation from the memory device. After C3, the memory device holds the data on data lines *D*. In cycle C4, all signals are deasserted.

The ISA read bus cycle uses a compromise strobe/handshake control method. The memory device deasserted the channel ready signal *CHRDY* before the rising clock edge in C2, causing the microprocessor to insert wait cycles until *CHRDY* was reasserted. Up to six wait cycles can be inserted by a slow device.

Figure 6.5(b) illustrates the bus timing for performing a memory write operation, referred to as a *memory write cycle*. During a memory write bus cycle, the microprocessor drives the bus signals to write a byte of data to memory. The operation works as follows. In clock cycle C1, the processor puts the 20-bit memory address to be written on the address lines and asserts the *ALE* signal. During cycles C2 and C3, the processor puts the write data on the data lines and asserts the memory write signal *MEMW* to indicate a write operation to the memory device. In cycle C4, all signals are deasserted. The write cycle also uses a compromise strobe/handshake control method.

6.3 Microprocessor Interfacing: I/O Addressing

Port and Bus-Based I/O

A microprocessor may have tens or hundreds of pins, many of which are control pins, such as a pin for clock input and another input pin for resetting the microprocessor. Many of the other pins are used to communicate data to and from the microprocessor, which we call processor I/O. There are two common methods for using pins to support I/O: port-based I/O and bus-based I/O.

In *port-based I/O*, also known as *parallel I/O*, a port can be directly read and written by processor instructions just like any other register in the microprocessor; in fact, the port is usually connected to a dedicated register. For example, consider an 8-bit port named *P0*. A C-language programmer may write to *P0* using an instruction like: *P0 = 255*, which would set all eight pins to 1s. In this case, the C compiler manual would have defined *P0* as a special variable that would automatically be mapped to the register *P0* during compilation. Conversely, the programmer might read the value of a port *P1* being written by some other device by typing something like *a = P1*. In some microprocessors, each bit of a port can be configured as input or output by writing to a configuration register for the port. For example, *P0* might have an associated configuration register called *CP0*. To set the high-order four bits to input and the low-order four bits to output, we might say: *CP0 = 15*. This writes 00001111 to the *CP0* register, where a 0 means input and a 1 means output. Ports are often bit-addressable, meaning that a programmer can read or write specific bits of the port. For example, one might say: *x = P0.2*, giving *x* the value of the number 2 pin of port *P0*.

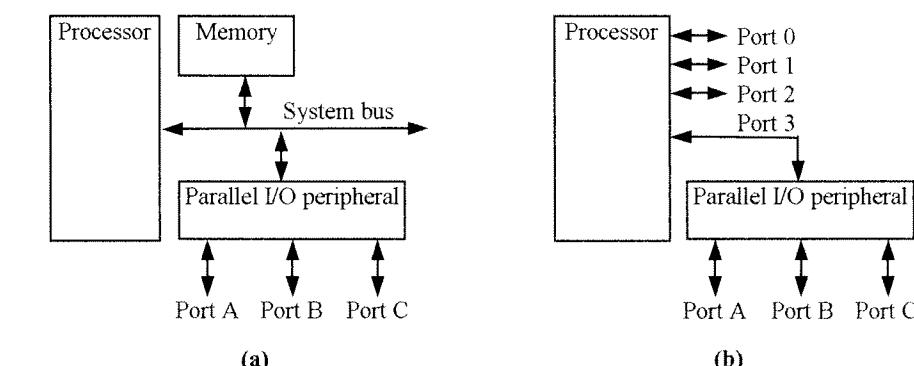


Figure 6.6: Parallel I/O: (a) adding parallel I/O to a bus-based I/O processor, (b) extended parallel I/O.

In *bus-based I/O*, the microprocessor has a set of address, data, and control ports corresponding to bus lines, and uses the bus to access memory as well as peripherals. The microprocessor has the bus protocol built in to its hardware. Specifically, the software does not implement the protocol but merely executes a single instruction that in turn causes the hardware to write or read data over the bus. We normally consider the access to the peripherals as I/O, but don't normally consider the access to memory as I/O, since the memory is considered more as a part of the microprocessor.

A system may require parallel I/O (port-based I/O), but a microprocessor may only support bus-based I/O. In this case, a parallel I/O peripheral may be used, as illustrated in Figure 6.6(a). The peripheral is connected to the system bus on one side, with corresponding address, data, and control lines, and has several ports on the other side, consisting just of a set of data lines. The ports are connected to registers inside the peripheral, and the microprocessor can read and write those registers in order to read and write the ports.

Even when a microprocessor supports port-based I/O, we may require more ports than are available. In this case, a parallel I/O peripheral can again be used, as illustrated in Figure 6.6(b). The microprocessor has four ports in this example, one of which is used to interface with a parallel I/O peripheral, which itself has three ports. Thus, we have extended the number of available ports from four to six. Using such a peripheral in this manner is often referred to as *extended parallel I/O*.

Memory-Mapped I/O and Standard I/O

In bus-based I/O, there are two methods for a microprocessor to communicate with peripherals, known as *memory-mapped I/O* and *standard I/O*.

In *memory-mapped I/O*, peripherals occupy specific addresses in the existing address space. For example, consider a bus with a 16-bit address. The lower 32K addresses may correspond to memory addresses, while the upper 32K may correspond to I/O addresses.

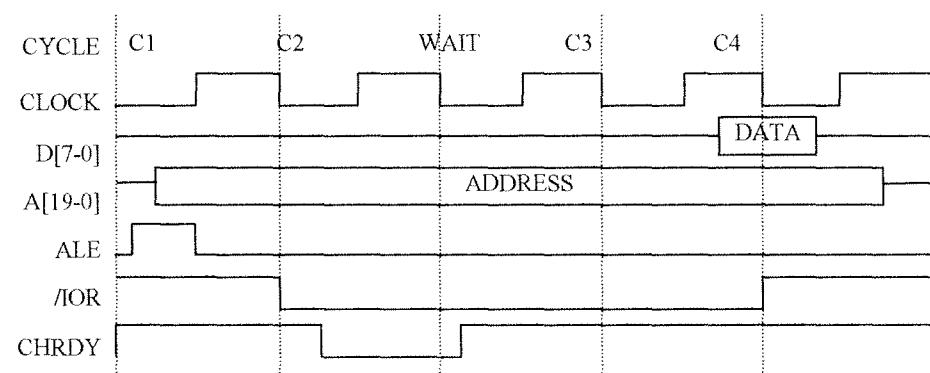


Figure 6.7: ISA bus protocol for standard I/O.

In *standard I/O* (also known as *I/O-mapped I/O*), the bus includes an additional pin, which we label *M/I/O*, to indicate whether the access is to memory or to a peripheral (i.e., an I/O device). For example, when *M/I/O* is 0, the address on the address bus corresponds to a memory address. When *M/I/O* is 1, the address corresponds to a peripheral.

An advantage of memory-mapped I/O is that the microprocessor need not include special instructions for communicating with peripherals. The microprocessor's assembly instructions involving memory, such as MOV or ADD, will also work for peripherals. For example, a microprocessor may have an ADD A, B instruction that adds the data at address B to the data at address A and stores the result in A. A and B may correspond to memory locations, or registers in peripherals. In contrast, if the microprocessor uses standard I/O, the microprocessor requires special instructions for reading and writing peripherals. These instructions are often called IN and OUT. Thus, to perform the same addition of locations A and B corresponding to peripherals, the following instructions would be necessary:

```
IN R0, A
IN R1, B
ADD R0, R1
OUT A, R0
```

Advantages of standard I/O include no loss of memory addresses to the use as I/O addresses, and potentially simpler address decoding logic in peripherals. Address decoding logic can be simplified with standard I/O if we know that there will only be a small number of peripherals, because the peripherals can then ignore high-order address bits. For example, a bus may have a 16-bit address, but we may know there will never be more than 256 I/O addresses required. The peripherals can thus safely ignore the high-order 8 address bits, resulting in smaller and/or faster address comparators in each peripheral. Note that we can build a system using both standard and memory-mapped I/O, since peripherals in the memory space act just like memory themselves.

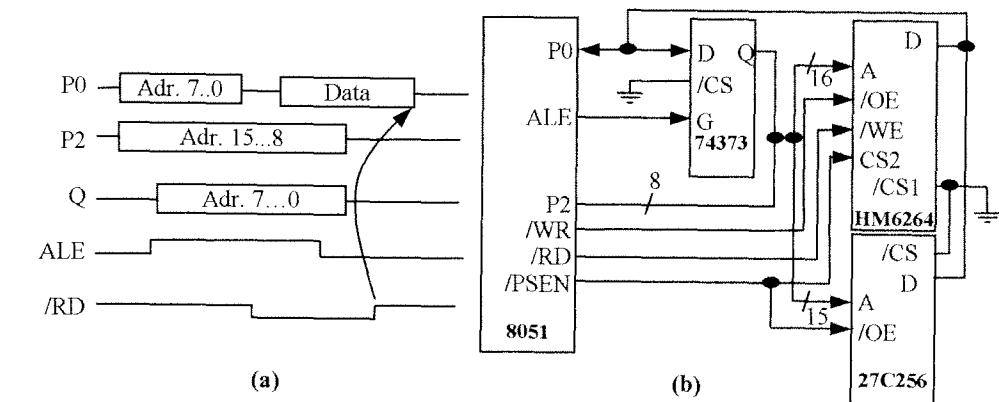


Figure 6.8: A basic memory protocol: (a) timing diagram for a read operation, (b) interface schematic.

Example: The ISA Bus Protocol — Standard I/O

The ISA bus protocol introduced earlier supports standard I/O. The I/O read bus cycle is depicted in Figure 6.7. During this bus cycle, the microprocessor drives the bus signals to read a byte of data from a peripheral, according to the timing diagram shown. Note that the cycle uses a control line distinct from */MEMR*, namely */IOR*, which is consistent with the standard I/O approach. The I/O device address space is limited to 16 bits, as opposed to 20 bits for memory devices. The I/O write bus cycle is similar to the memory write bus cycle but uses a control signal */IOW* and again limits the address to 16 bits. The I/O read and write bus cycles use the compromise strobe/handshake control method, as did the memory bus cycles.

Example: A Basic Memory Protocol

In this example, we illustrate how to interface 8K of data and 32K of program code memory to a microcontroller, specifically the Intel 8051. The 8051 uses separate memory address spaces for data and program code. Data or code address space is limited to 64K, hence, addressable with 16 bits through ports *P0* (least significant bits) and *P2* (most significant bits). A separate signal, called *PSEN* (program strobe enable), is used to distinguish between data/code. For the most part, the 8051 generates all of the necessary signals to perform memory I/O, however, since port *P0* is used both for the least significant address bits and for data, an 8-bit latch is required to perform the necessary multiplexing. The timing diagram depicted in Figure 6.8(a) illustrates a memory read operation. A memory write operation is performed in a similar fashion with data flow reversed and *RD* (read) replaced with *WR* (write). The memory read operation proceeds as follows. The microcontroller places the source address (i.e., the memory location to be read) on ports *P2* and *P0*. *P2*, holding the eight most significant address bits, retains its value throughout the read operation. *P0*, holding the eight least-significant address bits, is stored inside an 8-bit latch. The *ALE* signal (address

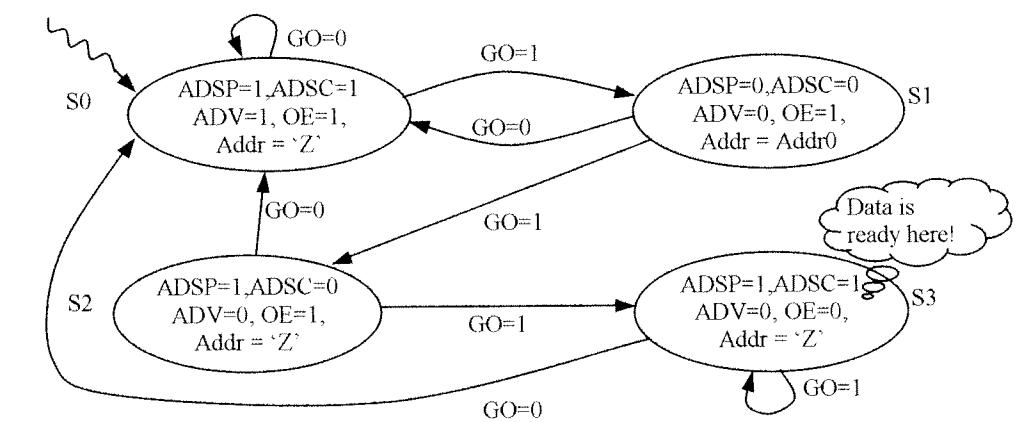


Figure 6.9: A complex memory protocol.

latch enable) is used to trigger the latching of port P_0 . Now, the microcontroller asserts high impedance on P_0 to allow the memory device to drive it with the requested data. The memory device outputs valid data as long as the RD signal is asserted. Meanwhile, the microcontroller reads the data and deasserts its control and port signals. Figure 6.8(b) illustrates the interface schematic.

Example: A Complex Memory Protocol

In this example, we will build a finite-state machine (FSM) controller that will generate all the necessary control signals to drive the TC55V2325FF memory chip in burst read mode (i.e., pipelined read operation), as described in Chapter 5. Our specification for this FSM is the timing diagram presented in the earlier example from Chapter 5. The input to our machine is a clock signal (CLK), the starting address ($Addr_0$) and the enable/disable signal (GO). The output of our machine is a set of control signals specific to our memory device. We assume that the chip's *enable* and *WE* signals are asserted. Figure 6.9 gives the FSM description. From the state machine description, we can derive the next-state and output truth tables. From these truth tables, we can compute next-state and output equations. By deriving the next-state transition table, we can solve and optimize the next-state and output equations. These equations can be implemented using logic components. (See Chapter 2 for details.) Any processor that is to be interfaced with one of these memory devices must implement, internally or externally, a state machine similar to the one presented in this example.

6.4 Microprocessor Interfacing: Interrupts

Another microprocessor I/O issue is that of interrupt-driven I/O. To introduce this issue, suppose the program running on a microprocessor must, among other tasks, read and process

data from a peripheral whenever that peripheral has new data; such processing is called *servicing*. If the peripheral gets new data at unpredictable intervals, how can the program determine when the peripheral has new data? The most straightforward approach is to interleave the microprocessor's other tasks with a routine that checks for new data in the peripheral, perhaps by checking for a 1 in a particular bit in a register of the peripheral. This repeated checking by the microprocessor for data is called *polling*. Polling is simple to implement, but this repeated checking wastes many clock cycles, so it may not be acceptable in many cases, especially when there are numerous peripherals to be checked. We could check at less-frequent intervals, but then we may not process the data quickly enough.

To overcome the limitations of polling, most microprocessors come with a feature called *external interrupt*. A microprocessor with this feature has a pin, say, *Int*. At the end of executing each machine instruction, the processor's controller checks *Int*. If *Int* is asserted, the microprocessor jumps to a particular address at which a subroutine exists that services the interrupt. This subroutine is called an *interrupt service routine*, or ISR. Such I/O is called *interrupt-driven I/O*.

One might wonder if interrupts have really solved the problem with polling, namely of wasting time performing excessive checking, since the interrupt pin is “polled” at the end of every microprocessor instruction. However, in this case, the polling of the pin is built right into the microprocessor's controller hardware, and therefore can be done simultaneously with the execution of an instruction, resulting in no extra clock cycles.

There are two methods by which a microprocessor using interrupts determines the address, known as the *interrupt address vector*, at which the ISR resides. These two methods are *fixed interrupt* and *vectored interrupt*. In *fixed interrupt*, the address to which the microprocessor jumps on an interrupt is built into the microprocessor, so it is fixed and cannot be changed. The assembly programmer either puts the ISR at that address, or if not enough bytes are available in that region of memory, merely puts a jump to the real ISR there. For C programmers, the compiler typically reserves a special name for the ISR and then compiles a subroutine having that name into the ISR location, or again just a jump to that subroutine. In microprocessors with fixed ISR addresses, there may be several interrupt pins to support interrupts from multiple peripherals.

Figure 6.10 provides a summary of the flow of actions for an example of interrupt-driven I/O using a fixed ISR address. Figure 6.11 illustrates this flow graphically for the example. In this example, data received by *Peripheral1* must be read, transformed, and then written to *Peripheral2*. *Peripheral1* might represent a sensor, and *Peripheral2*, a display. Meanwhile, the microprocessor is running its main program, located in program memory starting at address 100. When *Peripheral1* receives data, it asserts *Int* to request that the microprocessor service the data. After the microprocessor completes execution of its current instruction, it stores its state and jumps to the ISR located at the fixed program memory location of 16. The ISR reads the data from *Peripheral1*, transforms it, and writes the result to *Peripheral2*. The last ISR instruction is a return from interrupt, causing the microprocessor to restore its state and resume execution of its main program, in this case executing instruction 101.

Other microprocessors use *vectored interrupt* to determine the address at which the ISR resides. This approach is especially common in systems with a system bus, since there may be

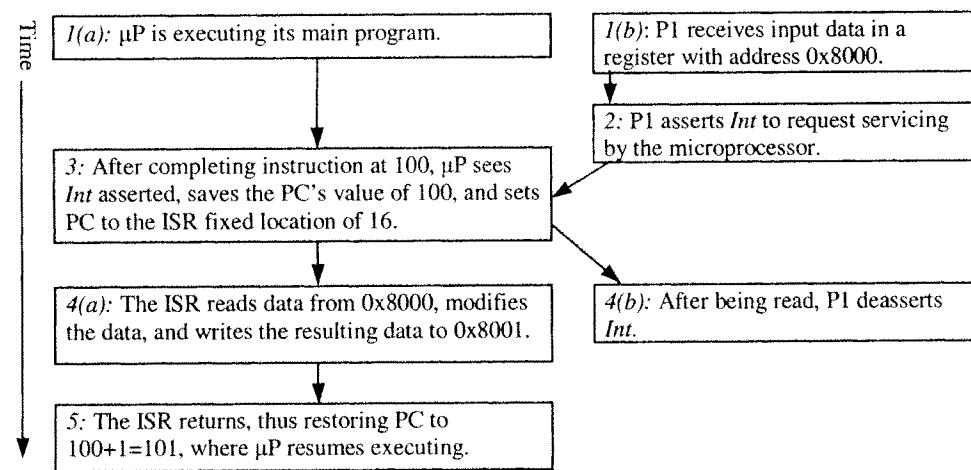


Figure 6.10: Interrupt-driven I/O using fixed ISR location: summary of flow of actions.

numerous peripherals that can request service. In this method, the microprocessor has one interrupt pin, say, *Int*, which any peripheral can assert. After detecting the interrupt, the microprocessor asserts another pin, say, *Inta*, to acknowledge that it has detected the interrupt and to request that the interrupting peripheral provide the address where the relevant ISR resides. The peripheral provides this address on the data bus, and the microprocessor reads the address and jumps to the corresponding ISR. We discuss the situation where multiple peripherals simultaneously request servicing in a later section on arbitration. For now, consider an example of one peripheral using vectored interrupt. The flow of actions is shown in Figure 6.12, which represents an example very similar to the previous one. Figure 6.13 illustrates the example graphically. In contrast to the earlier example, the ISR location is not fixed at 16. Thus, *Peripheral1* contains an extra register holding the ISR location. After detecting the interrupt and saving its state, the microprocessor asserts *Inta* in order to get *Peripheral1* to place 16 on the data bus. The microprocessor reads this 16 into the PC and then jumps to the ISR, which executes and completes in the same manner as the earlier example.

As a compromise between the fixed and vectored interrupt methods, we can use an *interrupt address table*. In this method, we still have only one interrupt pin on the processor, but we also create in the processor's memory a table that holds ISR addresses. A typical table might have 256 entries. A peripheral, rather than providing the ISR address, instead provides a number corresponding to an entry in the table. The processor reads this entry number from the bus, and then reads the corresponding table entry to obtain the ISR address. Compared to the entire memory, the table is typically very small, so an entry number's bit encoding is small. This small bit encoding is especially important when the data bus is not wide enough to hold a complete ISR address. Furthermore, this approach allows us to assign each peripheral a

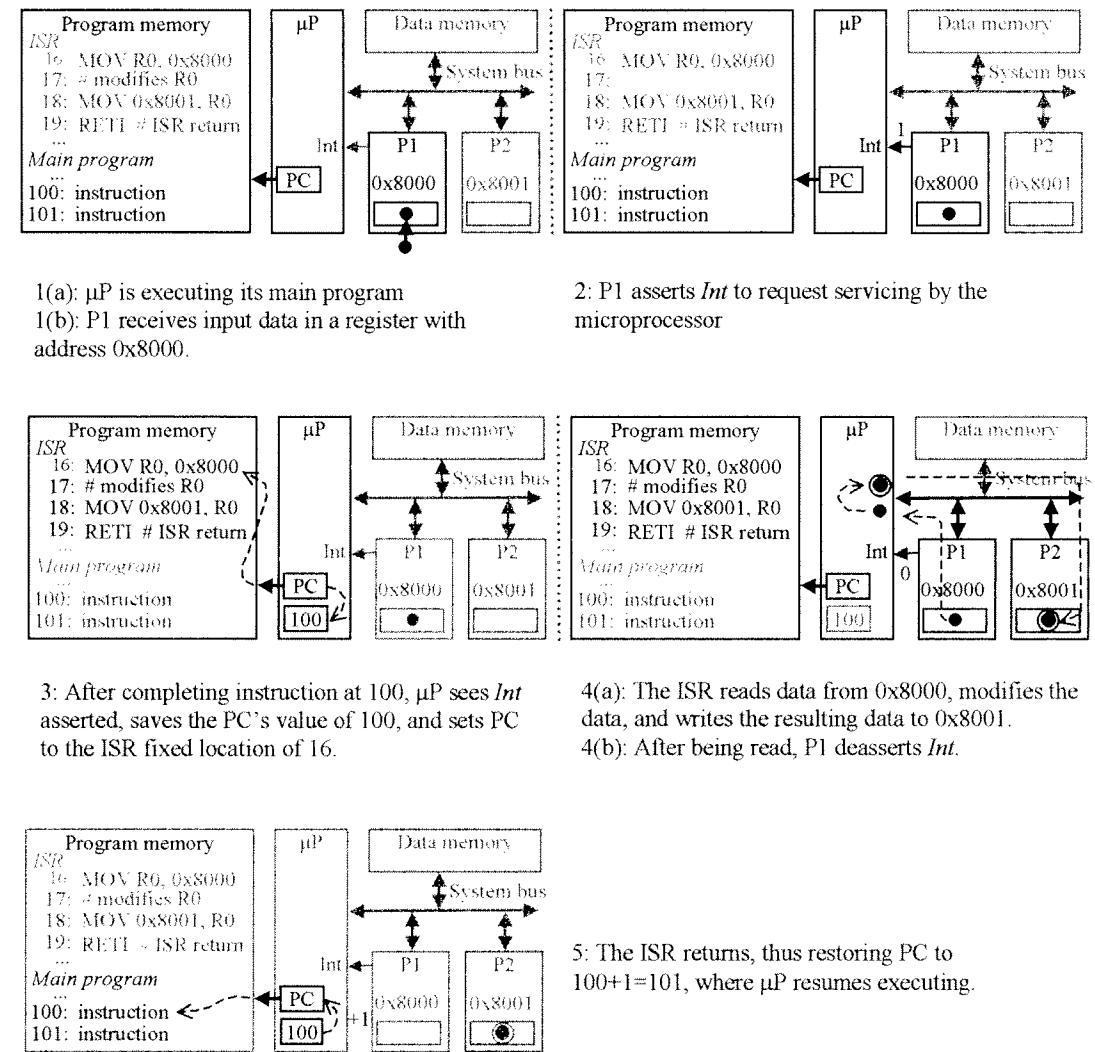


Figure 6.11: Interrupt-driven I/O using fixed ISR location: flow of actions.

unique number independent of ISR locations, meaning that we could move the ISR location without having to change anything in the peripheral.

External interrupts may be maskable or nonmaskable. In *maskable* interrupt, the programmer may force the microprocessor to ignore the interrupt pin, either by executing a specific instruction to disable the interrupt or by setting bits in an interrupt configuration register. A situation where a programmer might want to mask interrupts is when there exist time-critical regions of code, such as a routine that generates a pulse of a certain duration. The

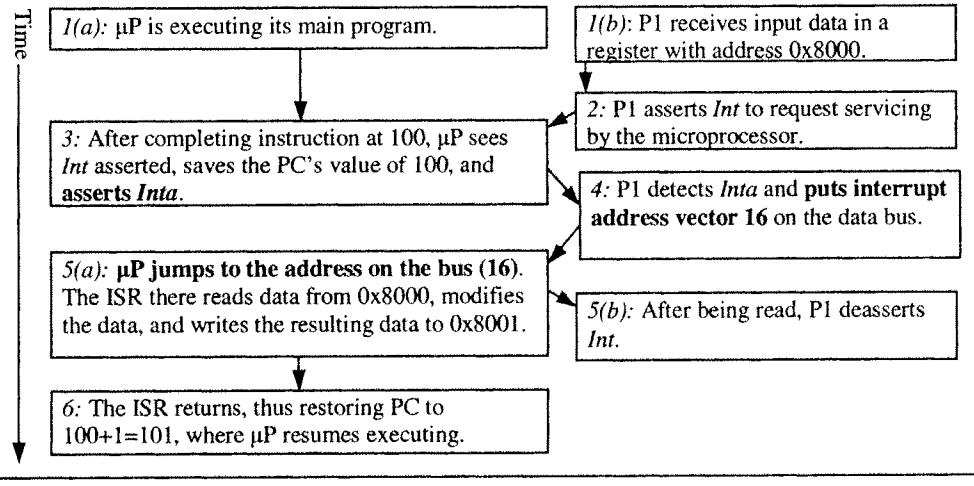


Figure 6.12: Interrupt-driven I/O using vectored interrupt: summary of flow of actions.

programmer may include an instruction that disables interrupts at the beginning of the routine, and another instruction reenabling interrupts at the end of the routine. *Nonmaskable* interrupt cannot be masked by the programmer. It requires a pin distinct from maskable interrupts. It is typically used for very drastic situations, such as power failure. In this case, if power is failing, a nonmaskable interrupt can cause a jump to a subroutine that stores critical data in nonvolatile memory, before power is completely gone.

In some microprocessors, the jump to an ISR is handled just like the jump to any other subroutine, meaning that the state of the microprocessor is stored on a stack, including contents of the program counter, datapath status register, and all other registers. The state is then restored upon completion of the ISR. In other microprocessors, only a few registers are stored, like just the program counter and status registers. The assembly programmer must be aware of what registers have been stored, so as not to overwrite nonstored register data with the ISR. These microprocessors need two types of assembly instructions for subroutine return. A regular return instruction returns from a regular subroutine, which was called using a subroutine call instruction. A return from interrupt instruction returns from an ISR, which was jumped to not by a call instruction but by the hardware itself, and which restores only those registers that were stored at the beginning of the interrupt. The C programmer is freed from having to worry about such considerations, as the C compiler handles them.

The reason we used the term *external interrupt* is to distinguish this type of interrupt from internal interrupts, also called *traps*. An internal interrupt results from an exceptional condition, such as divide-by-0, or execution of an invalid opcode. Internal interrupts, like external ones, result in a jump to an ISR. A third type of interrupt, called *software interrupts*, can be initiated by executing a special assembly instruction.

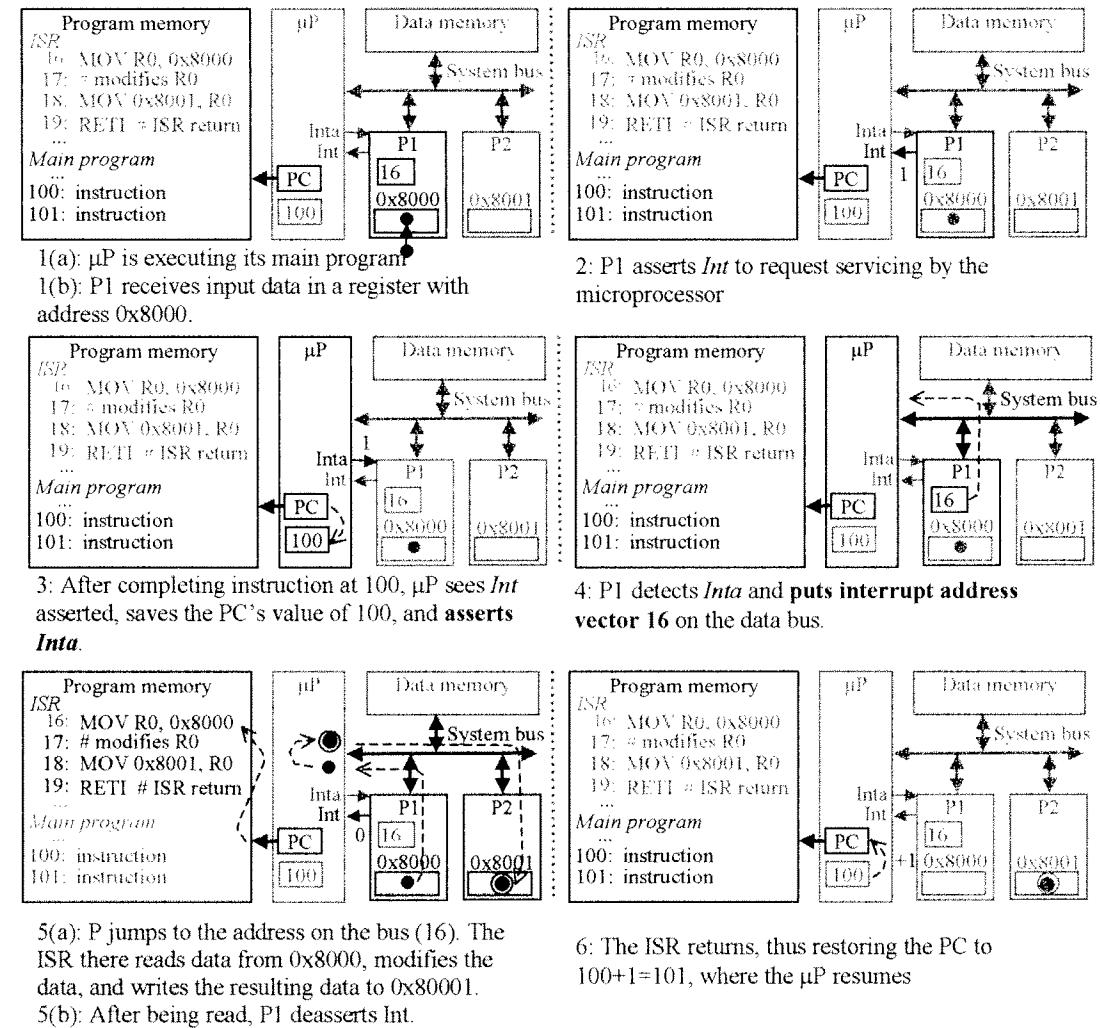


Figure 6.13: Interrupt-driven I/O using vectored interrupt: flow of actions.

6.5 Microprocessor Interfacing: Direct Memory Access

Commonly, the data being accumulated in a peripheral should be first stored in memory before being processed by a program running on the microprocessor. Such temporary storage of data that is awaiting processing is called *buffering*. For example, packet data from an

James K. Peckol

Embedded Systems Design, A Contemporary Design Tool

Wiley, ISBN 978-0-471-72180-2.

Chapter 6, Interfacing, pp 166-169

6.8 Advanced Communication Principles

In the preceding sections, we discussed basic methods of interfacing. Those interfacing methods could be applied to interconnect components within an IC via on-chip buses, or to interconnect ICs via on-board buses. In the remainder of the chapter, we study more advanced interfacing concepts and look at communication from a more abstract point of view. In particular, we study parallel, serial, and wireless communication. We also describe some advanced concepts, such as layering and error detection, which are part of many communication protocols. Furthermore, we highlight some of the popular parallel, serial, and wireless communication protocols in use today.

Communication can take place over a number of different types of media, such as a single wire, a set of wires, radio waves, or infrared waves. We refer to the medium that is used to carry data from one device to another as the *physical layer*. Depending on the protocol, we may refer to an actor as a *device* or *node*. In either case, a device is simply a processor that uses the physical layer to send or receive data to and from another device.

In this section, we provide a general description of serial communication, parallel communication, and wireless communication. In addition, we describe communication principles such as layering, error detection and correction, data security, and plug and play.

Parallel Communication

Parallel communication takes place when the physical layer is capable of carrying multiple bits of data from one device to another. This means that the data bus is composed of multiple data wires, in addition to control and possibly power wires, running in parallel from one device to another. Each wire carries one of the bits. Parallel communication has the advantage of high data throughput, if the length of the bus is short. The length of a parallel bus must be kept short because long parallel wires will result in high capacitance values, and transmitting a bit on a bus with a higher capacitance value will require more time to charge or discharge. In addition, small variations in the length of the individual wires of a parallel bus can cause the received bits of the data word to arrive at different times. Such misalignment of data becomes more of a problem as the length of a parallel bus increases. Another problem with parallel buses is the fact that they are more costly to construct and may be bulky, especially when considering the insulation that must be used to prevent the noise from each wire from interfering with the other wires. For example, a 32-wire cable connecting two devices together will cost much more and be larger than a two-wire cable.

In general, parallel communication is used when connecting devices that reside on the same IC, or devices that reside on the same circuit board. Since the length of such buses is short, the capacitance load, data misalignment and cost problems mentioned earlier do not play an important role.

Serial Communication

Serial communication involves a physical layer that carries one bit of data at a time. This means that the data bus is composed of a single data wire, along with control and possibly

power wires, running from one device to another. In serial communication, a word of data is transmitted one bit at a time. Serial buses are capable of higher throughputs than parallel buses when used to connect two physically distant devices. The reason for this is that a serial bus will have less average capacitance, enabling it to send more bits per unit of time. In addition, a serial bus cable is cheaper to build because it has fewer wires. The disadvantage of a serial bus is that the interfacing logic and communication protocol will be more complex. On the sending side, a transmitter must decompose data words into bits and on the receiving side, and the receiver must compose bits into words.

Most serial bus protocols eliminate the need for extra control signals, such as read and write signals, by using the same wire that carries data for this purpose. This is performed as follows. When data is to be sent, the sender first transmits a bit called a *start bit*. A start bit merely signals the receiver to wakeup and start receiving data. The start bit is then followed by N data bits, where N is the size of the word, and a *stop bit*. The stop bit signals to the receiver the end of the transmission. Often, both the transmitter and the receiver agree on the transmission speed used to send and receive data. After sending a start bit, the transmitter sends all N bits at the predetermined transmission speed. Likewise, on seeing a start bit, a receiver simply starts sampling the data at a predetermined frequency until all N bits are assembled. Another common synchronization technique is to use an additional wire for clocking purposes (see the I²C bus protocol). Here, the transmitter and receiver devices use this clock line to determine when to send or sample the data.

Wireless Communication

Wireless communication eliminates the need for devices to be physically connected in order to communicate. The physical layer used in wireless communication is typically either an infrared (IR) channel or a radio frequency (RF) channel.

Infrared uses electromagnetic wave frequencies that are just below the visible light spectrum, thus undetectable by the human eye. These waves can be generated by using an infrared diode and detected by using an infrared transistor. An infrared diode is similar to a red or green diode except that it emits infrared light. An infrared transistor is a transistor that conducts (i.e., allows current to flow from its source to its drain), when exposed to infrared light. A simple transmitter can send 1s by turning on its infrared diode and can send 0s by turning off its infrared diode. Correspondingly, a receiver will detect 1s when current flows through its infrared transistor and 0s otherwise. The advantage of using infrared communication is that it is relatively cheap to build transmitters and receivers. The disadvantage of using infrared is the need for line of sight between the transmitter and receiver, resulting in a very restricted communication range.

Radio frequency (RF) uses electromagnetic wave frequencies in the radio spectrum. A transmitter here will need to use analog circuitry and an antenna to transmit data. Likewise, a receiver will need to use an antenna and analog circuitry to receive data. One advantage of using RF is that, generally, a line of sight is not necessary and thus longer distance communication is possible. The range of communication is, of course, dependent on the transmission power used by the transmitter.

Typically, RF transmitters and receivers must agree on a specific frequency in order to send and receive data. Using *frequency hopping*, it is possible for the transmitter and receiver to communicate while constantly changing the transmission frequency. Of course, both devices must have a common understanding of the sequence for frequency hops. Frequency hopping allows more devices to share a fixed set of frequencies and is commonly used in wireless communication protocols designed for networks of computers and other electronic devices.

Layering

Layering is a hierarchical organization of a communication protocol where lower levels of the protocol provide services to the higher levels. We have already discussed the physical layer. The physical layer provides the basic service of sending and receiving bits or words of data. The next higher-level protocol uses the physical layer to send and receive packets of data, where a packet of data is composed of possibly multiple bytes. The next higher level uses the packet transmission service of its lower level to perhaps send different type of data such as acknowledgments, special requests, and so on. Typically, the lowest level consists of the physical layer and the highest level consists of the application layer. The application layer provides abstract services to the application such as ftp or http.

Layering is a way to break the complexity of a communication protocol into independent pieces, thus making it easier to design and understand, much like a programmer abstracting away complexities of a program by creating objects or libraries of routines. In communication and networking, the concept of layering is very fundamental.

Error Detection and Correction

Error detection is the ability of a receiver to detect errors that may occur during the transmission of a data word or packet. The most common types of errors are bit errors and burst of bit errors. A *bit error* occurs when a single bit in a word or data packet is received as its inverted value. A *burst of bit error* occurs when consecutive bits of a word or data packet are received incorrectly. Given that an error is detected, *error correction* is the ability of a receiver and transmitter to cooperate in order to correct the problem. The ability to detect and correct errors is often part of a bus protocol. We will next discuss parity and checksum error detection algorithms, which are commonly used in bus protocols.

Parity is a single bit of information that is sent along with a word of data by the transmitter to give the receiver some additional knowledge about the data word. This additional knowledge is used by the receiver to detect, to some degree, a bit or burst of bit error in receiving a word. Common types of parity are odd or even. Odd parity is a bit that if set indicates to the receiver that the data word bits plus parity bit contain an odd number of 1s. Even parity is a bit that if set indicates to the receiver that the data word bits plus parity bit contain an even number of 1s. Prior to sending a word of data, the transmitter will compute the parity and send that along with the data word to the receiver. On reception of the data word and parity bit, the receiver will compute the parity of the data and make sure that it agrees with the parity bit received from the transmitter. If a parity check fails, it indicates with

certainty that there was at least one transmission error. Parity checks will always detect a single bit error. However, burst bit errors may or may not be detected by parity checking — an even number of errors, for example, will not be detected.

As an example of parity-based error checking, consider wanting to transmit the following 7-bit word: 0101010. Assuming even parity, we would actually transmit the 8-bit word: 01010101, where the least-significant bit is the parity bit. Now, suppose during transmission, a bit gets flipped, so that a receiver receives the following 8-bit word: 11010101. The receiver detects that this word has odd parity; knowing that the word was supposed to have even parity, the receiver determines that this word has an error. Instead, suppose the receiver receives: 11110101. This word has even parity, and so the receiver thinks the word is correct, even though it contains two errors.

Checksum is a stronger form of error checking that is applied to a packet of data. A packet of data will contain multiple words of data. Using parity checking, we used one extra bit per word to help us detect errors. Using checksum, we use an extra word per packet for the same purpose. For example, we may compute the XOR sum of all the data words in a packet and send this value along with the data packet. Upon receiving the data packet words and the checksum word, the receiver will compute the XOR sum of all the data words it received. If the computed checksum word equals the received checksum word, the data packet is assumed to be correct. Otherwise, it is assumed to be incorrect. Again, not all error combinations can be detected. We can of course use both parity and checksum for stronger error checking.

As an example, suppose a packet consists of four words: 0000000, 0101010, 1010101, and 0000000. The XOR checksum of these four words is 1111111. A transmitter can thus send that checksum word at the end of the packet. Now, suppose the receiver receives 1000001, 0101010, 1010101, and 0000000. Note that two bits have switched in the first word, and that parity-based error checking would not detect this error. The receiver computes the checksum of this packet and obtains 0111110. This differs from the received checksum of 1111111, and thus the receiver determines that an error has occurred.

Note that errors can also occur in the parity bit or the checksum word itself.

When using parity or checksum error detection, error correction is typically done by a retransmission and acknowledgment protocol. Here, the transmitter sends a data packet and expects to receive an acknowledgment from the receiver indicating that the data packet was received correctly. If an acknowledgment is not received, the transmitter retransmits the data packet and waits for a second acknowledgment.

6.9 Serial Protocols

In this section, we describe four popular serial protocols, namely the I²C protocol, the CAN protocol, the FireWire protocol, and the USB protocol.

I²C

Philips Semiconductors developed the *Inter-IC*, or I²C, bus nearly 20 years ago. I²C is a two-wire serial bus protocol. This protocol enables peripheral ICs in electronic systems to

James K. Peckol

Embedded Systems Design, A Contemporary Design Tool

Wiley, ISBN 978-0-471-72180-2.

Chapter 16.7, Network Architecture, pp 627 - 637

*message
channel, port
Send, Receive*

As we begin to expand to greater distances, our thinking shifts to a network-based approach in which tasks can exchange information through messages. The sending process transmits a set of data values, the *message*, through the specified communication medium where it is accepted by the receiving process. The medium may be a *channel* or *port*. The basic supporting operations are *Send* and *Receive*.

The major differences between the remote and local device models are reflected in the details of the transport mechanism and the control and synchronization of the information exchange. Most remote intra- and intersystem communications within a distributed embedded system take place over a standard network using a serial scheme—EIA-232, I²C, Ethernet, USB, and so on.

Any modern automobile provides an excellent example of such a system. Processors throughout the vehicle manage everything from the fuel system to the passenger environment and entertainment systems. Internet and Internet appliances are further examples of contemporary distributed embedded applications.

As we move to networked systems, we introduce a new collection of opportunities and challenges. One of the basic goals is to ensure that the underlying architecture is invisible to the tasks comprising the application. That is, from any task's perspective, interaction with other tasks should not depend on where the tasks are physically located or on the computing engine on which each is executing. Furthermore, we want to be able to exchange information with any part of the system both easily and seamlessly. The notion of highly cohesive, loosely coupled modules that was introduced earlier continues now with movement outside of the processor.

Challenges arise because of the very nature of the distributed system. With such systems, the possibility of local failures now exists. The system can experience hardware or software failures on any of the distributed portions of the application while the remainder of the system continues to operate. The designs must be tolerant of such failures. The inter-process communication and synchronization problem is also exacerbated. As the application becomes increasingly distributed, the communication delays become longer and may become nondeterministic. The need to meet hard real-time constraints thus remains, increasing the complexity of any analysis and modeling of such problems.

16.6.1 Places and Information

places, information

Figure 16.39 gave a high-level architecture for the remote device model. In the model, the identity of the *places* where the *information* is to be written to or read from and the *information* itself are embodied in messages exchanged over the remote network structure. The format of the message can be implemented in a variety of ways depending on the nature and structure of the underlying protocol and supporting networks.

16.6.2 Control and Synchronization

*control and
synchronization*

The *control and synchronization* strategy is incorporated into the protocol by which the messages are exchanged.

16.6.3 Transport

transport

The physical *transport* of the information between the system core and the remote external devices can utilize any of the models introduced earlier in this chapter. Today, copper wire remains the medium of choice; however, fiber and air are gaining widespread support. Within programmable logic devices or networks on a chip, it is a silicon path.

16.7 IMPLEMENTING THE REMOTE DEVICE MODEL—A FIRST STEP

We will now begin our study of the message exchange portion of the remote device model. Central to any such communications between electronic devices is the protocol for transmitting and receiving the information or message. We will start with the transport level. Message exchange in distributed embedded applications occurs either via a proprietary network or one implemented according to one of the many standards. Typically, the transport topology is serial, and information flow is full duplex. Whether it utilizes a proprietary or standard topology, the typical transport architecture comprises a hierarchy of virtual networks.

Above the physical portion of the transport mechanism will be a varying number of software layers or levels as illustrated in Figure 16.40. The function at each level or layer on one machine interacts with the corresponding function at the same level on the second machine.

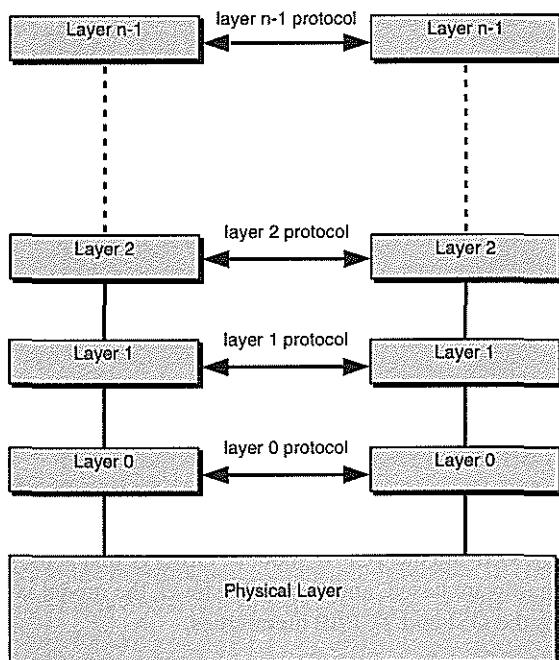


Figure 16.40 An N Layer Network Architecture

protocol

*service provider,
service consumer*

*network architecture
protocol stack, message*

At each level, potentially a different language, referred to as a *protocol*, is spoken. The function at each level is to provide services for the level above. Thus, between levels, we have the relationship of a *service provider* and a *service consumer*. At each level, the protocols may be implemented in either hardware or software. Typically, the lower levels are done in hardware and the upper in software.

The entire collection is called a *network architecture*, and the set of protocols used is called a *protocol stack*. The information sent on each level is called a *message*. It is possible that a message on a higher level is composed of several lower level messages. Synchronization between or among distributed processes is accomplished through such a message exchange.

Today, a wide variety of protocol standards are available. Although a proprietary network and protocol must sometimes be used, given a choice one should opt for one of the standards. The general objective of each standard is to facilitate message exchange in a specific application context such as small computer networks (EIA-232 or USB), simple local area networks (Firewire, Bluetooth, I²C), automotive networks (CAN bus), or manufacturing environments (CAMAC). Though unique to their particular context, most of these models trace their ancestry to two major protocol schemes or stacks, *OSI* and *TCP/IP*.

16.7.1 The OSI and TCP/IP Protocol Stacks

*Open Systems
Interconnection Model (OSI)
Transmission Control
Protocol/Internet Protocol
(TCP/IP),
physical, data link,
host to network*

The *Open Systems Interconnection model (OSI)* was proposed and developed by the International Standards Organization (ISO). The OSI protocol specifies a seven-layer virtual machine. The *Transmission Control Protocol/Internet Protocol (TCP/IP)*, comprises a five-layer virtual machine. We compare the two in Figure 16.41. The *physical* and *data link* layers of OSI are combined into the *host to network* layer in TCP/IP.

OSI	TCP / IP
Physical	Host to Network
Data Link	
Network	Internet
Transport	Transport
Session	Not Present
Presentation	Not Present
Application	Application

Figure 16.41 Layer in the Network Architectures for the OSI and TCO/IP Models

The following diagrams present and compare the hierarchical architecture and layers for the OSI and the TCP/IP models. Observe that at the network layer and below, the models are hardware based, and above that level, the model is expressed in software. Figure 16.42 gives the OSI and TCP/IP hierarchies.

OSI—Physical Layer

bits The physical layer moves collections of *bits* (1's and 0's) over a communications channel. There is no meaning or structure to the collections. At this level, concern is for mechanical and electrical interfaces, the integrity of bits, and the physical characteristics of the bits. Such characteristics include the number of volts and the width (in time) of each bit. Control issues address how a connection is established and released.

OSI—Data Link Layer

frames, data frames, acknowledgment frame The data link layer moves collections of bits aggregated as *frames*. The sender breaks the data stream into *data frames*. The receiver acknowledges reception via an *acknowledgment*

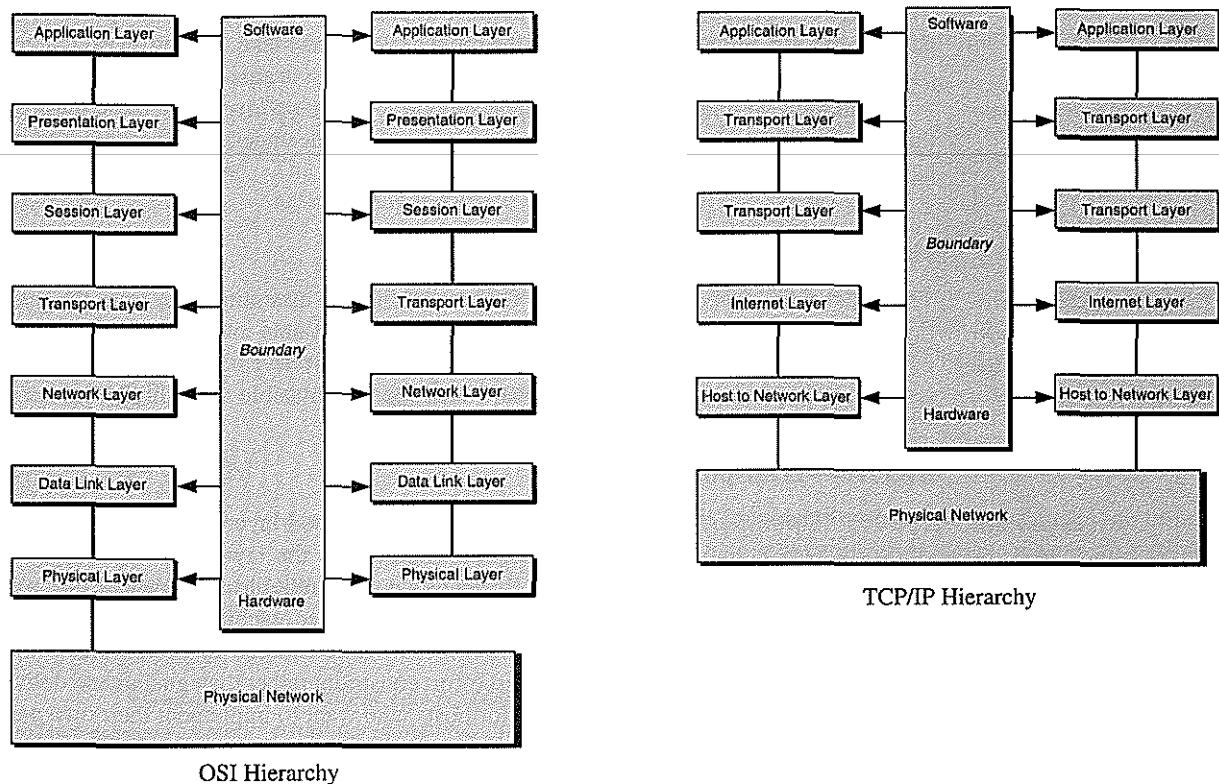


Figure 16.42 The Network Architectures for the OSI and TCO/IP Models

frame. The data link layer must create and recognize frame boundaries, which is facilitated by surrounding a frame with delimiters. The data link layer also manages flow control and some error management.

TCP/IP—Host to Network

No significant requirements are specified at this level. The host system must simply be able to connect to the network and to transmit or receive IP (Internet protocol) packets.

network layer, Internet

The OSI *network layer* corresponds to the TCP/IP *Internet layer*.

OSI—Network Layer

network layer

In the OSI stack, the *network layer* manages the routing of a transmission from the source to the destination. As part of that task, it must accommodate the different characteristics between or among networks such as addressing, message size, and protocols.

Activities are directed toward managing the network and the physical movement of data; bits are collected into manageable packets. Above the network layer is a collection of virtual machines that have the responsibility for managing the session.

TCP/IP—Internet Layer

*internet layer
IP, Internet Protocol*

The *Internet layer* is the key element of the TCP/IP model. It defines the official packet format and protocol, the *IP* or *Internet Protocol*. The main task of the Internet layer is to move

a message comprised of packets from point A to point B. No requirement is placed on the packet ordering during transmission or the route a packet may take. During an exchange, a packets may or may not take the same route from source to destination.

transport layer

Both the OSI and the TCP/IP models support a *transport layer*. The basic purpose of this layer in either model is to isolate the application from the underlying mechanics of managing the network below.

OSI—Transport Layer

transport layer, session layer network layer

The tasks of the *transport layer* in the OSI model include accepting data from the *session layer* that is immediately above and then subdividing that data into packets that are compatible with the *network layer* below. The fundamental objective is to ensure that the transactions are implemented such that the hardware appears invisible to the higher layers.

TCP/IP—Transport Layer

transport layer TCP, UDP Transmission Control Protocol UDP—User Datagram Protocol request-response, client-server

The TCP/IP *transport layer* is equivalent to the OSI transport layer and has similar responsibilities. Two communication protocols are defined for the layer: *TCP* and *UDP*. The former, *TCP—Transmission Control Protocol*—is very reliable and ensures that a data stream originating on one machine is delivered to any other machine on the network.

The latter, *UDP—User Datagram Protocol*—is considered to be unreliable in the sense that message delivery is under a best effort constraint rather than guaranteed delivery as is found with TCP. UDP is designed for hosts who want to implement their own packet sequencing and flow control. It finds application in *request-response* and *client-server* type applications in which speed is traded for accuracy.

OSI—Session Layer

session layer must dos offers to do

In the OSI model, the *session layer* permits users on different machines to communicate. Like the transport layer, it supports movement of data between machines. However, it offers a richer set of features and capabilities. At the session layer, we are moving from *must dos* to *offers to do*. The layer manages dialog control; for single-direction transmission, it tracks turns to send and manage tokens in token passing protocols. The session layer also synchronizes transaction and reassembles the message if necessary. Such cases occur if the transfer cannot be completed in a single session or if there is a major error such as a line drop or node crash.

OSI—Presentation Layer

presentation layer

The goal of the *presentation layer* is to offer a generic set of solutions to common problems. Potential services include mapping the information, including types, structures, and encoding, from the source computer representation to the network representation and then from the network representation to the destination representation. We will discuss this process in greater detail shortly.

The *session* and *presentation layers* in the OSI stack have no counterpart in the TCP/IP model. Both models support the top level, the *application layer*.

session, presentation layers application layer

OSI—Application Layer

This level also deals with incompatibilities between systems at opposite ends of the network. Although there is some accommodation for hardware differences, the primary focus

is on the software. Potential incompatibilities important to embedded applications include file systems and remote procedure execution. We will discuss remote procedures shortly.

TCP/IP—Application Layer

application layer

The *application layer* on the TCP/IP model duplicates most of the responsibilities that we have already identified in the OSI model.

In any distributed embedded design, understanding either of the base models facilitates the understanding of protocols derived therefrom. When one elects to communicate using any of the common standards, generally one integrates a commercially available protocol stack rather than choosing a proprietary design. It is also important to recognize that the OSI hierarchy is a model on which other protocols may be built. One typically does not find a specific implementation in practice.

16.7.2 The Models

*client-server, peer-to-peer
group multicast*

Most distributed embedded systems will implement a message-based exchange utilizing any of three patterns of communication and synchronization, *client-server*, *peer-to-peer*, and *group multicast*.

*client-server
producer-consumer, server*

client, request-reply

16.7.2.1 The Client-Server Model

The *client-server* model, shown in the state diagram in Figure 16.43, closely parallels the *producer-consumer* model we studied earlier. The model assumes that a *server* (or set of servers, analogous to the producer) exists that is able to provide some service required by some *client*. The client-server pattern involves the exchange of *request-reply* messages according to the following sequence.

1. Client: Transmit a request to the server process and block.
2. Server: Execute the request.
3. Server: Return the reply to the client.

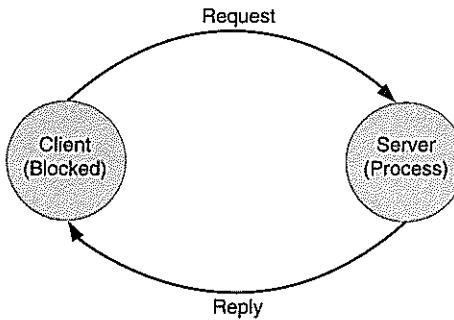


Figure 16.43 The Client-Server Model for Message-Based Exchange

remote procedure call

The server process, analogous to the producer process, is aware of the message as soon as it arrives. Activity in the sending process suspends or blocks until the reply is received, thereby providing a form of synchronization. The process is commonly represented at the language level as a *remote procedure call*, which thus hides the underlying communication operations—the invisibility of the infrastructure that was required earlier.

At the logical or functional level, the exchange appears to be directly between the client and the server processes. In reality, the interchange is managed by the local kernel. Signals move from or to the respective software drivers and to and from the physical network.

16.7.2.2 The Peer-to-Peer Model

peer-to-peer model The *peer-to-peer model* follows naturally from the client-server model. In the model, several (peer or equal) processes cooperate to solve a problem or to share information. The notion of a predesignated client or server does not exist; rather, any member of the network may request a service from or provide one to any of the others. Such an approach can remove a potential bottleneck that arises in the client-server model when a number of clients must interact with a single server at the same time. The peer-to-peer model permits several nodes to provide the requisite services. Synchronization is achieved through the message exchange as was done in the client-server model.

A portion of the architecture of such a system appears in Figure 16.44 as a minor modification of that for the client server.

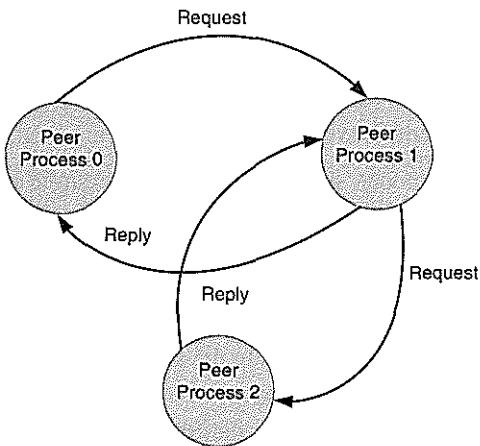


Figure 16.44 The Peer-to-Peer Model for Message-Based Exchange

16.7.2.3 The Group Multicast Model

group multicast The *group multicast* model, shown in Figure 16.45, comprises a single sender and multiple receivers. Such a scheme is used when it is necessary to pass information to all nodes within the network. We may use such a scheme to force all nodes into a known state, to initiate a

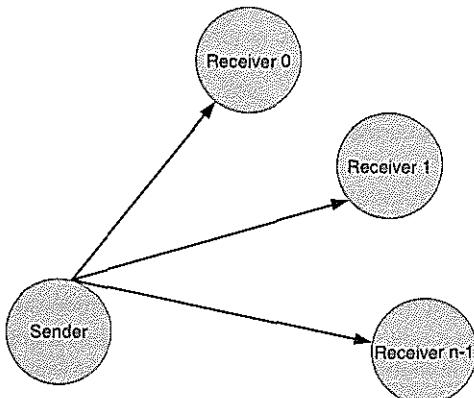


Figure 16.45 The Group Multicast Model for Message-Based Exchange

systemwide self-test, or to locate an object or service. In the last case, for example, the name of the desired resource or service might be multicast to a group of server processes, the one that holds the resource or that can provide the service responds. Similar ideas underlie Sun's Jini architecture. The USB uses a group multicast to require all nodes to "disconnect" from the network and to listen at a known address as an initial step in the enumeration process.

The approach can be inherently fault tolerant. The same task can be multicast to multiple servers; if one fails, the task can still continue. Multicast also works well if the same information is to be sent to a group of interested processes.

The multicast is not mutually exclusive with the other communication and synchronization patterns. In a peer-to-peer network, for example, a multicast might be used by a node entering the network to announce its presence and the services it can provide.

16.8 IMPLEMENTING THE REMOTE DEVICE MODEL—A SECOND STEP

We will now examine the client-server and group multicast communication schemes in greater detail. Peer to peer follows from client-server. We begin with the client-server model by taking a look at the fundamental components of such systems, including the underlying data structures and the messages.

16.8.1 The Messages

When designing and implementing distributed embedded systems, one quickly discovers that remote operations make up a substantial proportion of the interactions between processes. Such operations are initiated by one process sending a request message to another process. The receiving process responds with an acknowledgment or a reply indicating that the operation has been or will be carried out.

Viewed from the most abstract level, a message is simply a collection of bits, as we see in Figure 16.46, and the exchange is the movement of those bits from one place to another. To make the design of message-based communication tractable, rules are applied to the exchange and to the interpretation of the bits.

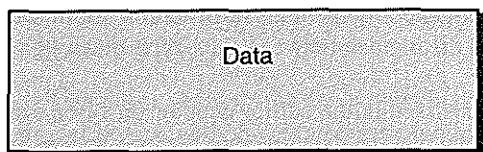


Figure 16.46 An Abstraction of the Basic Message

16.8.2 The Message Structure

*data, payload, header information
addressing control and synchronization*

One such interpretation views others of the bits as *data* or *payload* and others as *header* information. The payload is the *information* being transported. Moving the payload from one place to another is the ultimate objective of sending the message. The header information facilitates that job and is added by the communication driver to provide *addressing* and *control and synchronization* information. The message now appears as in Figure 16.47.

Not all messages are the same size. They may be simple, occupying only a few words of memory, or they can be complex, comprising a large number of blocks of data. The design of the exchange process is cleaner and more robust if the bits are organized to ensure that fixed sized groupings are always transferred. Such groupings are variously referred to



Figure 16.47 A Header Added to the Basic Message

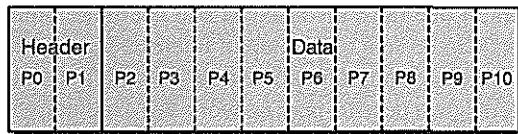


Figure 16.48 The Basic Message Decomposed into Packets

datagrams, packets as *datagrams* or *packets*. There are times when padding or fill bits must be included to ensure that the packets are the proper size if there is insufficient data available to complete the packet. Figure 16.48 now gives a logical view of the message that has been divided into packets.

16.8.3 Message Control and Synchronization

There are actually two kinds of control and synchronization in a message-based exchange: the header information and the data transfer scheme.

16.8.3.1 The Header

*start, end
start identifier, length field*

The header information is overhead that is necessary for getting the data from one place to another. This information is added (at each level within the protocol stack) by the communication driver software based on the requirements of the exchange protocol (at that level). Potential header elements might include the destination address or message identifier information. At minimum, this field identifies the destination for the message. If one is working in a networked context, the header might also provide routing information and identify both the sender and receiver of the message. The header field may also provide an indication of the size of the message. This is done in several ways: there may be unique *start* and *end* identifiers, or a *start identifier* and a *length field*.

The header generally includes information about the message type or structure. It may be desirable to distinguish between data-type messages and command-type messages, for example.

As was discussed in the earlier chapter on safety, an important element of communication is ensuring that the data given to the user following reception contains no errors that may have occurred during transmission. Note that there is no guarantee that there will never be transmission errors; these happen. Rather, at the end of the day, the guarantee is that, if passed to a task, the data will be correct. As discussed earlier, this is accomplished through a variety of schemes: all begin with recognizing that a transmission error has occurred. Thus, to this end, in the header field one might elect to include error management information as well. Such information may support detection only, for example, simple parity, or detection and correction with the inclusion of a block check or a CRC sequence.

16.8.3.2 The Transfer Scheme

*connection oriented,
connectionless*

The physical information transfer may follow a proprietary protocol or one of the standards or derivatives discussed earlier. Within such protocols, two kinds of services are identified: *connection oriented* and *connectionless*. A connection-oriented service establishes the con-

reliable circuit switching

nnection between the source and destination prior to the exchange of any data. The exchange follows, and the connection is terminated. Messages enter one end and are extracted from the other end; ordering is preserved. The exchange is designated as *reliable* because the reply is effectively an acknowledgment. Such a scheme is referred to as *circuit switching*. Each packet in Figure 16.49 will be sent, in turn, through the same physical path.

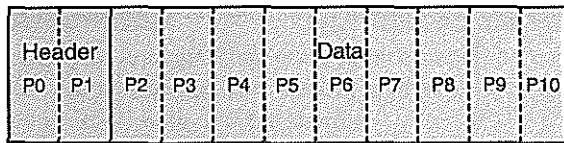


Figure 16.49 The Basic Message Decomposed into Packets

packet switching, best effort

A connectionless service does not establish a specific connection prior to the start of the exchange. Each packet carries full address information; each may arrive at the destination through any of several different routes and may not arrive in the same order in which it was sent. Address information has been added to each packet shown in Figure 16.50. Such a scheme is referred to as *packet switching*. A connectionless exchange is designated as *best effort*.



Figure 16.50 Address Information Added to Each Packet

datagram acknowledged datagram request-reply

Messages may be sent as a *datagram* service, in which the message is sent but the receiver does not acknowledge; as an *acknowledged datagram* service, in which the message is sent and the receiver acknowledges; or as a *request-reply* service, in which the sender transmits a datagram containing a request and the receiver returns a datagram containing the answer. This last-named scheme is often used in the client-server model.

16.9 WORKING WITH REMOTE TASKS

servers clients

The next step in examining the inner workings of message-based aspects of the remote device model begins with the client-server model. With this model, the processes (whether local or remote) either provide a service or request one. The former are the *servers* and the latter the *clients*. Any process may play either or both roles.

16.9.1 Preliminary Thoughts on Working with Remote Tasks

Before proceeding with the development of remote functionality, it is important to be aware of several major differences between local and remote tasks. Starting at a high level, the initial view of distributed client-server interaction is presented in the block diagram in Figure 16.51.

The client and the server each assume direct communication with the other. The communication link proceeds through the local kernel (client or server side) to the network and then to the remote node (server or client side), where the message is interpreted and passed to that local process.

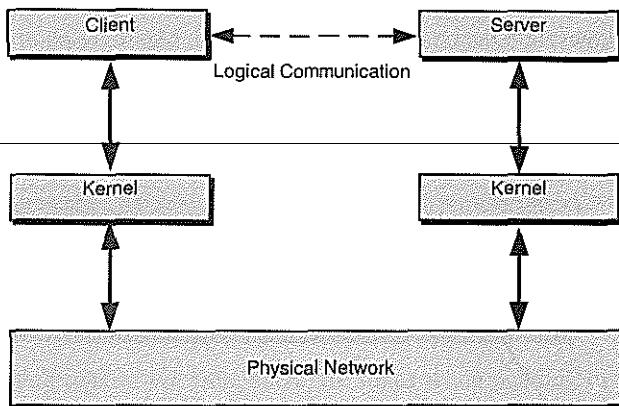


Figure 16.51 A High-Level View of a Client-Server Model

At this level, the implementation seems rather straightforward. Such is not always the case, however. Before taking the next step, let's anticipate some of the problems that might be encountered.

Local vs. Remote Addresses and Data

When working with distributed systems, one must remember that not all tasks are going to be using the same memory space or be on the same machine. The different addressing schemes and data formats must now be carefully considered. Data items in the programs are expressed as primitives such as arrays, structs, and classes, as well as richer, more complex structures built from these. In contrast, the information being exchanged in messages is (inherently) flat or sequential.

One must also remember that not all processors express data in the same way—different endianness or different word sizes are common. Such differences are evident in even simple elements such as integers. To permit the exchange of information among computers, one must ensure that at some level, the data values are expressed in an agreed upon common (external) form.

If all the components in an external system speak the same language, in the same way, there may be no need for conversion. More often, the outside world is made up of a heterogeneous collection of devices provided by a variety of different vendors. Under such conditions, communication and information interchange become more of a challenge. In such cases, part of establishing a connection may be negotiating a common language. Another alternative is to elect to communicate in some native form; such communication may have to include an architecture identifier.

Repeated Task Execution

When one is designing a distributed embedded application, one must consider the possibility that the request for an action or procedure invocation from a remote device may become corrupted and hence rejected. One must also address the possibility that the complete message may never be received. Consequently, the remote procedure may never be executed, be partially executed, or be completely executed.

Any of these alternatives may lead to serious safety problems. Although the complete execution of a requested remote procedure may seem innocuous, if confirmation that the task completed is not received or properly interpreted by the sender, additional requests may

be issued. If the request is of the form, “decrease flow rate of an inhibitor or increase the temperature of a process,” repeated requests may create serious safety problems. When designing the exchange, one must consider

- How to avoid such duplicate messages
- How to handle missed acknowledgments
- How to handle both the success and failure of an operation

*at most once
atomic transactions*

In an attempt to anticipate and manage such situations, contemporary distributed embedded applications incorporate what are called *at most once* semantics and *atomic transactions*. We will examine both of these shortly.

Node Failure, Link Failure, Message Loss

A distributed system is susceptible to a variety of failure modes not seen in the local model. One can generally detect failure, but often it is not possible to distinguish between a link failure, a node failure, or a message loss. Once a fault is detected, the appropriate action can be taken.

16.9.2 Procedures and Remote Procedures

procedure call

*remote procedure call (RPC)
remote procedure invocation*

(RPI)

*interface language processor
binding service
communication driver
request-reply protocol*

With these preliminary caveats in mind, we continue with the discussion. In a traditional software application, perhaps the most commonly used means for encapsulating a set of software instructions is the procedure. A *procedure call* is executed on the main processor by writing the name of the procedure followed by the associated parameters enclosed in parentheses. When that procedure resides in a remote address space, we would still like to be able to use similar semantics. Such an invocation is known as a *remote procedure call* (RPC). One may also see the terminology *remote procedure invocation* (RPI) used.

Remote procedure calls are similar to, yet different from, the familiar local procedure calls. Support for the remote call generally includes an *interface language processor*, a *binding service*, and a *communication driver*. The invocation is most commonly based on a *request-reply protocol*. The client invokes a service by sending request messages to the server. The server performs the requested service and sends a reply back to the client. Generally, the client waits for a reply before proceeding, analogous to a local call.

16.9.2.1 Calling a Remote Procedure—RPC Semantics

The remote procedure call (RPC) paradigm combines the familiar (local) procedure call model with the client-server model. The goal of the RPC model is to have tasks interact with local and remote procedures seamlessly.

When a local procedure is called, parameters and any return value are usually passed into and returned from that procedure via a stack. Following the call, the calling procedure then blocks waiting for the return. Such an approach is not possible with a remote procedure. Nonetheless, it is desirable that the remote call appear as if it had been local.

The first step in creating such an illusion is to write stubs for the procedure that are then placed on the client and server. These stubs have the same public interface—the same procedure name, return type, and signature—as the full procedure. The public interface masks the behind-the-scenes magic.

When the server process is ready, it will execute a blocking receive. When the client performs the call, the input parameters are passed to the server as values to arguments in a

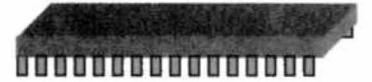
Frank Vahid/ Tony Givargis

Embedded System Design, A Unified Hardware/Software Introduction

Wiley, ISBN 0-471-38678-2.

Chapter 1, Embedded Systems Overview, pp 1 - 11

CHAPTER 1: *Introduction*



- 1.1 Embedded Systems Overview
 - 1.2 Design Challenge – Optimizing Design Metrics
 - 1.3 Processor Technology
 - 1.4 IC Technology
 - 1.5 Design Technology
 - 1.6 Tradeoffs
 - 1.7 Summary and Book Outline
 - 1.8 References and Further Reading
 - 1.9 Exercises
-

1.1 Embedded Systems Overview

Computing systems are everywhere. It is probably no surprise that millions of computing systems are built every year, destined for desktop computers like personal computers, laptops, workstations, mainframes, and servers. What may be surprising is that billions of computing systems are built every year for a very different purpose: They are embedded within larger electronic devices, repeatedly carrying out a particular function, often going completely unrecognized by the device's user. Creating a precise definition of such embedded computing systems, or simply *embedded systems*, is not an easy task. We might try the following definition: an embedded system is nearly any computing system other than a desktop computer. That definition isn't perfect, but it may be as close as we'll get. We can better understand such systems by examining common examples and common characteristics. Such examination will reveal major challenges facing designers of embedded systems.

Embedded systems are found in a variety of common electronic devices, such as consumer electronics (cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants), home appliances (microwave ovens, answering machines, thermostats, home security systems, washing machines, and lighting systems), office automation (fax machines, copiers, printers, and scanners), business equipment (cash registers, curbside check-in, alarm systems, card readers,

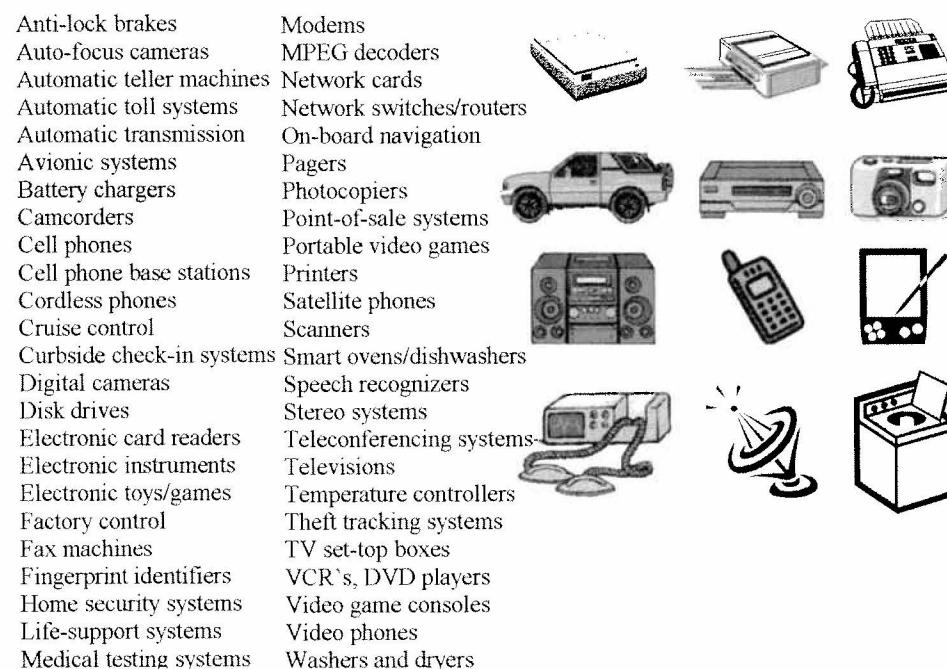


Figure 1.1: A short list of embedded systems.

product scanners, and automated teller machines), and automobiles (transmission control, cruise control, fuel injection, antilock brakes, and active suspension). Figure 1.1 is a short list of embedded system examples; a more complete list would require many pages. One might say that nearly any device that runs on electricity either already has or will soon have a computing system embedded within it. Although embedded computers typically cost far less than desktop computers, their quantities are huge. For example, in 1999 a typical American household may have had one desktop computer, but each one had between 35 and 50 embedded computers, with that number expected to rise to nearly 300 by 2004. Furthermore, the average 1998 car had 50 embedded computers costing several hundred dollars in all, with an annual cost growth rate of 17%. Several billion embedded microprocessor units were sold annually in recent years, compared to a few hundred million desktop microprocessor units.

Embedded systems have several common characteristics that distinguish such systems from other computing systems:

1. *Single-functioned*: An embedded system usually executes a specific program repeatedly. For example, a pager is always a pager. In contrast, a desktop system executes a variety of programs, like spreadsheets, word processors, and video games, with new programs added frequently. Of course, there are exceptions. One case is where an embedded system's program is updated with a newer program version. For example, some cell phones can be updated in such a manner. A second case is where

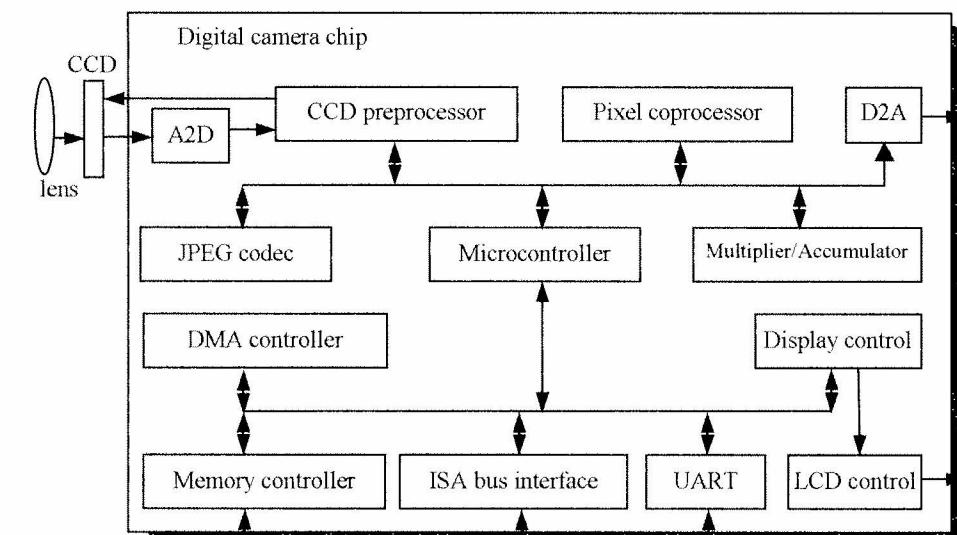


Figure 1.2: An embedded system example — a digital camera.

several programs are swapped in and out of a system due to size limitations. For example, some missiles run one program while in cruise mode, then load a second program for locking onto a target. Nevertheless, we can see that even these exceptions represent systems with a specific function.

2. *Tightly constrained*: All computing systems have constraints on design metrics, but those on embedded systems can be especially tight. A design metric is a measure of an implementation's features, such as cost, size, performance, and power. Embedded systems often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real time, and must consume minimum power to extend battery life or prevent the necessity of a cooling fan.
3. *Reactive and real time*: Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without delay. For example, a car's cruise controller continually monitors and reacts to speed and brake sensors. It must compute acceleration or deceleration amounts repeatedly within a limited time; a delayed computation could result in a failure to maintain control of the car. In contrast, a desktop system typically focuses on computations, with relatively infrequent (from the computer's perspective) reactions to input devices. In addition, a delay in those computations, while perhaps inconvenient to the computer user, typically does not result in a system failure.

For example, consider the digital camera chip shown in Figure 1.2. The charge-coupled device (*CCD*) contains an array of light-sensitive photocells that capture an image. The *A2D* and *D2A* circuits convert analog images to digital and digital to analog, respectively. The *CCD preprocessor* provides commands to the *CCD* to read the image. The *JPEG codec* compresses and decompresses an image using the *JPEG*¹ compression standard, enabling compact storage of images in the limited memory of the camera. The *Pixel coprocessor* aids in rapidly displaying images. The *Memory controller* controls access to a memory chip also found in the camera, while the *DMA controller* enables direct memory access by other devices while the microcontroller is performing other functions. The *UART* enables communication with a PC's serial port for uploading video frames, while the ISA bus interface enables a faster connection with a PC's ISA bus. The *LCD control* and *Display control* circuits control the display of images on the camera's liquid-crystal display device. The *Multiplier/Accumulator* circuit performs a particular frequently executed multiply/accumulate computation faster than the microcontroller could. At the heart of the system is the *Microcontroller*, which is a programmable processor that controls the activities of all the other circuits. We can think of each device as a processor designed for a particular task, while the microcontroller is a more general processor designed for general tasks.

This example illustrates some of the embedded system characteristics described earlier. First, it performs a single function repeatedly. The system always acts as a digital camera, wherein it captures, compresses, and stores frames, decompresses and displays frames, and uploads frames. Second, it is tightly constrained. The system must be low cost since consumers must be able to afford such a camera. It must be small so that it fits within a standard-sized camera. It must be fast so that it can process numerous images in milliseconds. It must consume little power so that the camera's battery will last a long time. However, this particular system does not possess a high degree of the characteristic of being reactive and real time, as it responds only to the pressing of buttons by a user, which, even in the case of an avid photographer, is still quite slow with respect to processor speeds.

1.2 Design Challenge — Optimizing Design Metrics

The embedded-system designer must of course construct an implementation that fulfills desired functionality, but a difficult challenge is to construct an implementation that simultaneously optimizes numerous design metrics.

Common Design Metrics

For our purposes, an implementation consists either of a microprocessor with an accompanying program, a connection of digital gates, or some combination thereof. A *design metric* is a measurable feature of a system's implementation. Commonly used metrics include:

¹ JPEG is short for *Joint Photographic Experts Group*. “Joint” refers to the group’s status as a committee working on both ISO and ITU-T standards. Their best-known standard is for still-image compression.

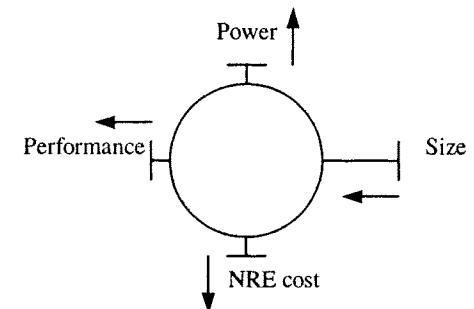


Figure 1.3: Design metric competition — improving one may worsen others.

- *NRE cost* (nonrecurring engineering cost): The one-time monetary cost of designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost; hence the term *nonrecurring*.
- *Unit cost*: The monetary cost of manufacturing each copy of the system, excluding NRE cost.
- *Size*: The physical space required by the system, often measured in bytes for software, and gates or transistors for hardware.
- *Performance*: The execution time of the system.
- *Power*: The amount of power consumed by the system, which may determine the lifetime of a battery, or the cooling requirements of the IC, since more power means more heat.
- *Flexibility*: The ability to change the functionality of the system without incurring heavy NRE cost. Software is typically considered very flexible.
- *Time-to-prototype*: The time needed to build a working version of the system, which may be bigger or more expensive than the final system implementation, but it can be used to verify the system's usefulness and correctness and to refine the system's functionality.
- *Time-to-market*: The time required to develop a system to the point that it can be released and sold to customers. The main contributors are design time, manufacturing time, and testing time.
- *Maintainability*: The ability to modify the system after its initial release, especially by designers who did not originally design the system.
- *Correctness*: Our confidence that we have implemented the system's functionality correctly. We can check the functionality throughout the process of designing the system, and we can insert test circuitry to check that manufacturing was correct.
- *Safety*: The probability that the system will not cause harm.

Metrics typically compete with one another: Improving one often leads to worsening of another. For example, if we reduce an implementation's size, the implementation's performance may suffer. Some observers have compared this phenomenon to a wheel with numerous pins, as illustrated in Figure 1.3. If you push one pin in, such as size, then the other

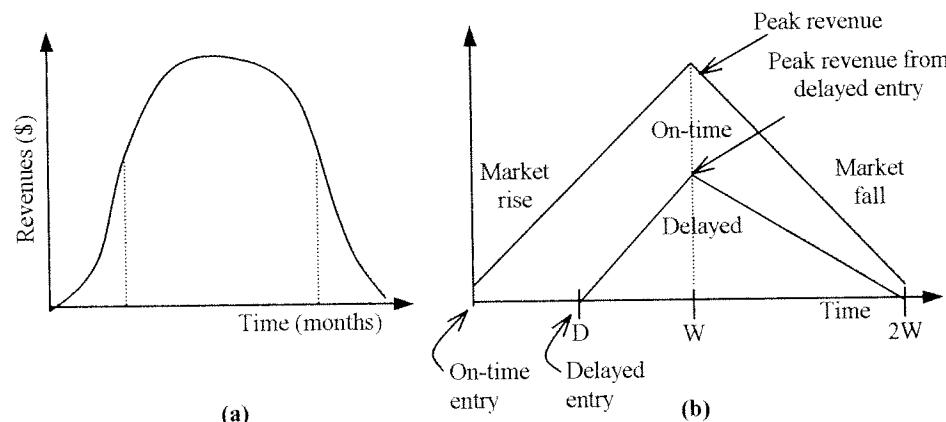


Figure 1.4: Time-to-market: (a) market window, (b) simplified revenue model for computing revenue loss from delayed entry.

pins pop out. To best meet this optimization challenge, the designer must be comfortable with a variety of hardware and software implementation technologies, and must be able to migrate from one technology to another, in order to find the best implementation for a given application and constraints. Thus, a designer cannot simply be a hardware expert or a software expert, as is commonly the case today; the designer must have expertise in both areas.

The Time-to-Market Design Metric

Most of these metrics are heavily constrained in an embedded system. The time-to-market constraint has become especially demanding in recent years. Introducing an embedded system to the marketplace early can make a big difference in the system's profitability, since market windows for products are becoming quite short, with such windows often measured in months. For example, Figure 1.4(a) shows a sample market window during which time a product would have highest sales. Missing this window, which means that the product begins being sold further to the right on the time scale, can mean significant loss in sales. In some cases, each day that a product is delayed from introduction to the market can translate to a one-million-dollar loss. The average time-to-market constraint has been reported as having shrunk to only 8 months!

Adding to the difficulty of meeting the time-to-market constraint is the fact that embedded system complexities are growing due to increasing IC capacities, as we will see later in this chapter. Such rapid growth in IC capacity translates into pressure on designers to add more functionality to a system. Thus, designers today are being asked to do more in less time.

Let's investigate the loss of revenue that can occur due to delayed entry of a product in the market. We'll use a simplified model of revenue that is shown in Figure 1.4(b). This model assumes the peak of the market occurs at the halfway point, denoted as W , of the product life, and that the peak is the same even for a delayed entry. The revenue for an

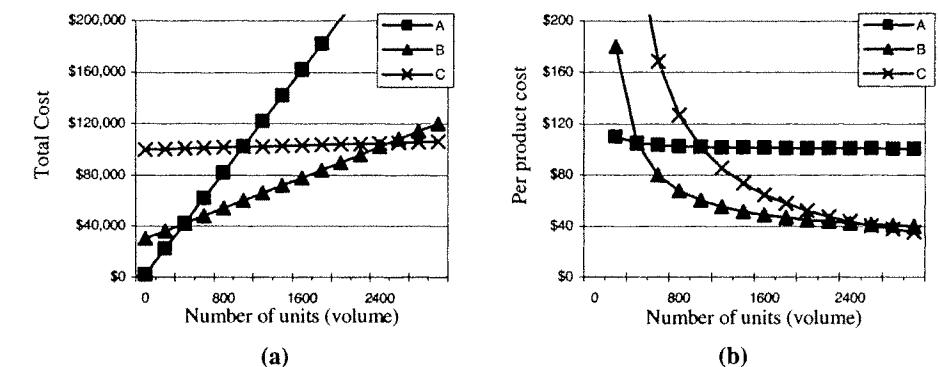


Figure 1.5: Costs for technologies A, B, and C as a function of volume: (a) total cost, (b) per-product cost.

on-time market entry is the area of the triangle labeled *On-time*, and the revenue for a delayed entry product is the area of the triangle labeled *Delayed*. The revenue loss for a delayed entry is just the difference of these two triangles' areas. Let's derive an equation for percentage revenue loss, which equals $((\text{On-time} - \text{Delayed}) / \text{On-time}) * 100\%$. For simplicity, we'll assume the market rise angle is 45 degrees, meaning the height of the triangle is W , and we leave as an exercise the derivation of the same equation for any angle. The area of the *On-time* triangle, computed as $\frac{1}{2} * \text{base} * \text{height}$, is thus $\frac{1}{2} * 2W * W$, or W^2 . The area of the *Delayed* triangle is $\frac{1}{2}(W - D + W) * (W - D)$. After algebraic simplification, we obtain the following equation for percentage revenue loss:

$$\text{percentage revenue loss} = (D(3W - D) / 2W^2) * 100\%$$

Consider a product whose lifetime is 52 weeks, so $W = 26$. According to the preceding equation, a delay of just $D = 4$ weeks results in a revenue loss of 22%, and a delay of $D = 10$ weeks results in a loss of 50%. Some studies claim that reaching market late has a larger negative effect on revenues than development cost overruns or even a product price that is too high.

The NRE and Unit Cost Design Metrics

As another exercise, let's consider NRE cost and unit cost in more detail. Suppose three technologies are available for use in a particular product. Assume that implementing the product using technology A would result in an NRE cost of \$2,000 and unit cost of \$100, that technology B would have an NRE cost of \$30,000 and unit cost of \$30, and that technology C would have an NRE cost of \$100,000 and unit cost of \$2. Ignoring all other design metrics, like time-to-market, the best technology choice will depend on the number of units we plan to produce. We illustrate this concept with the plot of Figure 1.5(a). For each of the three technologies, we plot total cost versus the number of units produced, where:

$$\text{total cost} = \text{NRE cost} + \text{unit cost} * \# \text{ of units}$$

We see from the plot that, of the three technologies, technology A yields the lowest total cost for low volumes, namely for volumes between 1 and 400. Technology B yields the lowest total cost for volumes between 400 and 2500. Technology C yields the lowest cost for volumes above 2500.

Figure 1.5(b) illustrates how larger volumes allow us to amortize NRE costs such that lower per-product costs result. The figure plots per-product cost versus volume, where:

$$\text{per-product cost} = \text{total cost} / \# \text{ of units} = \text{NRE cost} / \# \text{ of units} + \text{unit cost}$$

For example, for technology C and a volume of 200,000, the contribution to the per-product cost due to NRE cost is $\$100,000 / 200,000$, or \$0.50. So the per-product cost would be $\$0.50 + \$2 = \$2.50$. The larger the volume, the lower the per-product cost, since the NRE cost can be distributed over more products. The per-product cost for each technology approaches that technology's unit cost for very large volumes. So for very large volumes, numbering in the hundreds of thousands, we can approach a per-product cost of just \$2 — quite a bit less than the per-product cost of over \$100 for small volumes.

Clearly, one must consider the revenue impact of both time-to-market and per-product cost, as well as all the other relevant design metrics when evaluating different technologies.

The Performance Design Metric

Performance of a system is a measure of how long the system takes to execute our desired tasks. Performance is perhaps the most widely used design metric in marketing an embedded system, and also one of the most abused. Many metrics are commonly used in reporting system performance, such as clock frequency or instructions per second. However, what we really care about is how long the system takes to execute our application. For example, in terms of performance, we care about how long a digital camera takes to process an image. The camera's clock frequency or instructions per second are not the key issues — one camera may actually process images faster but have a lower clock frequency than another camera.

With that said, there are several measures of performance. For simplicity, suppose we have a single task that will be repeated over and over, such as processing an image in a digital camera. The two main measures of performance are:

- *Latency*, or *response time*: The time between the start of the task's execution and the end. For example, processing an image may take 0.25 second.
- *Throughput*: The number of tasks that can be processed per unit time. For example, a camera may be able to process 4 images per second.

However, note that throughput is not always just the number of tasks times latency. A system may be able to do better than this by using parallelism, either by starting one task before finishing the next one or by processing each task concurrently. A digital camera, for example, might be able to capture and compress the next image, while still storing the previous image to memory. Thus, our camera may have a latency of 0.25 second but a throughput of 8 images per second.

In embedded systems, performance at a very detailed level is also often of concern. In particular, two signal changes may have to be generated or measured within some number of nanoseconds.

Speedup is a common method of comparing the performance of two systems. The speedup of system A over system B is determined simply as:

$$\text{speedup of A over B} = \text{performance of A} / \text{performance of B}.$$

Performance could be measured either as latency or as throughput, depending on what is of interest. Suppose the speedup of camera A over camera B is 2. Then we also can say that A is 2 times faster than B and B is 2 times slower than A.

1.3 Processor Technology

We can define *technology* as a manner of accomplishing a task, especially using technical processes, methods, or knowledge. This book takes the perspective that three types of technologies are central to embedded system design: processor technologies, IC technologies, and design technologies. We describe all three briefly in this chapter and provide further details in subsequent chapters.

Processor technology relates to the architecture of the computation engine used to implement a system's desired functionality. Although the term *processor* is usually associated with programmable software processors, we can think of many other, nonprogrammable, digital systems as being processors also. Each such processor differs in its specialization towards a particular function (e.g., image compression), thus manifesting design metrics different than other processors. We illustrate this concept graphically in Figure 1.6. The application requires a specific embedded functionality, symbolized as a cross, such as the summing of the items in an array, as shown in Figure 1.6(a). Several types of processors can implement this functionality, each of which we now describe. We often use a collection of such processors to optimize a system's design metrics, as in our digital camera example.

General-Purpose Processors — Software

The designer of a *general-purpose processor*, or *microprocessor*, builds a programmable device that is suitable for a variety of applications to maximize the number of devices sold. One feature of such a processor is a program memory — the designer of such a processor does not know what program will run on the processor, so the program cannot be built into the digital circuit. Another feature is a general datapath — the datapath must be general enough to handle a variety of computations, so such a datapath typically has a large register file and one or more general-purpose arithmetic-logic units (ALUs). An embedded system designer, however, need not be concerned about the design of a general-purpose processor. An embedded system designer simply uses a general-purpose processor, by programming the processor's memory to carry out the required functionality. Many people refer to this part of an implementation as the "software" portion.

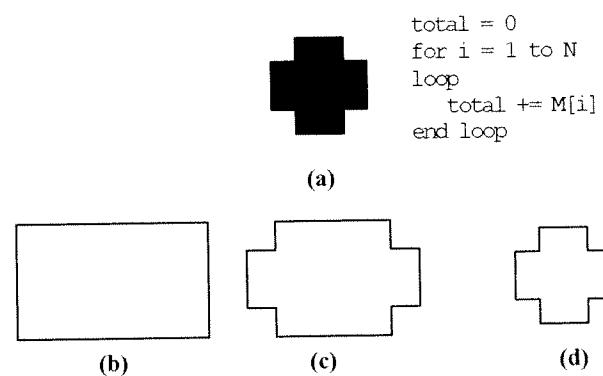


Figure 1.6: Processors vary in their customization for the problem at hand: (a) desired functionality, (b) general-purpose processor, (c) application-specific processor, (d) single-purpose processor.

Using a general-purpose processor in an embedded system may result in several design metric benefits. Time-to-market and NRE costs are low because the designer must only write a program but not do any digital design. Flexibility is high because changing functionality requires changing only the program. Unit cost may be low in small quantities compared with designing our own processor, since the general-purpose processor manufacturer sells large quantities to other customers and hence distributes the NRE cost over many units. Performance may be fast for computation-intensive applications, if using a fast processor, due to advanced architecture features and leading-edge IC technology.

However, there are also some design-metric drawbacks. Unit cost may be relatively high for large quantities, since in large quantities we could design our own processor and amortize our NRE costs such that our unit cost is lower. Performance may be slow for certain applications. Size and power may be large due to unnecessary processor hardware.

For example, we can use a general-purpose processor to carry out our array-summing functionality from the earlier example. Figure 1.6(b) illustrates that a general-purpose processor covers the desired functionality but not necessarily efficiently. Figure 1.7(a) shows a simple architecture of a general-purpose processor implementing the array-summing functionality. The functionality is stored in a program memory. The controller fetches the current instruction, as indicated by the program counter (PC), into the instruction register (IR). It then configures the datapath for this instruction and executes the instruction. It then determines the next instruction address, sets the PC to this address, and fetches again.

Single-Purpose Processors — Hardware

A *single-purpose processor* is a digital circuit designed to execute exactly one program. For example, consider the digital camera example of Figure 1.2. All of the components other than the microcontroller are single-purpose processors. The JPEG codec, for example, executes a single program that compresses and decompresses video frames. An embedded system

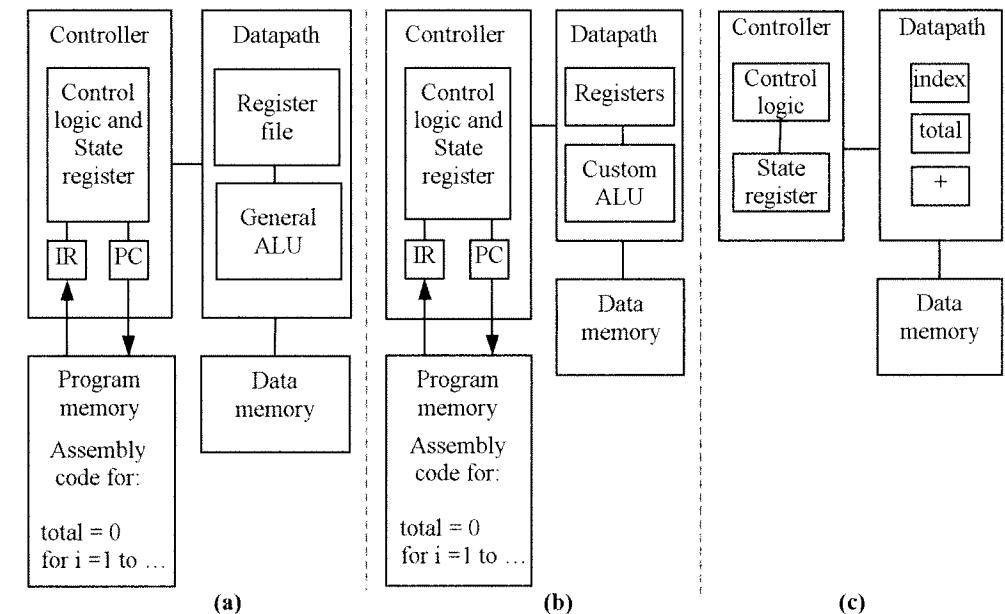


Figure 1.7: Implementing desired functionality on different processor types: (a) general-purpose, (b) application-specific, (c) single-purpose.

designer may create a single-purpose processor by designing a custom digital circuit, as discussed in later chapters. Alternatively, the designer may purchase a predesigned single-purpose processor. Many people refer to this part of the implementation simply as the “hardware” portion, although even software requires a hardware processor on which to run. Other common terms include *coprocessor*, *accelerator*, and *peripheral*.

Using a single-purpose processor in an embedded system results in several design-metric benefits and drawbacks, which are essentially the inverse of those for general-purpose processors. Performance may be fast, size and power may be small, and unit cost may be low for large quantities, while design time and NRE costs may be high, flexibility low, unit cost high for small quantities, and performance may not match general-purpose processors for some applications.

For example, Figure 1.6(d) illustrates the use of a single-purpose processor in our embedded system example, representing an exact fit of the desired functionality, nothing more, nothing less. Figure 1.7(c) illustrates the architecture of such a single-purpose processor for the example. The datapath contains only the essential components for this program: two registers and one adder. Since the processor only executes this one program, we hardwire the program's instructions directly into the control logic and use a state register to step through those instructions, so no program memory is necessary.

Poul Staal Vinje

Projektledelse af systemudvikling

Nyt Teknisk Forlag, 3. udgave

Kapitel 5.8 - 5.9, s. 100 – 103 (Projektmodel, udviklingsmetodik)

Kapitel 6.3, s. 119 – 131 (Udviklingsstrategier)

Kapitel 10.1 – 10.7, s. 259 – 277 (Styringsprocessen - RUP)

- strategier
- personaleressourcer
- økonomiske rammer.

5.8 Projektmodel

Formål

En projektmodel opfylder forskellige behov for projektlederen og -deltagere. Derfor optræder den både i dette kapitel om projektlederens værktøjskasse og senere i kataloget over produktivitetsfremmende foranstaltninger. Det samme gælder udviklingsmetodik i dette kapitel.

Formålet er at planlægge projektforløbet på grundlag af en generel model. Definitionen på en projektmodel er:

En projektmodel er en teoretisk og forenklet model af et projektforløb.

Den er teoretisk fordi modellen aldrig vil være identisk med et forløb i praksis. Der vil altid være en række afvigelser i et konkret projekt i forhold til standardforløbet. En model er desuden forenklet, fordi den kun indeholder aktiviteter, der forekommer i flertallet af projekter, og kun beskriver dem kortfattet og generelt.

På trods af teori og forenklinger er en projektmodel et nyttigt værktøj. Den giver væsentlige fordele inden for:

- udvikling
- kommunikation
- dokumentation.

Udviklingsmæssigt bruges modellen som grundlag for aktivitetsplanlægning og fremgangsmåde i projektet. Projektplanen kan i vid udstrækning bygges over de forud definerede aktiviteter. De overordnede faser, de fleste hovedaktiviteter og en del af detailaktiviteterne fås direkte fra modellen.

Projektet bruger således ingen tid på at definere rammerne, et minimum af tid på det gængse og sædvanlige og den meste tid på det usædvanlige og projektspecifikke.

Kommunikationsmæssigt spares der megen tid. Når en interessent modtager et oplæg fra en projektgruppe, er det praktisk at vide, hvad opslægget kan forventes at indeholde, dvs. hvorvidt specifikation af kravene til

svartider og tilgængelighed fastlægges endeligt i dette oplæg eller i et kommende oplæg.

Modellen anvendes til løbende at dokumentere i et standardiseret udseende.

Kreativitet og standardisering

En model går ikke ud over kreativiteten. En model fjerner planløshed og øger sikkerheden for at alle aktiviteter bliver husket. Erfarne projektledere følger ofte egne retningslinier, der kan være både solide og effektive. Der synes måske derfor ikke at være brug for en model. Problemet opstår når medarbejderne forlader organisationen.

Modeltyper

Der er et antal modeller til rådighed som støtter IT-projekter. Der findes gennemprøvede modeller som er rettet mod udvikling af administrative systemer, der findes modeller som er specielt rettet mod anvendelse af bestemte udviklingsmetodikker, og der er modeller, der støtter integrationsprojekter, hvor udvikling er en mindre del af arbejdet.

Til et almindeligt målstyret projekt finder der mange „færdige“ vandfaldsmodeller, bygget op med faserne:

- ▶ Foranalyse
- ▶ Analyse
- ▶ Design
- ▶ Konstruktion
- ▶ Test.

Hver fase er detaljeret i 10-20 hovedaktiviteter, og under dem flere hundrede detailaktiviteter i hver fase.

Hvis der er brug for en cyklisk (iterativ) model, kan man for eksempel se på DSDM (Dynamic Systems Development Method). Den er skræddersyet til et iterativt forløb med brug af prototyper. Den indeholder følgende faser:

- ▶ Business Study
- ▶ Functional Model Iteration
- ▶ Design and Build Iteration
- ▶ Implementation.

Modellens faser gennemføres flere gange, med stigende detaljeringsgrad.

Gennemførelse

I forbindelse med opstarten vurderes aktiviteterne i modellen:

- ▶ Hvilke aktiviteter er overflødige i dette projekt?
- ▶ Hvilke aktiviteter skal tilføjes i dette projekt?

Den reviderede model bruges derefter som:

- ▶ Skelet for aktivitetsplanlægningen
- ▶ Grundlag for tidsestimeringen
- ▶ Grundlag for kalenderfastsættelsen
- ▶ Skelet for bemanding.

En gennemarbejdet model indeholder også forslag til reviews og milepæle.

Resultat og anvendelse

Modellen giver i sig selv en grundlæggende systematik. Desuden fungerer den som skelet for mange andre aktiviteter, der er knyttet til bestemte tidspunkter eller opgaver i forløbet. Det kan være opfølgning på økonomi, kvalitetsstyring, godkendelser i styregruppen, udfærdigelse af uddannelsesmateriale eller fremstilling af brugervejledninger.

Modellen kan udbygges med formelle udviklingsmetodikker. Hvis projektet skal bruge E/R- eller OO-analyse, kan de nødvendige aktiviteter placeres i modellen. Resultatdokumentationen integreres med den øvrige systemdokumentation og bliver en del af den samlede dokumentation efter afslutningen af en fase.

5.9 Udviklingsmetodik

Formål

Dette afsnit er kun relevant for udviklingsprojekter og for eksempel ikke for integrations- og implementeringsprojekter.

Udviklingsmetodikker har til formål at understøtte en systematisk analyse- og designproces. Fejl i disse faser er de dyreste at rette og de mest afgørende for interesserernes tilfredshed.

Teknikkerne kan være enkeltstående eller integrerede, det sidste for eksempel som Rational Unified Process med værktøjsstøtte.

Valg af metodik

En god udviklingsmetodik giver produkter, der kan læses og forstås af interesserterne.

Desuden skal en metodik indeholde værktøjer til flere formål. Der bør være værktøjer til den kreative udredende fase, andre værktøjer, der dokumenterer udredningen i form af grafer og modeller, og værktøjer og teknikker til verifikation og test af resultatet. Det er en fordel hvis metodikken er automatiseret.

Grundlæggende skal der vælges mellem at bruge en totalmetode eller en kombination af flere forskellige. I sidste tilfælde kan valg af metodikker gøres mere situationsbestemt.

Totalmetoderne købes typisk sammen med en udviklingsmodel, og metode og model anvendes integreret.

For de fleste organisationer vil en løsning med flere mere eller mindre uafhængige metodikker være et naturligt valg. En totalmetode kræver ofte en tilsvarende totalomlægning af udviklingsafdelingens aktiviteter. Uddannelse af hele IT-afdelingens personale er omfattende og forudsætter hele afdelingens medvirken og motivation.

Projektlederen bør selv vælge nogle metodikker at arbejde med i projektet, hvis organisationen ikke har valgt nogen som standard.

Gennemførelse

Anvendelsen af en metodik kan ske i et samarbejde mellem interesserterne og projektgruppen. Det forudsætter at alle har mulighed for at afsætte tiden. Direkte medvirken forudsætter også en uddannelse i den valgte metodik.

Værktøjsstøtte

Fordelen ved automatisering er markant. Det er dog vigtigt at metodikken er formel og anerkendt i litteraturen. Automatisering er ikke garanti for systematisk indførelse, men når en metodik er valgt og uddannelsen er gennemført, kan værktøjer understøtte den praktiske brug. Metodikker bygger ofte på modeller og grafiske fremstillinger, der uden automativering kræver en del tid at vedligeholde manuelt.

Resultat og anvendelse

Metoderne indeholder normalt en fremgangsmåde og en eller flere diagrammeringsteknikker. For eksempel til at dokumentere data-, proces- og

view af kontrakten. Det holdes i leverandørorganisationen før kontrakten underskrives. De dele af organisationen, der skal involveres i projektet, tilkendegiver at de står inde for deres del. Det forudsætter at væsentlige forhold omkring projektets indhold, herunder målene, er klare.

Workshops

Hvis projektet er til RAD (Rapid Application Development), er der basis for en workshop. Det er i dette tilfælde en „Scoping Session“, hvor systemet afgrænses og beskrives ved hjælp af „Requirement Planning“. Det kræver at de egentlige beslutningstagere (og ikke nogle repræsentanter for dem) arbejder sammen i 1-2 uger og aftaler og dokumenterer kravene. På den måde brændes der 5-10 arbejdsuger af på kort tid, men det er jo det, der er målet i et RAD-forløb.

Når forudsætningerne for RAD er opfyldt, og det drejer sig i første række om at have kompetente personer til stede, byder RAD på mange fordele for projektlederen. Beslutninger om mål og krav kan træffes meget hurtigt og vil senere blive bakket op.

Function Points

Optælling af Function Points er et godt middel til at beskrive systemets omfang. Man kan begynde på det i en tidlig udgave under målformuleringen. Det tvinger i sig selv bedre beskrivelser frem. En Function Point-beskrivelse giver desuden et suverænt godt grundlag for at vurdere ændringsønsker senere i projektet.

Ændringer til målene

I løbet af et projekt opstår der ofte behov for ændringer. Det kan skyldes fejl og mangler i den første formulering af målene, eller at der er sket ændringer i virkelighedens verden, eller simpelt hen at køberen har fået en god idé.

Senere i dette kapitel er der beskrevet en procedure for behandling af ændringer, men allerede under målformuleringen skal man forestille sig, hvordan ændringerne skal håndteres. Studér målformuleringen som den foreligger, og vurdér om den er solid nok. Vurdér om projektet modsat står med et mangelfuld grundlag, der ikke er tilstrækkeligt diskuteret med interessenterne. Vurdér om det senere vil være let eller svært at afgøre om en ændring er et nyt ønske, der skal betales for.

Problemer med ændringer stammer ofte fra, at de ikke behandles for-

retningsmæssigt. Det må ikke gerne blive et problem i sig selv, at de forekommer, for det sker praktisk taget altid.

I nogle projekter er der en betydelig tidsmæssig afstand imellem målformulering og starten på projektet. Så er det fornuftigt at gennemgå målene igen og få en ny godkendelse.

6.3 Strategier

Indledning

Strategierne dokumenterer projektets overordnede styring af forløbet. Valg af strategi baseres på forandrings-, system- og projektmålene.

De godkendte strategier påvirker projektets organisation, fremgangsmåden i projektet, sammensætningen af projektdeltagere og valget af værkstøjer.

Projektledere med IT baggrund mangler ofte erfaring i strategisk planlægning og er ikke altid bevidste om behovet. Projektledere med en relevant videregående uddannelse er bevidste om behovet, men mangler tilstrækkelig indsigt i naturen af systemudviklingsprojekter til at vælge de rette strategier.

Derfor er strategisk planlægning et eksempel på et uddannelsesbehov, der skal tilgodeses uanset projektlederens baggrund. Det er desuden et eksempel på, at det er en fordel at rekruttere projektledere med forskellig udannelsesmæssig baggrund.

Styr på projektet

Projektlederen skal styre forløbet og ikke omvendt. Strategier er et middel til at sikre kontrollen.

Strategierne fastlægger fremgangsmåden i projektet ved at tage udgangspunkt i målene og øge sandsynligheden for at målene realiseres.

Strategier giver projektarbejdsformen spændstighed og dynamik. De giver aktiviteterne mening og definerer aktiviteternes indbyrdes sammenhæng.

Strategierne udarbejdes af projektlederen og godkendes af styregruppen/liniechefen, projektdeltagerne og nøgleinteressenterne. Strategierne sørger for at projektets forløb fastlægges på et tidligt tidspunkt. En yderligere fordel er, at der etableres en fælles opfattelse, og at der skabes nogle fælles forventninger til styringen af forløbet. Strategierne har også en po-

sitiv effekt i selve forløbet, idet behovet for løbende at få truffet beslutninger mindskes. Strategierne fungerer som overordnede retningslinier der definerer de styringsmæssige rammer for projektlederen's handlerum.

Et godt eksempel på strategiens betydning for at realisere et mål er teststrategiens betydning for høje krav til testen. Et mål om „o-fej“ skal bakes op af en strategi, der beskriver hvordan, hvem og hvornår, der gøres noget i projektet for at nå målet.

Indholdets omfang

En strategi er også et dokument og en del af projektets formelle dokumentation. Dokumentet beskriver mål og midler, det sidstnævnte i form af de taktiske tiltag der sikrer at målet nås.

Den generelle model for virksomhedsplanlægning arbejder med tre niveauer: det strategiske, det taktiske og det operationelle niveau. De samme tre niveauer bruges til at planlægge projektet. Det er kun naturligt, da et projekt kan betragtes som en minivirksomhed, blot med begrænset levetid.

Det er praktisk at beskrive et mål og de dertil hørende taktiske tiltag i samme fysiske dokument. Strategien dokumenterer således resultatet af de to øverste planlægningsniveauer. Projektplanen dokumenterer det operationelle niveau i form af de bemandede aktiviteter. Projektets fysiske planlægning består således af et antal strategier og af en projektplan med de operationelle aktiviteter.

Taktisk planlægning

Strategiske mål realiseres ved hjælp af taktiske tiltag inden for fire områder:

Opgaver: Hvilke aktiviteter skal udføres for at nå målet?

Aktører: Hvilke typer af medarbejdere skal udføre aktiviteterne?

Organisation: Hvilke indbyrdes relationer skal der etableres mellem aktørerne?

Teknologi: Hvilke metoder, teknikker og værktøjer er nødvendige?

Til de fire spørgsmål er der sædvanligvis flere mulige svar, og i valget af den taktiske løsning får planen det specifikke præg, der passer til projektet.

De fire områder kan gennemgås og afdækkes i fri rækkefølge. I praksis er der behov for at analysere dem i flere omgange. Analysen af et element skaber behov for at behandle de øvrige. Bearbejdningen af *opgaver* kan

bringe en ellers overset *aktør* frem i lyset, og valget af *teknologi* afføder ofte nye opgaver.

De taktiske løsninger skal hænge sammen. Det skal være muligt at løse opgaverne ved hjælp af de til rådighed stående ressourcer, og de valgte ressourcer skal kunne håndtere de til rådighed stående værktøjer. Grundlaget for planlægning er ikke altid ideelt, og det kan blive nødvendigt at justere på de ønskværdige strategiske mål, efterhånden som den taktiske planlægning gennemføres. Men den derved opnåede sammenhæng mellem mål og taktik og den opnåede sammenhæng mellem de taktiske elementer indbyrdes er med til at skabe den gode operationelle planlægning. Det bidrager til planens realisme.

Beskrivelse af en strategi fylder sjældent mere end et par sider tekst. Den skal være væsentlig, konkret og kortfattet.

Strategiske områder

De strategiske mål kan grupperes og samles fysisk i en eller flere strategier. En strategi vil dække et væsentligt område af projektet eller produktet. I et systemudviklingsprojekt vil der normalt være behov for tre strategier:

- ▶ En udviklingsstrategi, der sætter mål for og beskriver udviklingsforløbet
- ▶ En teststrategi, der sætter mål for og beskriver afprøvningsforløbet
- ▶ En strategi for indføring, der sætter mål for og beskriver implementeringen.

Udviklingsstrategier

Følgende grundtyper af udviklingsstrategier gennemgås i det følgende:

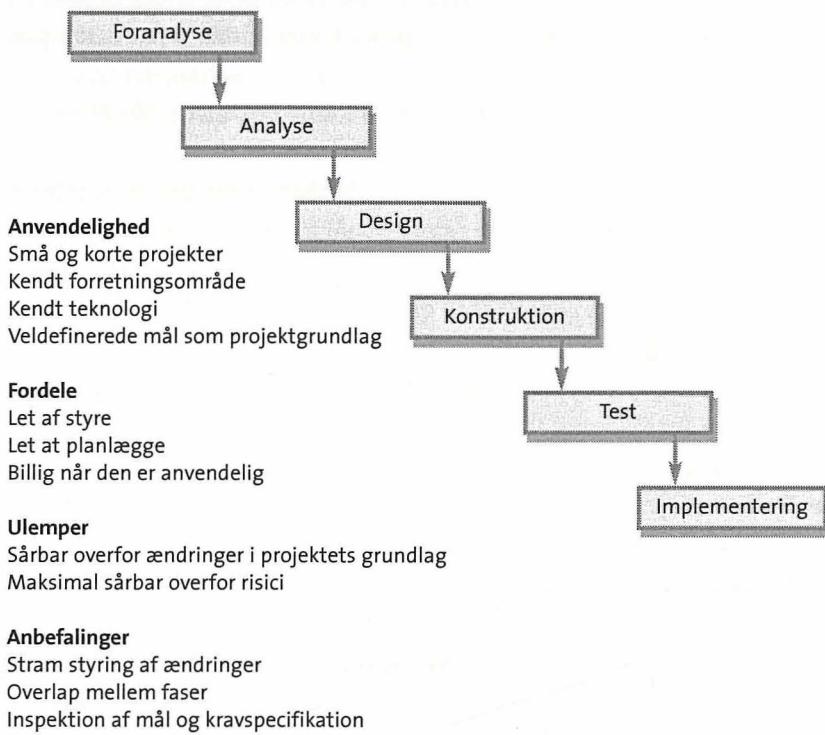
- ▶ Vandfald
- ▶ Delleveringer
- ▶ Eksperimentel eller cyklisk udvikling
- ▶ Versionsvis udvikling
- ▶ „Rapid Application Development“ (RAD)
- ▶ „Rapid Application Prototyping“ (RAP)
- ▶ Genbrug.

Den sidste strategi kan synes ude af niveau med de andre. Umiddelbart er genbrug et taktisk middel til at øge produktiviteten. Men i forbindelse med

at øge mængden af det mulige genbrug er det fornuftigt at beskrive det som et strategisk mål.

Vandfald

Denne velkendte strategi bygger på at gøre en hel fase af arbejdet færdigt før den næste startes. Foranalysen gøres færdig, derefter starter projektet forfra, men denne gang med analyse, og så videre med de andre faser. Det er altså hele systemet, der arbejdes igennem i hver fase, og vandfaldet symboliserer, at der kun går i én retning. Man kan ikke gå imod strømmen, i hvert fald ikke uden stort besvær og tidsspilde. **Strategien benyttes, når opgaven er veldefineret og velkendt.** Forløbet skal have en kort varighed, dvs. mindre end 3 måneder, inden for et kendt og veldefineret forretningsområde og under velkendte forhold med hensyn til udviklings- og testmiljø, udviklingsmetodik, platforme etc. Hvis disse betingelser ikke er opfyldt, er denne strategi ikke den rette.



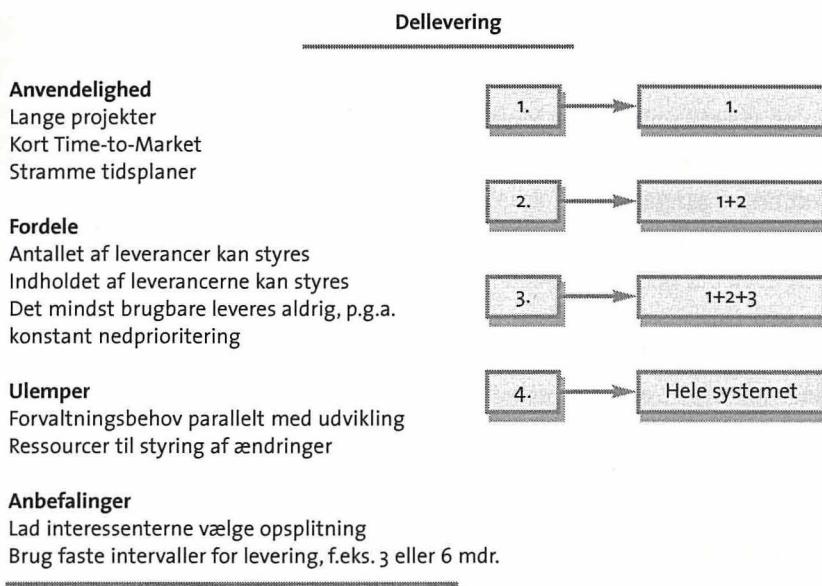
Figur 6.3: Vandfald med overlappende faser.

Strategien er let at planlægge og udføre. Figur 6.3 er praktisk taget en grovplan. **Ulemper** er primært dens sårbarhed over for ændringer i kravene. Der er brug for en stram og formel styring af ændringer, således at tidligere fasers produkter kan revideres i takt med godkendte ændringer.

Man kan overveje overlap mellem faserne, således som det fremgår af figur 6.3. Det giver mulighed for at åbne næste fases arbejde før den nuværende fase afsluttes. Formålet er at se, om der er overraskelser, der peger på at projektet skal blive lidt længere i den aktuelle fase, før denne lukkes endeligt.

Delleveringer

Man kan have brug for at udvikle og levere systemet i flere fysiske delleveringer. Strategien bruges når nogle dele har større betydning end andre. Det er brugernes eller leverandørorganisationens behov, der styrer opsplitning og udvælgelse. Forløbet drives altså frem af forretnings- eller markedsmæssige behov. Hvis der er en stram plan, fordi „Time-to-Market“ er væsentlig, kan de dele af systemet, der er vigtigst for kunder/markedet, leveres lang tid før det ville have været muligt at levere det samlede system. Omvendt kommer de mindre afgørende dele senere, end det ellers ville have været muligt.



Figur 6.4: Delleveringer.

Den største ulempe er, at der opstår et vedligeholdelsesbehov efter første dellevering. Den er i sagens natur den vigtigste, og der skal etableres forvaltnings- og supportgrupper.

Figur 6.4 viser at de fire delleveringer bygges op til det samlede system. Det er i øvrigt normalt at den sidste levering aldrig realiseres. Det er dele der også fra start kunne have været skrottet, men som først i en egentlig prioritering bliver sat nederst på listen. Der er til stadighed noget med høje-re prioritet.

Projektlederen skal ikke bestemme antallet eller indholdet af delleveringerne. Overlad det til køberen eller salgsafdelingen. Definér til gengæld en formel og aftalt leveringsplan, for eksempel med leverancer hver 3. eller 6. måned.

Dellevering = vandfald

Den enkelte dellevering køres som et isoleret vandfald, men således at den første skaber den overordnede arkitektur og bereder vejen for de efter-følgende.

Eksperimentel strategi/cyklist udvikling

Vi forlader nu den målstyrede systemudvikling til fordel for det eksperimentelle. I en ægte eksperimentel strategi arbejdes der med et ukendt antal versioner. Planlægning er vanskelig, fordi der mangler naturlige millepæle at hægte kvalitetsstyring og godkendelser op på.

Det kan være nødvendigt at eksperimentere når der arbejdes med nye forretningsområder eller ny teknologi. Strategien kan få mange til at bække et par meter fordi den forbindes med risici og usikkerhed. Men den er glimrende på områder hvor den optimale løsning ikke kan besluttes, men kun kan findes ved at lære mere. Figur 6.5 viser at der kan gås rundt i forløbet et ukendt antal gange.

Strategien er langt hen ad vejen „naturlig“ for udviklere og slutbrugere. Men de er heller ikke begrænset af budgetter eller leveringsdatoer. Og håndtering af ændringer er en smal sag, hvor det i andre strategier er en risiko.

Ulempene er den manglende planlægning og de deraf følgende dårlige muligheder for opfølgning på fremdrift. Det er også svært at afgøre hvornår processen skal i et tilbageløb til en tidligere fase eller drives fremad i detaljering.

Forudsætningerne for succes ligger dels i at informere om strategiens natur og konsekvenser, dels i at styre forløbet gennem lukketider – for eksempel ved at bekendtgøre at om to uger fortsættes der til en anden del af systemet, eller at der stiles mod en endelig godkendelse af datamodellen.

Anvendelighed

Søge-lære processer
Nyt forretningsområde
Ny teknologi
Ved uklare krav eller mål

Fordele

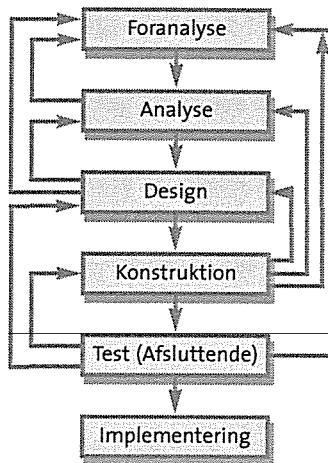
„Naturlig“ for udviklere og brugere
Ændringsønsker kan absorberes

Ulemper

Ingen planlægning
Ingen opfølgning
Testen skal gentages, og i sit fulde omfang til sidst

Anbefalinger

Brug et prototypewærktøj
Brugerne skal være aktive deltagere i projektet
Annoncering af lukketider
Strategien skal være ønsket af interessenterne



Figur 6.5: Eksperimentel/Cyklistisk.

Kompetente brugere, med tid og evner til at deltage konkret i projektet, styrker forløbet.

Versionsvis udvikling

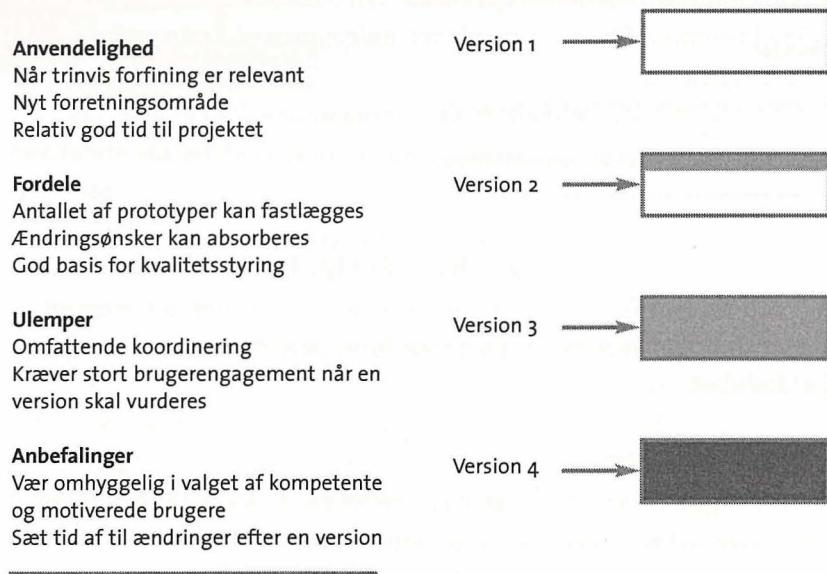
Denne strategi kan bruges til at give interessenterne indflydelse undervejs, men uden at det bliver egentlig eksperimentelt. Der udvikles prototyper/versioner i en søge-lære-proces, men det aftales fra start hvor mange versioner der udvikles. Hver version omfatter i principippet hele det nye system i stadig stigende detaljeringsgrad. Versionerne kan ikke sættes i drift, de kan kun bruges til at afgøre det videre forløb. Figur 6.6 viser et forløb med fire versioner.

Fordelen i forhold til en ægte eksperimentel strategi er aftalen om det specifikke antal versioner. Den første kan være en ren prototype på brugergrænsefladen, den næste en version der tydeliggør, hvilke data der bliver tilgængelige, og derefter et par stykker mere der viser den færdige funktionalitet. Versionerne planlægges med kendte milepæle, og brugere og andre interessenter forbereder sig på formelle godkendelsesreview.

Det bringer ulemperne på bane. Der er brug for koordinering af mange menneskers indsats, og der er behov for at planlægge hvordan en version gøres tilgængelig for brugerne. Den slags er altid nemmere i teorien end i

praksis. Det kræver meget af projektlederen at sørge for interessenternes medvirken. Det må ikke blive ligegyldige seancer, hvor interessenterne siger, at det ser da meget pænt ud. Den risiko er altid til stede ved foreløbige udgaver.

Projektlederen skal sikre sig to ting: at der reelt er interessenter der kan – og vil – sige deres mening. Dernæst at der er vilje og evne i projektgruppen til at tage sig af det, der bliver sagt. Det sidste er vigtigt i den forstand, at forløbet er tidsspilde, hvis der ikke seriøst kan modtages og gennemføres ændringer.



Figur 6.6: Versionsvis udvikling.

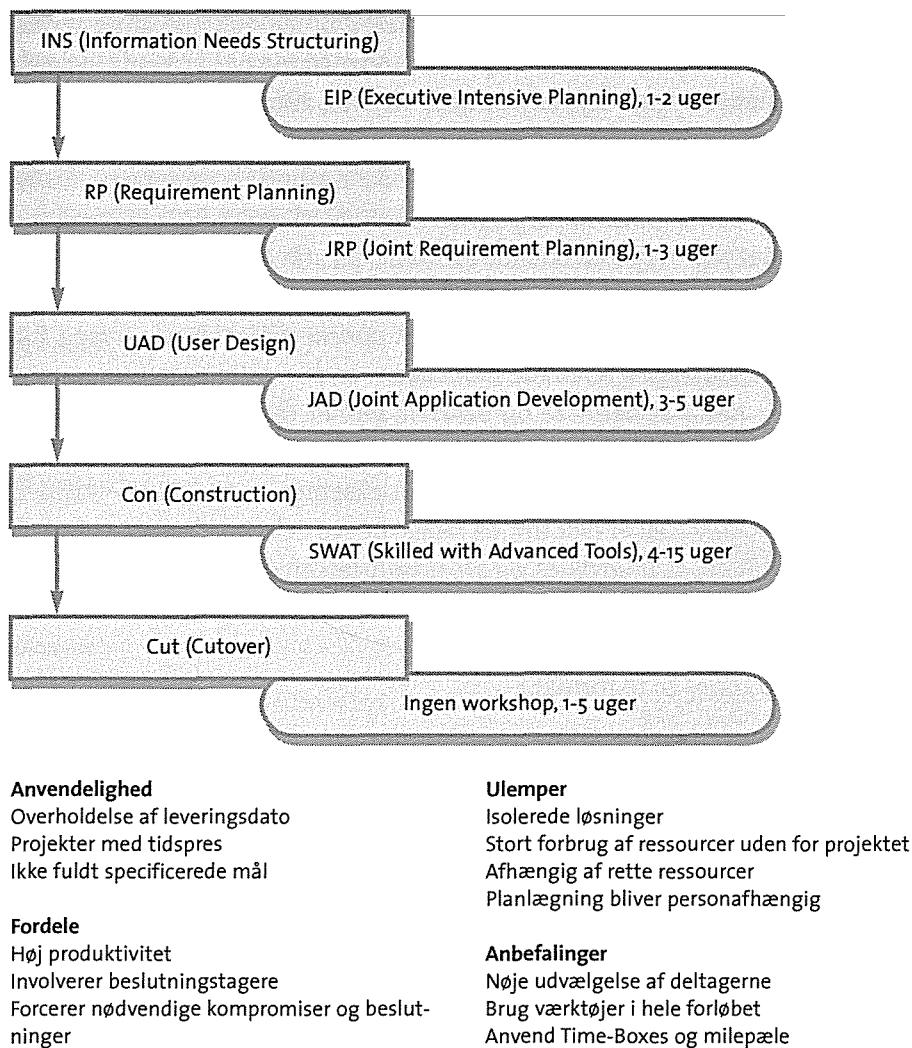
Rapid Application Development (RAD)

Når det handler om kort leveringstid og overholdelse af datoer, er RAD det rigtige valg.

RAD bygger på at samle de rigtige personer på de rigtige tidspunkter og i det nødvendige tidsrum. Der anvendes workshops af 1-3 ugers varighed, hvor deltagerne arbejder sammen på opgaven.

De første workshops i et RAD-forløb involverer beslutningstagerne. Der ved spares der tid ved at fjerne den del af processen, hvor de egentlige beslutningstagere skal vurdere en projektgruppens arbejde. I stedet sættes de til at beskrive det, de gerne vil have. Den første workshop beslutter, hvilke data/informationer systemet skal håndtere, for eksempel i form af „Busi-

ness Objects". Derefter kan det være andre beslutningstagere, der på næste workshop fastlægger kravene til funktionalitet, fulgt op med 3-5 ugers samarbejde mellem slutbrugere og udviklere til at designe systemets udseende. Til sidst er der brug for 4-15 ugers konstruktionsarbejde. Figur 6.7 viser et forløb på ca. et halvt år. Kasserne viser faserne i forløbet, og ovalerne er workshops. Det karakteristiske ved strategien er den høje ind- og udskiftning af personer, idet der vælges i forhold til behov og opgave.



Figur 6.7: Rapid Application Development.

Det giver høj produktivitet at arbejde så fokuseret og intenst. Der er omvendt en risiko for at arbejde med skyklapper på, og der er risiko for manglende integration med andre systemer og forretningsområder.

Rapid Application Prototyping (RAP)

RAP er i sin grundstruktur versionsvis udvikling. Men RAP lægger mere vægt på det prototypeudviklende element og ikke mindst på arbejdssprocessen i forbindelse med at udvikle og vurdere prototyper. RAP er meget dynamisk, og som eksempel kan man netop bruge den udviklingsmodel, der hedder DSDM, eller andre 'Agile' metoder til hurtigt at komme i gang med at fremstille prototyper. DSDM er en metode/model, der også bygger på en veldefineret strategi, nemlig et dynamisk samarbejde mellem udviklere og interesserter støttet af en række navngivne prototyper. DSDM ser sig selv som et RAD-koncept, men i forhold til andre RAD-koncepter er det især det prototypeudviklende element, der står centralt. Et rigtigt RAD-forløb skal sammenlignes med at affyre en kanon mod et mål. Der er ingen vej tilbage, og alle kræfter er sat ind på at ramme målet i første forsøg. RAP er trinvis forfining og ændrede modeller.

Det er nemt at definere og beskrive emner teoretisk, fx modeller og strategier i hvert sit kapitel. Men når man ser de praktiske tilbud fra leverandører, er de i nogle tilfælde – som for eksempel med DSDM – samlet i samme produkt. Det er helt fint, så længe begge dele er velegnede i forhold til systemet der skal udvikles, og arbejdssprocessen det skal udvikles under.

Genbrug

Genbrug bruges til to formål:

- ▶ Strategisk at skabe nye komponenter til fremtidens systemer
- ▶ Taktisk at genbruge/købe allerede udviklede komponenter.

Det er to meget forskellige formål. De har dog det tilfælles, at det skal besluttes tidligt i projektet. Og at bruger/køber/markedet skal indstille sig på, at løsningen ikke er 100 % skræddersyet.

Anvendelse af genbrug spiller især en rolle som produktivitetsmiddel. Skabelsen af genbrugelige komponenter skaber fremtidens produktivitet, men kan koste tid i det skabende projekt.

Genbruget kan omfatte forskellige produkter: design, kode, databeskrivelser, Design Patterns, Use Cases, testdata, planer, varigheder i form af

realiserede estimer, kravspecifikationer, dokumentation, risikoanalyser (pas på med dét) eller noget helt tolvtæ.

Genbrug er ofte et supplerende element i en af de andre strategier. Men det er nødvendigt at give det en strategisk status for ad den vej at få det taktiske apparat på plads. Ellers risikerer det hele at blive skønne tanker i stedet for skøn virkelighed.

Skelettet i projektet er ret specielt, som det fremgår af figur 6.8. Efter en overordnet foranalyse gennemføres et designstudium, der undersøger, hvilke færdige komponenter der er til rådighed, samt hvilke komponenter der kan skabes til fremtidig genbrug. Det følges op med en beslutning, evt. med kompromiser og „80-pct.'s løsninger“. Så følger en almindelig analyse, der dokumenterer funktionaliteten. Det kræver en endelig beslutning, og derfra kører projektet rutinemæssigt.

Pointen er at uden aktive beslutninger opnås der mindre end det mulige genbrug.

Anvendelighed

Alle typer projekter, undtagen „Mission critical“
Stram Time-to-Market (kun anvendelse, ikke skabelse af genbrug)
Stram økonomi (kun anvendelse, ikke skabelse af genbrug)

Fordele

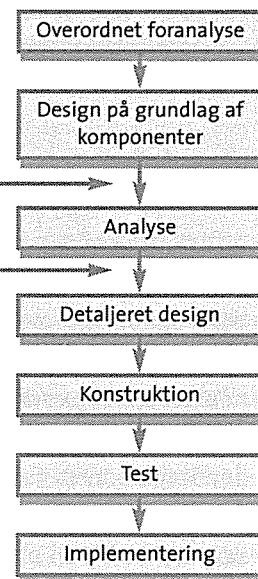
Høj produktivitet
Standardisering

Ulemper

„80%'s løsninger“
Vanskelige beslutninger i starten af projektet

Anbefalinger

Opbyg en kritisk masse af muligt genbrug
Fokusér meget på emnet fra start til slut, det må ikke drukne



Figur 6.8: Genbrug.

Teststrategi

Målsætningen for testens effektivitet fastlægges på grundlag af købernes behov for korrekthed og pålidelighed, konkurrenternes formåen og evner på området og leverandørorganisationens behov for at blive opfattet som pålidelig. Fejlfri produkter er altid at foretrække, men da der er en pris på

det meste her i verden, må projektlederen tage udgangspunkt i, hvor mange penge der bevilges til test.

Projektlederen kan dog som minimum lade være med selv at lægge op til en mangelfuld test. Det gøres ellers let ved at vente med at planlægge testen til sidste øjeblik, lade som om der ikke vil være fejl at finde, lade være med at bruge andres erfaringer og lade være med at opstille et budget til test.

Der er tre basisstrategier: den traditionelle med at vente til sidst, V-modellen der planlægger tidligst muligt, og „o-fejl“ der udfører tidligst muligt.

Den traditionelle

Testen både planlægges og udføres sidst i udviklingsforløbet. Strategien er velegnet til mindre opgaver på velkendte forretningsområder. Fordelen er at testen kan baseres på et færdigt system. Ulempen er at det er dyrere og vanskeligere at rette fejl i et færdigt system. Reelt er strategien derfor kun omkostningseffektiv, når der ikke findes fejl af anden type end lidt tilpassninger af tekster eller finpudsning af skærmbilleder og uddata.

I praksis betyder det at testaktiviteterne først optræder på projektplanen efter konstruktionsfasen.

V-modellen

Testen planlægges parallelt med udviklingen. Når testen kan planlægges tidligt, er det i sig selv et udtryk for, at objekt/data/funktionsmodeller er forståelige og konsistente. Accepttesten planlægges detaljeret efter foranalySEN, bruger- og systemtest planlægges detaljeret efter analysefasen, og integrationstesten planlægges detaljeret efter designfasen. Komponenttesten ligger dog uændret i forhold til den traditionelle strategi. Testens praktiske udførelse er altså uændret i forhold til den traditionelle, dvs. den ligger sidst i forløbet. Denne strategi er grundlaget for IEEE's standarder for test. Det betyder at testfaserne planlægges modsat den rækkefølge, de udføres i. Den største forskel for testerne er, at planlægningen baseres på de tidlige modeller af systemet, ikke på det færdige system. System- og brugertest baseres på data- og funktionsmodeller, og for OO-systemer på de tre grundmodeller: objektmodellen, der fokuserer på klassehierarki og -relationer, kommunikationsmodellen, der viser „messages“, og den funktionelle model, der viser Events/Use Cases. V-modellen udvides med reviews og inspektioner, og afhængigt af hvor tidligt og hvor omfattende, bevæger projektet sig over mod „o-fejl“.

Planlægning af en testfase kan ske før, samtidig med eller lige efter en udviklingsfase. I figur 6.9 er testen lagt samtidig med den relevante udviklingsfase.

O-fejl

Kaldes også „zero-defect“. I denne strategi hindres fejlene i overhovedet at opstå. **Verifikation og validering udføres parallelt med udvikling.** Ulempen er prisen; den er høj. Resultatet er til gengæld et fejlfrit, stabilt og vedligeholdesesvenligt system. Strategien sørger også for at leveringsdatoer overholdes. Projektlederen vil ikke opleve at testtiden forkortes fordi udviklingen forsinkes, og leveringsdatoen alligevel skal overholdes. Udfordringen i denne strategi er at kunne teste i hver udviklingsfase i en grad af detaljering og pålidelighed, der finder alle fejl. Produktet skal være fejlfrit på ethvert givet tidspunkt. Det betyder at testerne, herunder brugerne/køberne, skal indstille sig på, at det er en rigtig test, der udføres. De skal behandle den seriøst og bruge testscenarier, der er lige så gennemarbejdede som i en traditionel test på et færdigt system.

Strategien kræver omfattende brug af prototyper, formelle metoder og løbende inspektioner og verifikationer.

Opsummering af teststrategier

Den korte udgave af teststrategier lyder således: I et traditionelt testforløb både planlægges og udføres test, som man gjorde det i gamle dage, se figur 6.9. V-modellen er den for tiden mest anvendte eller i det mindste den mest ønskede. I et V-forløb planlægges testens faser i modsat rækkefølge af den traditionelle, men udførelsen er som vanligt. Den udvidede V-model forsynes med et passende antal reviews og inspektioner. Der „lånes“ så meget fra „o-fejl“-strategien som budgettet tillader. Den ægte „o-fejl“-strategi er et opgør med fortiden, idet både planlægning og gennemførelse af test sker i „modsat“ rækkefølge af vanetænkningen.

Vær opmærksom på at man får et traditionelt forløb, hvis der ikke gøres noget aktivt. Se meget mere om teststrategier i litteraturlistens ‘Strukturet test’.

Betatesten er kun vist skematisk. Nogle foretrækker betatesten før accepttest, andre efter. Den er her vist uden planlægning, men det er selvfølgelig tilladt at planlægge sin betatest ordentligt. Det er bare ikke så afgørende i strategisk sammenhæng.

STYRINGSPROCESSEN

10

- 10.1 Indledning
- 10.2 RUP – konceptet
- 10.3 RUP i praksis
- 10.4 Faserne
- 10.5 Workflows
- 10.6 Rollerne
- 10.7 Artifacts
- 10.8 Use Cases

10.1 Indledning

Projektlederen kan anlægge flere indfaldsvinkler, når det drejer sig om at vælge hvilke styringsprocesser der skal vælges til et projekt.

"Veldefineret" vs.
agile tilgang

Den ene indfaldsvinkel er en veldefineret tilgang kontra en 'Agile' tilgang. Hvis man vælger en veldefineret indfaldsvinkel tages der udgangspunkt i, at projektet kan planlægges på forhånd ved hjælp af aktiviteter med indhold, navne, struktur og resultat, der er kendt på forhånd. Den bedste model for en sådan tilgang er Rational Unified Process (RUP). Den eksisterer komplet med rolle-, proces- og aktivitetsbeskrivelser. Et 'Agile' alternativ er Extreme Programming (XP). I det tilfælde lægges der mindre vægt på det formelle, og der satses i højere grad på kommunikation.

Agile betyder behændig eller smidig og bruges om den type udviklingsmetoder, som XP er udtryk for. Der er andre agile metoder end XP, for eksempel:

SCRUM, der supplerer XP fordi den i højere grad sigter på ledelsesproessen. SCRUM er velegnet til projekter med ustabile krav, konflikrende interesser og krav om effektiv levering af fejlfri software.

Crystal Clear, er medlem af en familie af processer. Alastair Cockburn, der er kendt fra sit arbejde med 'Use Cases', er den drivende kraft.

ASD, Adaptive Software Development, sigter på at tilpasse sig til situationen, gerne ved selvorganisering på tværs af den konkrete organisation, og på tværs af virtuelle teams.

DSDM, Dynamic Systems Development Method, er en gammel kending. Den er ikke udbygget med de teknikker og mekanismer, der gør den rigtig 'Agile'. Dens væsentligste bidrag er 'Prototyping' og 'Timeboxing'.

Fælles for 'Agile' metoder er, at de er iterative, at de er drevet af konkrete brugerønsker, at de er 'Timeboxed', at de har til formål at levere fejlfrit software, at de er drevet af risici, ved at kende og forholde sig til dem, og at de er tolerante over for ændringer. Endelig må vi sige, at de generelt er kommunikative, ved at mennesker er i dialog, snarere end at der kommunikeres ved udarbejdelse af dokumenter. Selvfølgelig er der dokumentation, men ikke som fasedokumentation i en traditionel projektmodel.

Den anden indfaldsvinkel er vandfald kontra en iterativ fremgangsmåde. Måske er det en lidt gammeldags diskussion, fordi der i dag er bred enighed om at en iterativ metode er at foretrække. Men når man kigger under kålerhjelmen på flot iterativt planlagte projekter, ser man desværre ofte traditionelle vandfaldsprojekter, blot med en opdeling i mindre delprojekter. De udgør således stadigt et stort samlet projekt, der faktisk har lige stor, og på visse punkter større, risiko for at fejle. Problemet er, at man stadig har det store mål for øje, og at alle delprojekterne skal passe sammen. Det giver en dyb afhængighed, og de førstomtalte 'visse' punkter drejer sig om, at man let risikerer at miste overblikket i delprojekterne, fordi man grundlæggende ikke er frigjort fra at have et vandfaldsorienteret mål.

Vandfald vs.
iterativt forløb

I et ægte iterativt forløb udbygges med de på ethvert tidspunkt nyttigste funktioner og faciliteter. Der foregår en løbende formulering af krav og ønsker, og der foregår en løbende prioritering af indholdet af den næste iteration. Det gør en kolossal forskel i praksis, om det er ægte iterativt.

'Ægte' iterativt
forløb

Den tredje vinkel er, om der er tale om en 'Over-the-Wall' specifikation, eller om man taler sammen med kunden undervejs. 'Over-the-Wall' hentyder til, at kunden skriver en kravspecifikation, og kaster den hen over muren til udviklerne. Inde bag muren udvikles systemet, som udviklerne derafter kaster tilbage til kunden. Det forudsætter en kravspecifikation, der definerer det fulde indhold af systemet.

"Over the wall" vs.
kundekontakt
projekt → system → kunde

Der er flere vigtige pointer i valget. I en 'Over-the-Wall' taler parterne ikke sammen, de ser end ikke hinanden. Det har den ulempe at det ikke er muligt at ændre på noget, samtidigt med at eventuelle fejl og mangler i kravene bliver til fejl og mangler i produktet. Men der er på den anden side to potentielle fordele. Kunden tvinges til at detaljere og tydeliggøre kravene, fordi vi gør tydeligt opmærksom på, at vi kaster det tilbage, vi ser i kravene. For det andet tvinger det kunden til at specificere testen, både med testaktiviteter og -cases. 'Tvinger' er så voldsomt et ord, men projektet kan gøre stærkt opmærksom på at det er en god ide, og understrege at produktet godkendes med den test, der følger med.

Der har i tidens løb været eksperimenteret med mange forskellige måder at specificere krav på. 'Over-the-Wall' og XP repræsenterer på sin vis to yderpunkter, og et projekt kan vælge at lægge sig et sted imellem dem. Erfaringerne siger dog at det ofte giver et utilfredsstillende resultat. Det lyder i udgangspunktet tillokkende: lad os vælge et kompromis, lad os tage det bedste fra begge verdener. Men det holder ikke. For den del af projektet,

OHW vs. agile
Over-the-Wall
vs.
Agile
Over-the-Wall
vs.
Agile
Over-the-Wall

der handler om at afdække kravene, er det bedre at gøre en af delene og gøre det fuldt og helt.

Hvis valget er en 'Over-the-Wall' så kan den sagtens blive succesfuld. Men så skal der skrives den kravspecifikation, der skal til, og det skal gøres ordentligt. Man skal i hvert fald ikke vælge kompromiset, hvis det kun er for at springe over, hvor gærdet er lavest. Så bliver det noget i retning af at skrive den del af kravspecifikationen, der er lettest og mest indlysende, og så overlade det til projektdeltagerne at rede trådene ud senere. Men uden at give dem muligheden for at gøre det effektivt, enten i form af tid eller brugermedvirken.

Det er min erfaring, at udviklingsorganisationer har brug for begge typer af projekter. Der er både behov for projekter, der bygger på en 'Over-the-Wall' kravspecifikation, og for at gennemføre projekter der benytter en ægte itera-tiv fremgangsmåde.

Omvendt, hvis valget er XP, så gør det helt og fuldt. Således som det beskrives senere i de 12 discipliner i kapitel 12.

I litteraturen forbindes 'Over-the-Wall' med veldefinerede procedurer, og 'Agile' forbindes med den modsatte situation hvor parterne udarbejder krav og løsninger undervejs. Men i praksis er situationen mere flydende. Der er intet i vejen for at skrive en detaljeret kravspecifikation, og derefter brug 'Agile' metoder internt i projektet, som det underliggende produktionsapparat.

På www.agilemanifesto.org kan man læse at man i 'Agile' verdenen sætter:

- ▶ Individer og dialog højere end procesbeskrivelser og værktøjer
- ▶ Software der kan ses i arbejde højere end omfattende dokumentation
- ▶ Samarbejde med kunden højere end kontraktforhandlinger
- ▶ Ændringsparathed højere end at følge en plan

Men der sker i dag et livligt lån, metoderne imellem. I et RUP/UML projekt kan man jo sagtens bruge Pair-Programming, Pair-Testing og daglig integration fra Extreme Programming. Rational har udarbejdet en del dokumentation, dels hvor RUP sammenlignes med XP, og dels retningslinier for at anvende RUP'en 'Agile'.

Organisationer der vælger systematisk brug af RUP/UML, har taget et stort skridt op ad CMMI. Niveau 2 og halvdelen af niveau 3 er dækket.

Resten af dette kapitel gennemgår RUP som et eksempel på en veldefineret projektmodel. De to følgende kapitler gennemgår henholdsvis SCRUM og 'Extreme Programming'.

10.2 RUP – konceptet

Rational Unified Process (RUP) fra Rational, nu en del af IBM, er en projektmodel. Den er et eksempel på formel udvikling, der bygger på et veldefineret udviklingsforløb.

RUP giver projektledere mange fordele. Den har en stærk styring af krav, fokuserer på at styre risici, den er iterativ og støttes fuldt ud af værktøjer. Værktøjerne er integrerede og kan give sporbarhed fra krav til testcases og retur.

RUP er moderne med hensyn til test. Der er løbende verifikation og validering, men det er på den anden side også nødvendigt, når der satses på et iterativt forløb.

Som andre moderne modeller er den dokumenteret elektronisk og tilgængelig med en browser. Skabeloner er lette at nå via links, og umiddelbart anvendelige.

RUP støttes direkte af Unified Modeling Language (UML), der visuelt modellerer systemet der udvikles. UML modellerne kan både bruges til at kommunikere med forretningssiden og indbyrdes mellem udviklere.

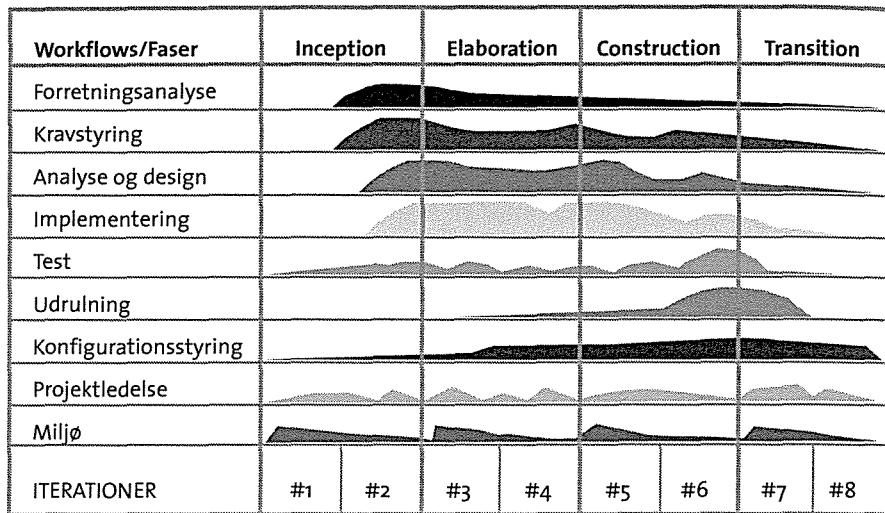
Vi har altså at gøre med en samlet model af udviklingsprocessen, RUP, der går hånd i hånd med en samlet model af systemet, UML. De kan bruges hver for sig med stor nytte, og sammen med endnu større udbytte.

10.3 RUP i praksis

RUP består af:

- ▶ Faser, hvoraf der er 4.
- ▶ Processer, kaldet 'Workflows', hvoraf der er 9.
- ▶ Roller, hvoraf der er 33.
- ▶ Aktiviteter, og dem er der mange af.
- ▶ Produkter, kaldet 'Artifacts', hvoraf der er 63.
- ▶ Skabeloner, kaldet 'Templates', og dem er der mange af.
- ▶ Retningslinier, kaldet 'Guidelines', og dem er der mange af.

Figur 10.1 viser RUP, dens faser og workflows.



Figur 10.1: Rational Unified Process.

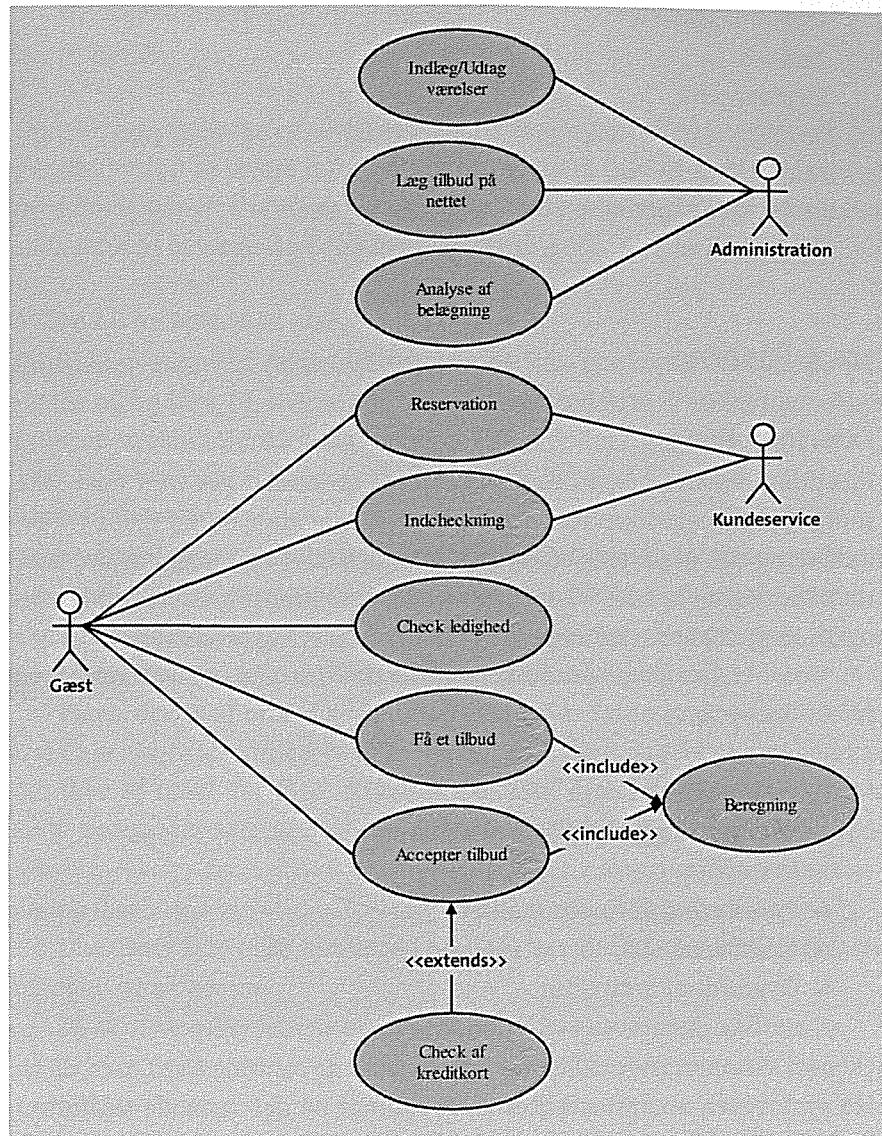
De fire faser hedder 'Inception', 'Elaboration', 'Construction' og 'Transition', og er ikke oversat i denne tekst.

Når projektlederen beslutter at en fase kan lukkes, fordi en eller flere iterationer har frembragt et grundlag for den næste fase, afholdes et review. Det kan være i form af en workshop, der formelt afgør om milepælen er nået. Afslutningen af en fase repræsenterer en vigtig milepæl i projektet. Faserne er det tidsmæssige forløb. Langs med faserne gennemfører projektet et antal iterationer. Projektets egen produktionsplan er derfor en iterationsplan. I en iteration udføres det arbejde, der defineres af 'workflows'. Samtlige 'workflows' kan være aktive i en iteration.

Iterationer har meget forskellig længde. I mindre projekter varer de få uger, i større projekter kan iterationer vare 2-4 måneder.

Hele processen drives af Use Cases (brugsmønstre), der er modelleret i en Use Case Model. Use Cases eksisterer begrebsmæssigt både i RUP og UML. Figur 10.2 viser en Use Case model, og sidst i dette kapitel er et eksempel på en Use Case. Projektlederen kan med fordel bruge Use Cases som det drivende element i både udvikling og test. Til udvikling repræsenterer de et aftalegrundlag med omverdenen, og til test er de grundlag for at definere testscenarier.

Hotellet



Figur 10.2: Use Case Model.

De engelske betegnelser på 'Workflows' er oversat i denne tekst:

RUP
Business Modeling
Requirements

Dansk
Forretningsanalyse
Kravstyring

Analysis & Design	Analyse og design
Implementation	Implementering
Test	Test
Deployment	Udrulning
Configuration & Change Management	Konfigurations- og ændringsstyring
Project Management	Projektledelse
Environment	Miljø

10.4 Faserne

Fase: Inception

Inception betyder begyndelse, så det er jo et relativt generelt navn. Men indholdet er meget specifikt. **Fasens formål er at etablere en vision for produktet**, og det gøres meget konkret, blandt andet ved hjælp af det dokument der netop hedder 'Vision'. Det beskriver omfang, indhold og økonomi og en lang række andre væsentlige forhold omkring produktet.

Visionen har afsnit der dokumenterer de fleste af de aktiviteter, der udføres i 'Inception':

- ▶ Problembeskrivelse
- ▶ Afgrænsning af produktets omfang
- ▶ Indholdet
- ▶ Prioritering af funktionalitet
- ▶ Risikoanalyse
- ▶ Overordnet arkitektur.

Ud over visionen fremstilles der en liste over de centrale Use Cases, de første prototyper, accepttestkriterier, et estimat på det samlede tidsforbrug for hele projektet og et budget.

Fasen kan i mange tilfælde gennemføres i en enkelt iteration, men det kan fint være i flere. Til at drive arbejdet i fasen bruges især disse processer i RUP'en:

- ▶ Forretningsanalyse
- ▶ Kravstyring
- ▶ Analyse og design

- Test
- Projektledelse
- Miljø.

Opdelingen i iterationer allerede i denne første fase er en stor fordel. Det gør det muligt at levere information om systemet ud af projektet og dermed holde gang i kvalitetsstyringen. Det er direkte med til at reducere risikoen for at bevæge sig i den forkerte retning.

Iterativ inception

Milepæl: Inception er slut

Milepælen sikrer at nøgleinteressenterne kan godkende 'Visionen', estimerer og budget, og checker at nøgleinteressenterne er indbyrdes enige. Milepælen skal forenklet sagt sikre, at målet er det rigtige.

Fase: Elaboration

'Elaboration' betyder at sætte detaljer på. Så det er jo også et passende generelt navn. Men fasen skal styres særdeles håndfast af projektlederen. I elaboreringsprocessen er der mange muligheder for misforståelser, der senere viser sig som fejl i produktet, når systemet bruges.

Fasen drives af arbejdet med Use Cases. Når fasen er slut skal 80 % af Use Cases være identificeret, og indholdet skal være skrevet så detaljeret, at der ikke er risiko for misforståelser. Det skal desuden være de vigtigste 80 %. Betragt de 20 % som dem, det ikke er nødvendigt at se på nu, fordi de ikke indebærer risici. Risici er i det hele taget i centrum for processen i denne fase.

80% UC's
beskrivet.

Use Cases skal også foreligge i en Use Case model, der viser sammenhængen og aktørerne.

UC diagram

Til at drive arbejdet i fasen bruges især disse processer i RUP'en:

- Kravstyring
- Analyse og design
- Implementering
- Test
- Projektledelse
- Miljø.

Fasen omfatter arbejde med de centrale modeller i UML: Use Cases, klasse-modeller, sekvensdiagrammer, aktivitetsdiagrammer og samarbejdsdia-

Dette er en teknisk dokumentation

grammer. Udfordringen i at komme helskindet igennem elaborering er langt større end i 'Inception'. Mængden af dokumentation er vokset. Det er i denne fase, at man høster fordelene ved at bruge en formel metode; eller smider dem væk. Projektlederen kan simpelt hen ikke investere for meget tid og kræfter i kvalitetssikring. Reviews og inspektioner skal på banen, med så mange interesser og aktører, som projektlederen kan involvere. En af udfordringerne er at få kontrolleret, at kravene til de ikke-funktionelle egenskaber er opfyldt, eller i det mindste kunne konstatere at projektet er på rette vej til at opfylde dem. UML er perfekt til at dokumentere det funktionelle indhold. Men hvad med de ikke-funktionelle egenskaber? Egenskaberne omfatter så vigtige forhold som robusthed, sikkerhed, brugervenlighed og performance. Det er fristende at vente til konstruktionsfasen med at kontrollere dem, men da kan det være for sent. Processen 'Kravstyring' indeholder de rette og nødvendige trin og retningslinier til at specificere kravene til egenskaberne. Projektlederen skal sørge for at de bliver brugt.

Det er en god ide at bruge risikoanalyse til kvalitetsstyring. Med udgangspunkt i de mulige risici kan reviews, inspektioner og demonstratiorer ved hjælp af prototyper, bruges til at checke om der er opstået fejl eller misforståelser.

Milepæl: Elaboration er slut

Milepælen sikrer at interesserne er enige om at indholdet er det rigtige. Milepælen skal afgøre, at der er sammenhængende dokumentation, der både er udviklet og dokumenteret på den rigtige måde.

Fase: Construction

Konstruktionsfasen skal sørge for at levere koden, tabeller, ASP'en, og de andre fysiske dele. Målet er at levere noget hurtigst muligt, der kan testes af brugere. Konstruktionen sker på grundlag af diagrammerne og modellerne, der er dokumenterede ved hjælp af UML.

Der er brug for at tænke både horisontalt og vertikalt. Det betyder at projektlederen skal afgøre hvor detaljeret, der skal arbejdes på langs af projektmodellens faser, og hvor omfattende iterationerne skal være. Et godt råd er at detaljere den foregående elaboreringsfase maksimalt. At detaljere modellerne til det yderste før der tages fat på konstruktion. For således at have fjernet risici, og reducere konstruktionsfasen til det simplest mulige. Men for ikke at ende op i en vandfaldstankegang, er der derfor brug for at

arbejde med korte iterationer. For således at levere dele af systemet tidligt. Det gode råd er altså at modellere detaljeret, men på en lille del af systemet ad gangen.

Samtlige workflows kan være aktive, afhængigt af iterationernes dybde og bredde, men disse workflows vil naturligt være aktive:

- Implementering
- Test
- Udrulning.

De færdige dele skal testes med de gængse testformer, herunder komponent- og integrationstest. Når iterationerne har leveret et sammenhængende produkt kan der også gennemføres en systemtest. Derefter kan der arrangeres alfatest, hvor brugere tester i udviklernes miljø. Projektlederen skal desuden holde øje med hvornår der er betakandidater, for yderligere at give brugerne mulighed for at anvende systemet i det endelige miljø.

Milepæl: Construction er slut

Efter hver iteration undersøges det om denne milepæl er nået. Det er den, når der foreligger et sammenhængende produkt, med en kvalitet der er god nok som betakandidat. Milepælen skal sikre, at systemets kvalitet er så høj, at det ikke vil være spild af brugernes tid.

Fase: Transition

Fasens formål er implementering. Når et antal iterationer har afsluttet konstruktion, gennemføres en eller flere 'Transition' iterationer. Det kan omfatte en samlet alfatest, samt en beta- og accepttest. Før, under eller efter disse afsluttende test, kan der foretages en produktmodning i form af afpudsniner, fejlrettelser, dokumentation, konverteringer og integration til kørende systemer. Implementeringen kan også omfatte aktiviteter uden for projektet, typisk brugeruddannelse, opdatering af helpdesk, og de andre aktiviteter, der er nødvendige, når produktet skal bruges i praksis.

Igen kan samtlige workflows teoretisk være aktive, alene på grund af rettelser og produktmodning, men de centrale er:

- Implementering
- Test
- Udrulning.

Milepæl: Transition

Hele projektet er færdigt, når den oprindelige 'Vision' er opfyldt, og en godkendt accepttest dokumenterer det.

10.5. Workflows

Workflows er det bærende element i en udviklingsmodel som RUP, der baserer sig på veldefinerede processer. Workflows binder roller, aktiviteter og 'Artifacts' sammen. Der er glimrende oversigter i RUP, der både viser sammenhængen med udgangspunkt i rollerne, og sammenhængen med udgangspunkt i 'Artifacts'.

Til hvert workflow er der desuden et antal vejledninger, der er en uvurderlig hjælp i gennemførelsen af aktiviteterne i workflowet. Endelig er der skabeloner til at dokumentere resultaterne, der produceres i workflowet.

Alle workflows er teoretisk mulige at bruge i hver fase. Det er kun 'Udrulning' der ikke er aktiv i 'Inception'. Men i praksis har hver proces sin tyngde i forhold til en eller to faser.

Workflow: Forretningsanalyse

Forretningsanalysen kan være begrænset til videreudvikling af systemer indenfor et eksisterende forretningsområde, eller det kan være i lidt større målestok med videreudvikling af et forretningsområde, eller yderligere en bredt anlagt forretningsanalyse af et hele forretningen. Der er tilsvarende forskellige veje igennem workflowet, svarende til ambitionsniveaueret.

I alle tilfælde bruges resultatet af forretningsanalysen som grundlag for at definere kravene til IT-systemerne.

Til dette workflow er der 35 vejledninger til rådighed. Det er mange, men der skal være støtte til forretningsmodellering, fremstilling af en Use Case model, og andre omfattende aktiviteter.

Analysen kan dokumenteres ved hjælp af UML diagrammerne:

- ▶ Use Case Model
- ▶ Use Cases
- ▶ Klassemodel
- ▶ Objektmodel
- ▶ Samarbejdsdiagrammer (Collaboration Diagrams)
- ▶ Sekvensdiagrammer

- Aktivitetsdiagrammer
- Tilstandsdigrammer.

Ved at benytte UML til at dokumentere forretningsanalysen, er der fuld værktøjssupport. Modellerne kan bruges til at forstå den organisation, som systemerne skal bruges af, og til at illuminere de problemer i den, som IT-systemerne skal løse. Modellerne sørger for at alle interessenter synliggøres, at interessenterne får samme grundlag at agere på, og at interesser og projektgruppen kan arbejde på et fælles grundlag.

Workflow: Kravstyring

Kravstyring er et godt eksempel på hvordan et workflow arbejder på tværs af alle faserne. I 'Inception' er der brug for problem- og interressentanalyse. I 'Elaboration' er der behov for systemanalyse, og i alle faser er der brug for styring af krav i en omskiftelig verden. Workflowet omfatter alle disse aktiviteter, og der er vejledninger til at støtte dem. Af de 17 vejledninger, der er til rådighed, er der dog gengangere fra 'Forretningsanalyse'.

Visionen fremstilles i dette workflow, som fundament for kravspecifikationen og Use Case modellen.

Kravspecifikationen er i mange projekter det konkrete aftalegrundlag, og den skal derfor udvikles og vedligeholdes i hele projektforløbet. Kravspecifikationen er opdrags-giverens mulighed for at specificere både de funktionelle og de ikke-funktionelle krav. De sidstnævnte, der omfatter så vigtige forhold som brugervenlighed, ydeevne og sikkerhed, dokumenteres i RUP'en i 'Supplementary Specification'. Man kan vælge ISO 9126 som struktur i de ikke-funktionelle krav, hvis man foretrækker det. De funktionelle krav kan med fordel beskrives i en Use Case Model, og i specifikke Use Cases. Projektlederen bruger resultatet af dette workflow til at afgrænse systemet, estimere udviklingstiden og planlægge iterationerne.

Rational (IBM) leverer værktøjet RequisitePro til at støtte denne proces. Det er også i denne proces, og ved hjælp af værktøjer, at grundlaget skabes for sporbarhed mellem krav og testcases.

Jeg har selv brugt visionsdokumentet i en række projekter, herunder både produkt- og procesudvikling. Det er ganske enkelt fremragende, og det holder i praktisk brug.

Jeg kan anbefale projektledere af enhver type projekter, at de satser på at få en god proces omkring udarbejdelsen af visionen. Inddrag nøgleinteressenterne i det omfang, de er kendt, og sats på en høj kvalitet, ved systematisk at få visionen synliggjort og reviewet.

Workflow: Analyse og Design

Dette workflow går også på tværs af alle faserne. I 'Inception' starter det med arkitektur, i 'Elaboration' handler det om omfattende analyser, der dokumenteres ved hjælp af diagrammerne i UML. Selv i 'Transition' kan der være brug for at dokumentere de sidste detaljer på komponenterne. Men der er en overvægt af arbejde med dette workflow i 'Elaboration'. Workflowet støttes af 31 vejledninger, og det er her at den store mængde af detaljerede UML diagrammer produceres.

Det er i dette workflow, og i fasen 'Elaboration' at projektlederen træffer beslutninger om, hvor langt projektet skal arbejde i detaljer i UML diagrammerne. Projektlederen skal for eksempel tage stilling til mængden og detaljeringen af aktivitetsdiagrammer. Man kan vælge at diagrammere systemets logik ud i detaljer, der gør kodningen i 'Construction' til et enklere arbejde. Det er også i dette workflow at systemet tager fysisk form i klasser og leverancer af 'packages'.

Workflow: Implementering

Dette workflow beskæftiger sig med at udvikle komponenterne. De 8 vejledninger støtter den konkrete udvikling. Tyngden ligger i konstruktionsfasen, men workflowet er aktivt i mindst én iteration i 'Elaboration' for at definere en arkitektur til komponenterne.

Det er relevant at arbejde med arkitekturen i den fysiske implementering fra første færd i projektet. Hvis man satser på komponentbaseret udvikling, er det ikke noget, der kommer af sig selv, og det kan være for sent at starte på konstruktionstidspunktet. Hvis man arbejder med webservices som arkitektur, for eksempel i .net arkitekturen, skal fundamentet støbes i de tidlige iterationer.

Der er mange veje til prototyper, og mange typer af dem, og der er brug for at arbejde med dem i dette workflow tidligt i projektets iterationer.

Workflowet omfatter også komponenttest. Og de testede komponenter skal integreres, da det først er efter en succesfuld integration med andre komponenter, at man har sikkerhed for, at den fungerer, og dermed kan tælle med i fremdriften.

Workflow: Test

Test er et gennemgående workflow, der er særdeles aktivt i alle 4 faser. Der er 9 vejledninger, og blot den første der nævnes, Test Cases, er meget værdifulde for de projektledere, der ikke har solid praktisk erfaring med alle

dele af et testforløb. Workflowet gennemgår de forskellige testformer, og går i dybden med teknikkerne i at definere testcases på grundlag af Use Cases.

Workflowet sikrer at kvaliteten er på det ønskede niveau, på ethvert givet tidspunkt i projektets forløb. Hvis projektet bruger V-modellen til test, er dette workflow aktivt med til tidlig planlægning af testens indhold og i review af 'Artifacts'. Hvis der satses på en o-fejl strategi, er der yderligere brug for prototyper, der kan gennemføres egentlige tests på.

I takt med integrationen af komponenter, testes systemet i sin helhed. Enten i form af systemtest eller brugertest. Kvaliteten måles, og det er en væsentlig del af workflowet at producere grundlaget for projektlederens rapportering af kvaliteten af systemet. Det omfatter både den funktionelle test, typisk baseret på Use Cases, og den ikke-funktionelle test baseret på 'Supplementary Specification'.

Endelig er der også det lidt kedelige aspekt, at der findes fejl. Det betyder at de rettes, og at den pågældende del af systemet skal gentestes, samt at iterationen skal regressionstestes.

Det er afgjort en god ide at have en gruppe ISEB-certificerede testere i organisationen. Alternativt bør der entreres med eksterne uafhængige testere, for at være sikker på at testen er grebet rigtigt an. Det gammeldags synspunkt om, at alle vel kan finde ud af at teste, bliver let unødvendigt kostbart.

Som med mange af de andre workflows skal det være aktivt i de tidlige iterationer. Man kan spørge sig selv, om alle workflows er aktive hele tiden, og svaret er ja. Det reducerer risici i de fleste projekter at arbejde med mindre iterationer og at arbejde parallelt i alle workflows.

Workflow: Udrulning

Dette workflow beskriver aktiviteter inden for flere væsentlige områder. Det ene er de aktiviteter der skal til for at sætte brugerne i stand til at bruge produktet. Det er ikke altid nok med et system, der kan også være brug for at 'rulle' uddannelse ud, og der kan være brug for supportfunktioner. Det andet store område i dette workflow er de afsluttende test. Det er alfatesten, hvor brugerne tester i et produktionslignende miljø, og betatesten hvor der testes i det endelige produktionsmiljø.

Der er kun en enkelt vejledning: 'Deployment Plan'.

Workflowet minder også projektlederen om de mange praktiske detal-

Min bog 'Struktureret test' går i dybden med teststrategierne. Der er også støtte til hente i IEEE 829-1998 standarden.

jer såsom distribution og installation af softwaret. Der kan også være behov for konverteringer af eksisterende data og software.

Workflow: Konfigurations- og ændringsstyring

Det er en støtteproces, sammen med de to efterfølgende. Der er ingen vejledninger til denne første, men den beskæftiger sig selvagt med versionsstyring af komponenterne, og med styring af ændringsønsker.

Emnet er simpelt, men nødvendigt, og praktisk taget altid misligholdt. Undskyld til dem der ikke misligholder det. Emnet er vigtigt, fordi verden sjældent står stille. Der er behov for at styre ændringer. Desuden findes der fejl i testen, og der opstår også af den grund nye versioner.

Budskabet er relativt simpelt: brug værktøjer eller dø.

Workflow: Projektledelse

Workflowet er vigtigt for projektlederen, fordi det giver konkrete anvisninger på hvordan projekterne skal planlægges og styres. Der er brug for en overordnet plan for hvordan faserne planlægges hen over tiden. Men RUP'en er ikke en vandfaldsmodel. I stedet brydes den op i en række mindre iterationer, og der er derfor brug for en detaljeret plan der viser iterationerne, under den overordnede plan.

En anden væsentlig del af 'Projektledelse' handler om risikostyring. Som en moderne metodik erkender RUP at succesfuld projektstyring baseres på intensiv risikostyring og aktiv imødegåelse af risici.

Der er 5 vejledninger, og der er for eksempel megen værdifuld hjælp at hente i 'Risk List'.

Workflowet opfordrer også til at bruge metrikker og målinger, hvad der også kendetegner en moderne metodik. Det er muligt at projektlederen synes at dette og hint er 'godt nok', men det er bedre at fortælle i tørre tal, hvor 'godt nok' det er. Der bør være metrikker for systemets omfang, systemets kvalitet, projektets fremdrift, testens dækning og testens kvalitet. Således at de nævnte forhold kan måles og dokumenteres i de ligeledes nævnte tørre tal.

Opdragsgiver og brugerorganisationen skal forsynes med viden om kvaliteten af systemet. De skal også kende kvaliteten og omfanget af testen. Herunder den fysiske dækning som testen har opnået. Det er ikke nok at vide hvor mange (eller for den sags skyld, hvor få) fejl der er fundet, hvis det kun er 50 % af det fysiske system, der er testet.

Workflowet omfatter også retningslinier for de basale styringselemente

ter i et projekt, herunder planlægning, estimering, opfølging og rapportering. Planlægningen vil typisk ske med iterationer, og således kræve de nævnte forskellige niveauer af planer, for både at tilgodese projektets egne behov, og planer der viser brugerne, hvornår der forventes leverancer, der kan nyttiggøres forretningsmæssigt.

Workflow: Miljø

Det handler om at få fastlagt hvordan projektet skal forløbe. Hvilke 'Artifacts' skal man vælge, hvordan skal reviews lægges til rette, etc. Der er desuden en række vejledninger der støtter håndteringen af hvert workflow. Der er i alt 15 vejledninger.

RUP er omfattende, både med hensyn til aktiviteter, processer, roller, 'Artifacts', skabeloner og vejledninger. Dette workflow støtter projektlederen i at få valgt de rigtige dele af RUP'en til det givne projekt. Desuden er der brug for at se på værktøjer, selvom det for de fleste organisationers vedkommende ligger som overordnede beslutninger uden for det enkelte projekt.

Når projektlederen har valgt hvordan RUP'en skal anvendes, skal projektdeltagerne uddannes tilsvarende. Roller og processer skal implementeres i praksis.

Jeg har kendt projekter hvor man var i den spændende situation, ikke at vide hvor meget der var dækket af testen. Man testede, men uden at måle dæknin gen af den. Projektlederne ændrede på dette i deres efterfølgende projekter.

10.6 Rollerne

Hver rolle i RUP er beskrevet med hensyn til hvilke artifacts rollen styrer eller bruger. Rollerne er afgørende for at drive processen i hvert workflow. Det er projektdeltagernes roller, og deres til rollen tilknyttede opgaver, ansvar og kompetence, der giver et workflow indhold.

Der er både fordele og ulemper ved roller. Fordelene er bedre styring af aktiviteterne, og bedre brug af vejledningerne. Ansvaret for de forskellige dele af udviklingsforløbet og af produktet synliggøres. Rollerne i RUP er desuden velbeskrevne, og umiddelbart til at anvende. Rollerne giver et konkret grundlag for at fordele arbejdet, og det letter kommunikationen i projektet og til og fra interesserne, at opgaver, ansvar og kompetence er på plads. Alt i alt er der store fordele at hente for projektlederen.

Hvad med ulemper? Der er kun en enkelt af betydning, men den er også så stor, at mange projekter ikke har magtet at forcere den: Indlærings tærskelen. Det koster tid og energi at implementere roller. Fordi det handler

om mere end opgaver, svarende til blot at uddele aktiviteter. Det handler også om ansvar og kompetence. Der skal bruges tid (meget) på at 'massere' ind i projektet, hvad det er for rollerettigheder og rolleansvar, der skal manifestere sig i det daglige arbejde. Risikoen, der opstår, er at projektlederen tror at rollerne fungerer af sig selv, hvorimod den mest almindelige konsekvens af roller, der ikke implementeres omhyggeligt, er, at der ikke sker noget.

Det sker som regel på grund af en undervurdering af, hvad der er nødvendigt at gøre for at få rollerne til at fungere. Det ser enkelt ud på RUP'ens diagrammer. Men det er ret sikkert, at hvis det eneste projektlederen gør, er at fortælle deltagerne hvilke roller, de skal dække, og det blot gøres, efterhånden som man når til at skulle nyttiggøre dem, ja så er der ikke stor sandsynlighed for at høste fordelene.

Hvad der kan gøres for at imødegå risikoen for at roller ikke fungerer:

- ▶ grundlæggende at fortælle om risikoen
- ▶ grundlæggende at fortælle om rollerne
- ▶ coaching, peer-to-peer, og det vil på dansk sige at projektlederen står til rådighed for sparring
- ▶ coaching, master-trainee, og det vil på dansk sige at projektlederen overvåger og følger op
- ▶ reviews, peer-to-peer, det vil på dansk sige kollegareview
- ▶ eksempler på hvordan roller udøves i det daglige
- ▶ uddannelse og træning
- ▶ tæt opfølgning i starten af projektet ved at spørge på opfølgningsmøder, eller tage fat i den enkelte, og det vil, afhængigt af måden det gøres på, ofte svare til coaching.

I softwaretest.dk lægger Klaus og jeg stor vægt på testroller. Det giver en helt anden kvalitet i planlægning og gennemførelse af test. I modsætning til at testen gennemføres som et bijob, når der er tid til det. Hvad der er tilfældet når testen arrangeres ved 'at vi tester alle sammen', som det kan udtrykkes så smukt.

10.7 Artifacts

'Artifacts' er alle de produkter der fremstilles. Det er modeller, dokumenter, kode, og alle de andre håndfaste ting. Deltagerne påtager sig altså veldefinerede roller, der udfører veldefinerede aktiviteter, med veldefinerede produkter som resultat.

RUP'en beskriver og anvender mere end 60 'Artifacts'. De fleste projekter vælger kun at arbejde med en del af dem.

RUP'en indeholder et 'Artifact Overview' med de væsentligste dokumentationsenheder, og sammenhængen imellem dem:

- ▶ Interessenternes ønsker
- ▶ Vision
- ▶ Forretningsmodel
- ▶ Risici
- ▶ Kravspecifikation, herunder en Use Case model og supplerende krav til egenskaber
- ▶ Begreber – forklaringer
- ▶ Udviklingsplan
- ▶ Udrulningsplan
- ▶ Arkitektur, software
- ▶ Analysemodel
- ▶ Designmodel
- ▶ Implementeringsmodel
- ▶ Testplan.

10.8 Use Cases

Use Cases er scenarier, udbygget med bestemte nøjere definerede oplysninger, og i et standardiseret format. Formelle scenarier, kan man kalde dem. På dansk bruges ofte 'Brugsmønster'.

En Use Case skal beskrive tre ting:

- ▶ et hændelsesforløb således som det vil foregå, når systemet bruges forretningsmæssigt, for eksempel: 'kunde beder om levering af vare'.
- ▶ brugerens reaktion på hændelsen, for eksempel: 'brugeren aktiverer skærbilledet til levering af vare'.
- ▶ den forventede reaktion fra systemet på det brugeren gør, for eksempel: 'viser skærbilledet xxx med customiseret brugeropsætning'.

En Use Case svarer til en enkelt forretningsmæssig hændelse. Den omfatter og beskriver alle varianter inden for 'hændelsen'. Det kan være særlige

Poul Staal Vinje

Projektledelse af systemudvikling

Nyt Teknisk Forlag, 3. udgave

Kapitel 3.1 - 3.4, s. 22 - 37 (Projektlederen)

Kapitel 3.7, s. 50 - 60 (Projektgruppen)

Kapitel 5.4, s. 83 - 92 (Risikoanalyse)

Kapitel 5.5, s. 94 (SWOT)

Kapitel 6.4, s. 137 - 140 (Aktivitetsplanlægning)

Kapitel 6.5, s. 140 - 141 (Tidsestimering)

Kapitel 6.7, s. 178 - 183 (Fysiske planer)

3.1 Generelt

Projektorganisationen i sin fuldt udviklede form er et yderst fleksibelt og effektivt instrument. Fleksibiliteten er nødvendig fordi opgavernes forskellighed kræver det. Fleksibiliteten skaber også nogle problemstillinger, der skal løses i det enkelte projekt.

Projektledelse er tæt knyttet til opgaven, der skal løses. De organisatoriske forhold bestemmes tilsvarende af opgavens karakter.

Afhængigt af opgaven kan man vælge at etablere en projektorganisation med en mindre del af de mulige elementer, eller man kan etablere en organisation i sin fuldt udviklede form.

Det enkelte organisatoriske element kan etableres mere eller mindre formelt afhængigt af dets betydning i det givne projekt. En referencegruppe kan eksempelvis etableres som en formel gruppe, hvor deltagerne samarbejder om fælles løsninger. Gruppen kan alternativt etableres relativt uformelt med enkeltpersoner, der yder ad hoc-bistand.

Fleksibiliteten rækker i sin yderste konsekvens fra kun at bestå af en projektleder, der bruger en del af sin tid på projektet og eventuelt gennemfører aktiviteter ved hjælp af underleverancer i andre afdelinger, og til at anvende den fuldt udviklede form hvor alle dele er bemandet med fuldtidsbevigede ressourcer.

Problemet der opstår som konsekvens af fleksibiliteten, er at styringsgrundlaget, udtrykt som ansvar, opgaver og kompetence, kan variere tilsvarende i forhold til opgaven, der skal løses.

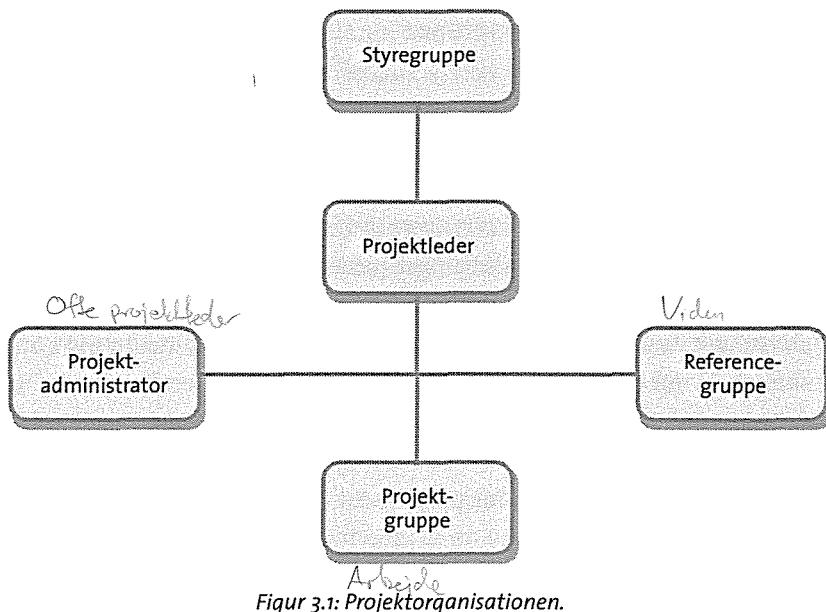
Forudsætningen er således at den til opgaven valgte struktur og kompetence er kendt og anerkendt af den øvrige organisation.

I praksis er det projektlederen, der sørger for at de daglige arbejdsgange i projektet fungerer, uanset den valgte organisatoriske form.

3.2 Den fuldt udviklede organisation

Figur 3.1 viser elementerne i den fuldt udviklede projektorganisation:

1. *Styregruppen*, hvis primære opgave er at løse de problemer der ligger uden for projektlederens kompetence. Styregroupens kompetence begynder der hvor projektlederens slutter. Styreguppen er den organisatoriske forankring for projektet.



Figur 3.1: Projektorganisationen.

2. *Projektlederen* der har ansvaret for at idegrundlaget realiseres samt har det daglige ansvar for projektgruppen.
3. *Projektadministrationen* der har ansvaret for planlægning og opfølging. Projektadministrationen varetages ofte af projektlederen, men kan udskilles i en selvstændig organisatorisk enhed, der refererer til projektlederen og organiseres som en stabsfunktion.
4. *Referencegruppen* er en gruppe af eksperter, som stiller viden til rådighed for projektet. Opgaverne er typisk af kortere varighed og har sjældent konstruerende karakter. Det er oftere vurderinger af produkter og indstillinger om det videre forløb. Gruppen kendes også under andre navne, fx som høringsgruppe. Der kan etableres flere referencegrupper hvis det skønnes hensigtsmæssigt. En gruppe kan bestå af teknikere, og en anden af brugerrepræsentanter. Referencegruppen refererer til projektlederen som stabsfunktion.
5. *Projektgruppen*, der har ansvaret for at gennemføre aktiviteterne på projektplanen. Deltagerne kan være hel- eller deltidsbeskæftigede i projektet, men refererer altid til projektlederen med hensyn til det faglige indhold af aktiviteterne.

Opgaver, ansvar og kompetence

Det er væsentligt for projektets succes, at alle opgaver bliver løst, og at ingen opgaver løses mere end ét sted i organisationen.

Lige så vigtigt er det at ansvaret i det enkelte element er kendt og accepteret. Både af enheden selv og af den øvrige organisation.

Endelig skal kompetencen være defineret. En projektorganisation skal være såvel fagligt som beslutningsmæssigt kompetent. Det betyder at den ideelle organisation kan løse alle opgaver kompetent og kan træffe alle beslutninger ledelsesmæssigt forsvarligt. Det er ikke altid muligt, men der skal etableres og tildeles en kompetence, der sikrer en kontinuerlig fremdrift i projektet. Der må ikke forekomme et stadig tilbagevendende behov for eksterne faglige bistand. Eller et stadigt tilbagevendende behov for at få truffet beslutninger uden for projektet.

3.3 Styregruppen

Styregrupper sikrer en kompetent beslutningsproces. Gruppens sammensætning skal repræsentere flere interesser, og beslutninger skal tænde mod afbalancering af interesse modsætninger og ikke tilgodese en enkelt interessen.

Projektet behøver under alle omstændigheder en organisatorisk forankring. Projektlederen har brug for at få truffet beslutninger uden for sin egen kompetence. Styregrupper er skabt netop til dette formål og er som sådan en ideel styringsmodel. For tværorganisatoriske projekter er en styregruppe tæt på at være uundværlig.

De praktiske vanskeligheder i at få etableret en styregruppe afhænger af relationerne mellem køber og leverandør. Når IT-afdelingen er intern eller kun yder service til en lukket kreds af brugere/købere, er det problemfrit. Nøgleinteresserne er en del af samme organisation og i en eller anden grad knyttet sammen af et fælles ejerforhold.

Når køber og leverandør repræsenterer forskellige virksomheder, kan det være vanskeligt at få etableret en fælles styregruppe. Det er dog almindeligt i forbindelse med store projekter. Konsekvenserne der kan følge af at mangle en styregruppe, gør det naturligt og nødvendigt.

I forbindelse med mindre projekter er det almindeligt at arbejde uden styregruppe. Hvis der er skrevet kontrakt, kan den fra købers synspunkt være styrende; kontrakten skal blot opfyldes. Leverandøren kan oprette en

intern styregruppe, og projektlederen har dermed to fornuftige pejlemærker: kontrakt og styregruppe.

I interne IT-afdelinger arbejdes der i en vis grad både uden kontrakt og uden styregruppe, blot med kontakt til en brugergruppe. Projektlederens liniechef fungerer som organisatorisk forankring for projektet. Brugergruppen er normalt uden formel ledelsesmæssig kompetence til at beslutte forhold der ændrer på projektets leveringsbetingelser og budget. I praksis har de stor indflydelse, og hele modellen tilfører maksimal risiko for projektforløbet.

Den formelle styregruppe repræsenterer en stram model for styring. Brugergruppen som styrende må betegnes som en løs model. Med en løs model kan projektlederen komme i tvivl om hvem der har kompetence til at træffe en given beslutning. Med en stram model kan det være vanskeligt at få truffet beslutninger der ændrer på projektets indhold.

Valget af den rigtige styringsmodel er en væsentlig faktor i bestræbelserne på at øge mængden af succesfulde projekter. Typisk er der behov for strammere modeller for interne projekter, og mere dynamiske modeller i forbindelse med „fast tid og pris“ projekter.

Styregruppens sammensætning

Styregruppen skal sammensættes på en måde der sikrer at projektets behov for beslutninger opfyldes. En typisk sammensætning vil omfatte:

Køberen af projektets resultat

Køberen stiller midler til rådighed og skal med i de beslutningsprocesser, der påvirker økonomi og leveringsbetingelser.

Brugerens af projektets resultat

Brugerens er som repræsentant i styregruppen en fremmende faktor for at beslutningerne kan træffes både hurtigere og med større kvalitet, og virker samtidigt fremmende for en smertefri igangsætning.

Projektlederens liniechef

Liniechefen har en særlig interesse i at beslutningerne der træffes, har sammenhæng med afdelingens øvrige aktiviteter.

Underleverandører

Det er kun afgørende underleverandører, der kan være med i styregruppen. Nogle projekter er imidlertid mere præget af entreprisestyring, og der kan forekomme afgørende leverancer.

Projektlederen

En sammensætning, der ikke omfatter projektlederen, må generelt vurderes som risikabel. Styregruppen vil mangle viden om baggrund og konsekvenser, når der skal løses problemer og træffes beslutninger.

I nogle tilfælde suppleres der med nøgleinteressenter, der har en afgørende rolle i det givne projekt. Det kan være interesseorganisationer, eller der kan være spørgsmål om arkitektur, kvalitetsstyring etc.

Et særligt problem skal løses for de organisationer der bruger en central styregruppe for alle projekter. Det kan være en god løsning når der kun kører 3-4 projekter ad gangen, og medlemmerne ville være de samme i alle styregrupper. Det er også muligt at projekterne netop skal ses og styres i sammenhæng. Men uanset grunden skal der gives mulighed for at projektlederne får lejlighed til at deltage under de punkter på styregruppemødernes dagsorden, der har betydning for deres projekt.

Arbejdet i styregruppen

Styregruppen er først og fremmest problemløser og skal opfattes som sådan, mere end som styrende.

Styregruppen skal altid involveres når der er behov for at ændre ambitionsniveau, tidsplan eller ressourceforbrug.

Styregruppen spiller også en rolle i at formidle konsekvenserne af de ændringer, der opstår i projektets omverden. Herunder ændringer i lovgivning, strategier eller markedsforhold.

Ændringer er uundgåelige. Det er kun størrelse og konsekvens, der varierer imellem projekter. Styregruppen er i reglen involveret i at få gennemført dem uden at ryste på hånden og med en realistisk opfattelse af konsekvenserne for pris og leveringstid.

Styregruppens opgaver

Styregruppen skal registrere og tolke de ændringer i projektets omverden, der har betydning for projektets rammer.

Styregruppen skal løse de af projektets problemer, der ligger uden for projektlederens kompetence.

Ud over disse principielle opgaver er det hensigtsmæssigt, at styregruppen påtager sig følgende konkrete opgaver:

- Vurdering af om projektrapporterne giver anledning til at revurdere sandsynligheden for projektets succes. Hvis fremdriften ikke forløber

som ventet, skal styregruppen vurdere, om rammerne skal ændres. Det kan være i form af ressourcer, eller af at leveringsdatoen skal ændres.

- ▶ Fremskaffelse af de nødvendige ressourcer. Det gælder ikke kun ved projektets start, men også når der træffes beslutning om ændringer. Styregruppen skal enten skaffe de nødvendige personer, maskiner og budgetter eller initiere den nødvendige fremskaffelsesproces.
- ▶ Godkendelse af de strategier projektlederen udarbejder for udvikling, test og implementering. Strategierne er styrende for leveringstakt og -tidspunkt, og godkendelsen ligger derfor naturligt i styregruppen.
- ▶ Godkendelse af forandringsanalysen. Projektlederen sørger for at konsekvenserne af systemets ibrugtagning dokumenteres ved hjælp af en forandringsanalyse. Resultatet forelægges styregruppen, med mulighed for at ændre projektet på en måde, der kan afbøde eller ændre konsekvenserne.
- ▶ Vurdering af om projektlederen lever op til sit ansvar. Det gør projektledere i reglen, men styregruppen skal håndtere undtagelserne.
- ▶ Godkendelse af afslutnings- og evaluéringsrapporten.

Alt i alt må man sige at i et roligt og velledet projekt, har styregruppen ikke mange aktiviteter. Projektet skal initieres formelt, og afslutningen godkendes.

Styregruppens ansvar

Det er styregruppens ansvar at der er sammenhæng mellem projektets ressourcer, indhold og leveringsdatoer. Styregruppen lever op til sit ansvar ved at godkende de nødvendige justeringer af de tre forhold.

Det er klart at styregruppen må støtte sig til projektlederens vurdering af projektets produktivitet og omkostninger. Men det ændrer ikke på placeringen af svaret.

Det er projektlederens opgave at følge op på projektets fremdrift. Eventuelle forskelle mellem status og forventninger skal håndteres af styregruppen. Når disse mekanismer fungerer, er forudsætningerne til stede for at undgå overraskelser i form af projekter, der gennem lang tid rapporterer

en tilfredsstillende status – for så pludselig at ændre det til en meget utilfredsstillende.

Styregruppens kompetence

Styregruppen har kompetence til at ændre projektets rammer i form af ressourcer, tidsplan og indhold. Hvis styregruppen ikke har denne kompetence, har den selvsagt heller ikke ansvar for, at der er sammenhæng mellem disse forhold.

Styregruppen har kompetence til at udskifte projektlederen.

Succesfaktorer

Der er nogle faktorer, som styregruppen skal være opmærksom på er bestemmende for et succesfuldt samarbejde med projektgruppen.

For det første skal styregruppens behov for information fastlægges. Det skal ske på et tidligt tidspunkt i projektforløbet, gerne i forbindelse med projektopstarten. Risikoen ved ikke løbende at have den rette information er at projektgruppen vurderer at der træffes beslutninger på et for spinkelt grundlag. Specielt hvis den samlede information i projektgruppens øjne peger på andre løsninger end styregruppens, kan der i sin yderste konsekvens opstå et modsætningsforhold mellem projekt- og styregruppe.

For det andet skal styregruppen sørge for at behandle problemstillinger på en måde, der set ud fra projektet, er forsvarlig. Styregruppen skal undgå at træffe 'her og nu' beslutninger, også selvom det er ud fra en positiv holdning om ikke at forsinke projektet. Det er en god tommelfingerregel at problemer, der ikke var formuleret på et styregruppemødes begyndelse, og som derfor ikke har været kendt og behandlet af projektgruppen, ikke kan finde en løsning på samme møde.

Endelig skal styregruppen sørge for at uddanne sig, både med hensyn til projektarbejde og til domæneviden. Vel kan man udøve ledelse uden at kende til teknik, men projektledelse er tæt på den virkelige verden. Styregrupsers beslutninger har konsekvens for projektledelsen, og en vis indsigt er ønskelig.

Hvis disse forhold ikke ydes tilstrækkelig opmærksomhed, risikerer styregruppen at blive stemplet som „politisk“. Det skal i den forbindelse opfattes negativt, og som en vurdering af at styregruppen træffer beslutninger, der er ude af takt med virkeligheden eller bygger på skjulte motiver.

Det er sjældent at styregupper har skjulte motiver, og misforståelser kan forhindres ved hjælp af:

- Tilstrækkelig information
- Tilstrækkelig tid
- Uddannelse.

En af årsagerne til manglende opmærksomhed på disse forhold er gruppens navn. Det antyder at dens primære opgave er at træffe beslutninger. Men den er faktisk i højere grad at løse projektets problemer. Styregruppemedlemmers opfattelse af forskellen kan have betydning.

Styregruppens uddannelse

Hvilken uddannelse og træning er relevant for styregruppemedlemmer? De kan deltage på projektopstartseminarer. Det er god træning i projektarbejde og en effektiv introduktion til konkrete projekter.

Dernæst er det særdeles nyttigt at sætte sig ind i projektstrategier for udvikling, test og implementering. Det giver en god basis for at diskutere med kunder og projektleder. Strategier er styrende for udviklingstakt, kvalitet, omkostninger og andre emner, der nok bør have styregruppens opmærksomhed.

Et projektlederkursus er også relevant. Det øger muligheden for at hjelpe projektet.

Endelig bør der arrangeres interne seminarer, for eksempel en gang om året, for nuværende og kommende styregruppemedlemmer. Holdningerne kan afdupses, og retningslinierne for arbejdet kan revideres og udbygges.

Det giver et uddannelsesforløb som indeholder:

- Projektopstartsseminar (1-2 dage pr. gang)
- Strategier, teori (2 dage)
- Projektledelse (5 dage)
- Seminar (1 dag om året).

Uddannelsen er en god investering, der blandt andet kan skabe den nødvendige opmærksomhed på succesfaktorerne.

3.4 Projektlederen

Projektlederen er en nøgleperson i virksomheden. Organisationens nye produkter fremstilles i projekter, og projektlederen har en afgørende indflydelse på virksomhedens kultur og image.

Virksomhedens kultur præges i høj grad af projektlederens daglige ledelsesform. Med hensyn til image er projektlederen for nogle virksomheder særlig vigtig, fordi det er projektlederen, der har den primære kontakt til kunden.

Projektlederens betydning på disse områder er endda blevet større med tiden. Der er fra nogle virksomheders ledelse større opmærksomhed på projektlederjobbets indhold og betydning end på mellemlederrollens. Som en afgaende konsekvens har mellemlederrollen fået nyt indhold og dermed også nye muligheder for at udvikle sig og reformulere sin egen rolle i virksomheden.

Samtidig med at projektlederens kompetence øges, sker der også en udvidelse af arbejdsopgaverne for projektledere af store projekter. Det gælder økonomistyring, budget- og personaleansvar.

Konsekvensen er ikke kun en ændret rollefordeling mellem projekt- og mellemledere, men også en differentiering blandt projektledere. Det er ikke alle projektledere, der har den fulde kompetence.

Om projektlederen

Projektlederen skal være opsøgende i forhold til interessenterne.

Den opsøgende proces skal sikre den nødvendige udveksling af information, afdække interesse modsætninger og identificere uventede hændelser så tidligt som muligt.

Mangel på information er ikke en acceptabel forklaring på et uheldigt projektforløb, da projektlederen aktivt skal sikre, at den nødvendige information går til/fra interessenterne.

Interessekonflikter er normalt forekommende i et projektforløb. Interessenters bestræbelser på at få maksimal nytte af et projekt vil ofte være indbyrdes modarbejdende. Projektlederen skal identificere de naturlige modsætninger og sætte aktiviteter i gang, der løser dem.

Uventede hændelser er ligeledes normalt forekommende. Opsøgende arbejde kan påpege et behov for at ændre planer og dermed begrænse uønskede konsekvenser.

Hvem kan fungere som projektledere?

Man kan have projektledere ansat til jobbet og have en tilsvarende stilningskategori i personalesystemet. Men alle skal være indstillet på at kunne fungere som projektleder. Alle kan „risikere“ at blive trukket ud af deres daglige arbejde for at realisere et produkt eller en forandring. Deres „normale“ job kan evt. passes af deres liniechef.

Denne tendens hen imod en mere fleksibel opfattelse af projektleder-jobbet er udtalt. Også i virksomheder med en fast gruppe af projektledere vil der fra tid til anden være behov for, at andre påtager sig projektledelse.

Rekruttering af en linieleder kan være begrundet i projektets strategiske betydning for virksomheden, eksempelvis hvis eksistensen, eller den fortsatte vækst, er afhængig af projektets resultat.

Ledelsesindholdet

I såvel den interne som den eksterne del af jobbet er ledelsesindholdet fremtrædende og komplekst.

Intern er det i forholdet til projektdeltagerne. Uddannelsesniveauet blandt menige projektmedlemmer vokser til stadighed. Dette niveau skal vedligeholdes og udvikles i lighed med virksomhedens øvrige investeringer. Antallet af specialister øges, uanset at man kunne ønske det anderledes. Aktiviteterne i projektet bliver stadig mere varierede, ikke mindst afleadt af anvendelsen af ny teknologi.

Der påhviler også projektlederen et krav om at **etablere en hensigtsmæssig kultur** i gruppen. Samarbejdsformerne skal fastlægges og accepteres. De enkelte projektdeltageres krav og behov skal tilgodeses i den udstrækning, det ikke hämmer gruppen som helhed. Normerne i gruppen skal fastlægges, også under hensyntagen til normerne der gælder for hele virksomheden.

Alle projektdeltagerne skal tage aktiv del i samarbejdsprocessen. I takt med den øgede delegering der finder sted, hvor medarbejderne i stigende grad løser opgaverne selvstændigt, er det rimeligt at alle i projektgruppen tager ansvar for, at der eksisterer et godt samarbejdsklima.

Projektkulturen er et fælles anliggende, men der påhviler projektledere en særlig forpligtelse til at sætte den proces i gang, der udvikler og vedligeholder kulturen.

Eksternt er ledelsesindholdet også komplekst.

Det eksterne element er projektlederens „politiske“ liv. Udtrykket skal i denne forbindelse opfattes neutralt eller endda positivt. En vellykket politisk indsats kan mindske spændinger og opbløde modsætninger.

Det politiske liv er præget af forholdet til interessenterne. Antallet af dem vokser, og deres magtbaser kan være vanskelige at vurdere arten og størrelsen af.

Nøgleinteressenterne i form af køber, bruger, driftsenhed etc. skal tilgodeses. Disse „klassiske“ interesser er velkendte og har velkendte magtbaser. Men andre grupper af interesser skal ydes lignende opmærksomhed.

Underleverandører spiller en voksende rolle som succesfaktor. De interne underleverandører skal også håndteres kommersielt – uanset om organisationen benytter intern kontering mellem afdelinger. Det er nødvendigt at specificere det aftalte produkt og de dertil knyttede leveringsbetingelser. Det skal ske efter forretningsmæssige principper og kan i en vis udstrækning gøres efter de retningslinier, der i dag benyttes over for eksterne leverandører.

For hver underleverance skal der gennemføres en planlægning, der tager højde for forsinkelser og andre uopfyldte forpligtelser. Den enkelte underleverandørs kritiske betydning skal vurderes med henblik på at tage præventive forholdsregler.

Lovgivningsmagten er også en del af det politiske liv. Det gælder i form af love og regler, der skal overholdes, og som det kan være vanskeligt at få overblik over.

Endelig skal **interesseorganisationer**, herunder faglige organisationer, håndteres politisk. En interesseorganisations magtbase kan variere fra projekt til projekt, og typen og mængden af samarbejde skal indrettes tilsvarende.

Faktorer der letter projektledelsen

Kravene til projektlederen er mange og mangeartede. Når alle kravene summeres op og stilles i forhold til en enkelt projektleder, kan de næsten virke uoverkommelige.

Kravet om de mange forskellige lederegenskaber man skal besidde, og kravet om de forskellige ledelsesformer, der skal anvendes over for forskellige grupper af interesser, kan i sig selv blive en belastning.

Heldigvis er virkeligheden ikke således indrettet at alle projektledere skal betjene sig af alle ledelsesformer i alle projekter. Der er i virkeligheden adskillige forhold der hjælper med til, at det er muligt at få succes.

For det første kastes projektlederen ikke ud i alle problemer samtidigt.

For det andet er en given projektleder fra naturens hånd udstyret med

en række hensigtsmæssige egenskaber. Det vil være forskelligt fra projektleder til projektleder, hvilke opgaver de har let ved at løse fra starten af karrieren. Nogle har et naturligt talent for forhandling, og andre har aldrig haft problemer med personaleledelse. Talentet kan være medfødt, eller det kan være et resultat af opdragelse, skolegang og sociale sammenhænge, og ikke mindst et resultat af personlige erfaringer.

En tredje positivt hjælpende faktor er projektgruppen. Det hverken kan eller skal være meningen, at projektlederen alene bærer ansvaret for ledelsesdimensionen. I en moderne projektorganisation vil både projektgruppen og mange af interessenterne opfatte sig som medansvarlige for at opnå succes. De vil finde det naturligt at medvirke aktivt til succesen ved at være indstillet på mere end blot at løse egne opgaver tilfredsstillende. Det er dog rimeligt at pålægge projektlederen et særligt ansvar med hensyn til at sætte de processer i gang, der skaber den gode ledelse.

For det fjerde har andre projektledere prøvet noget lignende før og kan give råd og vejledning. Denne mulighed kan bruges oftere end det ses i praksis.

Endelig er der en femte væsentlig faktor, nemlig værktøjssiden. Alle ledelses- og styringsaktiviteter understøttes af værktøjer.

Anvendelse af værktøjer

Værktøjerne er i særlig grad en understøttende faktor. De kan bruges i forskellige sammenhænge, bestemt af om der er behov for analyse af en problemstilling, beskrivelse af en problemstilling, eller om behovet er en eller anden form for kommunikation.

Mange af værktøjerne kan bruges i alle tre sammenhænge. Interessentanalysen er et sådant eksempel. Analysen af projektets omverden er i sig selv en værdifuld anvendelse, og den dertil knyttede interessentmodel sikrer at analysens resultat fastholdes og dokumenteres. Uden dokumentation ville det være vanskeligere at bruge analysens resultat som grundlag for andre værktøjer. For interessentmodellens vedkommende er det informationsanalyse og kvalitetsstyring, der anvender den færdige model som grundlag. Interessentmodellen kan desuden bruges i forbindelse med kommunikation til projektets omverden. Det er en forudsætning for at interessenter, der ikke deltager direkte i udviklingsarbejdet, bliver opmærksomme på projektets eksistens og egenart. Der kan være behov for at forberede og gennemføre aktiviteter inden for deres eget styrings- eller fagområde.

Et så grundlæggende værktøj som projektmodellen kan også bruges i

de tre nævnte sammenhænge: støtte til analyse og udvikling, dokumentation af de opnåede resultater og mulighed for at kommunikere resultaterne til omverdenen. I den analytiske anvendelse fungerer værktøjet også som checkliste. Projektet bliver mindet om at gennemføre alle de nødvendige aktiviteter.

Der er værktøjsstøtte til projektlederen på alle arbejdsmråderne. Det IT-faglige er rigeligt dækket, og med hensyn til de rene ledelsesaktiviteter er der tale om at skulle vælge mellem mange muligheder. Udbuddet af modeller, der understøtter personaleledelse, personaleudvikling, forhandling, kulturanalyse, risikoanalyse, målformulering og projektets stærke/svage sider, er righoldt.

Men det er mennesker der i sidste ende er afgørende for resultatet. Et værktøj er ikke nyttigt før et menneske bruger det. Intet værktøj repræsenterer i sig selv en sandhed eller producerer automatisk en løsning. Kvaliteten af at bruge et bestemt værktøj afhænger også af kvaliteten af anvendelsen.

Alle de mange forskellige ledelsesmodeller, principper og værktøjer kan betragtes som projektører, der hver især bidrager til at belyse et stykke af scenen. Hver projektør har sit eget udgangspunkt og kan derfor producere en unik effekt.

På et tidspunkt vil introduktion af nye projektører være overflødig. Enten fordi effekten ikke kan spores, eller fordi den yderligere del der oplyses, ikke er betydningsfuld for den samlede opfattelse.

Lederegenskaber

Sammensætningen af lederegenskaber hos en projektleder vil præge den enkeltes lederstil i et omfang, der reelt udgør en klassificering. Denne klassificering udtrykker ikke en kvalitetsforskælf. Den enkelte type leder kan fungere mere eller mindre succesfuldt.

Den enkeltes personlige opfattelse af god ledelse, kombineret med de personlige lederfærdigheder og hidtidige erfaringer, vil give lederen svage og/eller stærke sider i sin egenskab af:

- Stifinder
- Iværksætter
- Kulturskaber
- Psykolog.

Stifinderen kan lede projektet forbi forhindringer. Er der en vej frem, bliver den brugt til at give projektet fremdrift. Stifinderen ændrer holdningen hos en interesseret, finder det eneste mulige kompromis, lægger en strategi der udnytter alle mulighederne i den pågældende organisation såvel som i omverdenen. Den velfungerende stifinder har planlagt projekts vej så god tid i forvejen, at den øvrige projektorganisation ikke registrerer at en blindgyde er undgået, eller at en vanskelig passage er foretret.

Iværksætteren sørger for projektets fremdrift ved at skabe et dynamisk og iderigt projektforløb. Iværksætteren formulerer nye ideer og inspirerer andre til at gøre det samme. Ideerne tolkes om nødvendigt og omsættes til konkrete arbejdsopgaver. Ofte handler det om at udnytte eksisterende strømninger til fordel for projektet. Iværksætteren inspirerer og begejstrer sin omverden ved at påpege mulighederne.

Kulturskaberen er projektlederen der skaber resultater gennem en stærk kultur. Der opstilles fælles klare mål, som alle slutter op omkring. Projektorganisationen oplever at der er sammenhæng i hverdagen, og at projektets samlede aktiviteter giver mening. Kulturskaberen formår at frigøre ekstra reserver hos andre, uden at det opfattes som en belastning, men derimod som aktiviteter, der giver mening og sammenhæng.

Psykologen bruger sine evner til at opfange signaler i omverdenen og fra enkeltpersoner. Ved at aflæse og tolke mekanismer og reaktioner i organisationen er det muligt at forudsige resultatet af en beslutningsproces. Det er ikke mindst evnen til at forstå baggrunde og motiver, der giver de ledelsesmæssige fordele. Psykologen arbejder også med enkeltpersoner og magter at kommunikere meningsfuldt med mange forskellige mennesker. Resultatet er et roligt projektforløb uden misforståelser på grund af fejl-kommunikation.

En projektleder vil sjældent være lige velfungerende i alle roller. Det behøver ikke i sig selv at give problemer fordi projektets fremdrift kan sikres på mange måder. Der kan dog opstå situationer, hvor en projektleder med en udpræget profil i en enkelt af kategorierne må søge hjælp hos andre for at nå et ønsket resultat. Et projekt, der af en eller anden grund virker ide-forladt og uinspireret, har ikke behov for at en „psykolog“ skaber ro om-

kring dette forhold. I den situation må andre involveres for at skabe nye ideer. Den ideelle projektleder behersker færdigheder inden for alle kategorier og kan betjene sig af dem frit og situationsbestemt. Men i erkendelse af at projektledere også er mennesker og sjældent guder, vil der derfor være behov for at gøre brug af andres egenskaber. Projektdeltagerne kan inddrages i ledelsesprocessen som et middel til at øge mængden af lederprofiler.

En projektgruppes samlede lederpotentiale er større end summen af de enkelte medlemmers, fordi samspillet mellem forskellige egenskaber giver ekstra muligheder.

Projektlederens personlige egenskaber

Som supplement til de lederegenskaber der udtrykkes ved hjælp af begreber fra ledelsesteorier, kan man også tale om mere direkte og almindelige menneskelige egenskaber.

Hvilke karakteristika har en god projektleder? Det er et spørgsmål, der er vanskeligt at besvare, fordi virksomheder, projektgrupper og opgaver er forskellige. En specifik situation stiller krav om specifikke egenskaber. Men der er nogle hensigtsmæssige karaktertræk, om hvilke man kan sige at tilstede værelsen af et stort antal af dem hos en projektleder vil virke fremmende for projektet:

- ▶ Evne til at kommunikere med forskellige mennesker, både socialt og fagligt. Det gælder i samspillet med mennesker i den direkte projektorganisation og over for kunder, leverandører og liniechefer.
- ▶ Fornemmelse for at levere situationsbestemte resultater. Timing er vigtigere end en personlig evne til at arbejde mange timer i døgnet.
- ▶ Tro på sig selv som menneske, gerne suppleret med evne til at formidle selvsikkerheden ligefremt og ukompliceret.
- ▶ Evne til at skabe tiltro til planer og strategier. Forudsætningen er selvfølgelig at indholdet af dem berettiger til det.
- ▶ Evne til at komme videre fra fastlåste stillinger, med mange vindere og få eller ingen tabere.

- ▶ Talent for problemløsning, gerne støttet af evne til at formulere et kompromis. Det er menneskeligt at vælge side og tage parti i diskussioner og beslutningsforløb, men det er hensigtsmæssigt for projektledere at holde sig fri af ydersynspunkterne.
- ▶ Evne til både at lede en proces og lede enkeltpersoner. Det er ikke nogen enkel opgave både at sikre det rigtige resultat og samtidig være opmærksom på enkeltpersoner.
- ▶ Evne til at øge produktiviteten i et projekt uden at sænke motivationen. Det kræver i særlig grad karaktertræk som åbenhed, ærlighed og en vis portion frygtløshed.
- ▶ Evne til at håndtere katastrofer roligt og behersket. Det gælder også for almindelige problemer og kriser. Det er uhensigtsmæssigt at tilføje en urolig problemløsningsproces til en i forvejen problemfyldt situation.
- ▶ Evne til at formulere sin ledelsesfilosofi uden at rødme over at bruge ordet filosofi.
- ▶ Evne til at formulere acceptable mål både for projektet og for enkeltpersoner. Målet for den enkelte skal udvikle uden at stresse.
- ▶ Evne til at kritisere andres resultater på en måde der af modtageren opfattes som en vurdering af resultatet, og ikke af personen der fremstillede det.
- ▶ Evne til at acceptere at mennesker begår fejl. Selvfølgelig skal alle bestræbe sig på at arbejde fejlfrit, men fejl skal accepteres som et naturligt resultat af udviklingsarbejde.
- ▶ Åbenhed over for pragmatiske løsninger når situationen kræver det. Dogmer og principper skal kunne fraviges, også således at fravigelsen er velovervejet og synlig i den konkrete situation.

Rekruttering af projektledere

Projektledere rekrutteres generelt fra tre forskellige grupper:

nøglepersoner i organisationen. Og dermed en knap ressource. Det er en naturlig udvikling for en virksomhed at uddanne sine medarbejdere i takt med at en bestemt viden skal bruges i flertallet af projekter. Når en nøgleperson efterspørges stærkt, kan det være en indikation på et uopfyldt efteruddannelsesbehov. Men tilbage står at der altid vil eksistere nøglepersoner i et vist omfang, og at projektlederens afhængighed af dette kun kan mindskes ved i god tid at få accept på personernes medvirkken.

3.7 Projektgruppen

Projektdeltagerne er på flere måder projektlederens vigtigste ressource. Først og fremmest er de den væsentligste ressource for at nå projektets mål, men det er også den dyreste af de ressourcer projektlederen bliver betroet at forvalte.

Projektdeltagerne repræsenterer en investering for virksomheden. Genanskaffelsessummen i form af uddannelse, indføring i virksomheden og indføring i opgaven er høj.

Det er projektlederens direkte ansvar at passe godt på de betroede talenter.

Selv i tilfælde hvor projektdeltagerne refererer til en afdeling i linieorganisationen med hensyn til det formelle personaleansvar, påhviler der projektlederen en række opgaver, der udspringer af vedkommendes funktion som daglig leder af mennesker.

Projektlederen er ofte fritaget for formelt personaleansvar for arbejdsforhold, arbejdsmiljø, løn, ansættelse, fyring etc. Men tilbage står den daglige ledelse, og ikke mindst ansvaret for anvendelsen af projektdeltagernes tid.

Projektledelse er ofte en persons debut som personaleleder, og projektlederen oplever måske for første gang:

„At skabe resultater gennem andre.“

Man oplever måske en situation, hvor den opgave, der nu skal løses af andre, er en man selv kan løse, og måske har løst tilfredsstillende et antal gange før. Nu skal opgaven kun formuleres, fordi den gennemføres af andre. Vurderingen af resultatet indgår i en samlet vurdering af projektgruppen, og vil dermed uundgåeligt også indgå i en samlet vurdering af projektlederen.

En personaleleder skal tænke i „mennesker“ såvel som i „opgaver“. Mennesker skal motiveres for at yde deres bedste, og de har krav på en personlig og faglig udvikling. Det stiller krav til projektlederen om at være opmærksom på de faktorer, der motiverer den enkelte projektdeltager. Desuden skal projektlederen være opmærksom på betydningen af sin egen ledertil. Det er en faktor der påvirker kvaliteten af personaleledelse i væsentlig grad. Generelt kan man sige at god personaleledelse er ligeligt betinget af lederens kendskab til medarbejderne og til sig selv.

Det er karakteristisk for personaleledelse at omverdenen i høj grad er opmærksom på, når den udføres med for ringe kvalitet. Der er langt mindre opmærksomhed på god personaleledelse. Omverdenen, og projektgruppen selv, vil hurtigt blive opmærksomme på problemer, konflikter og ineffektivitet. Simpelthen fordi konsekvensen opleves i hverdagen. Det er naturligt, at omverdenen ikke i samme grad er opmærksom på de tilfælde, hvor der ikke eksisterer samarbejdsproblemer, hvor konflikterne undgås fordi de løses før de kommer til udtryk, og hvor alle opgaver løses motiveret og effektivt.

Dette forhold er ikke til at ændre og må blot accepteres. Personaleledelse skal under alle omstændigheder udøves bedst muligt. Projektlederens egen motivation opstår i sidste ende når projektresultatet er godt, dvs. når projektet kan sluttet af med en velmotiveret og samarbejdende projektgruppe.

Motivation og udvikling

Motivation og faglig udvikling skal tilgodeses samtidig. De to begreber skal fungere i et indbyrdes afstemt forhold.

Ved hjælp af motivation får medarbejderen lyst til at gennemføre de nuværende opgaver og påtage sig de kommende opgaver med lyst og energi.

Den faglige udvikling skal tilgodeses, ved at medarbejderen får nye opgaver af en mængde og et indhold, der giver en passende udviklingstakt og -retning.

En medarbejder der motiveres til at gøre en stor indsats, men som ikke får mulighed for nye udfordringer, vil blive frustreret og utilfreds. Motivation skal følges op af nye opgaver eller større selvstændighed.

En medarbejder der stilles over for nye opgaver eller gives større selvstændighed, men uden en samtidig motivering, oplever blot et øget arbejdsspres. Nogle positivt mente handlinger kan således blive opfattet negativt.

Den korrekte procedure for projektlederen er at identificere den enkel-

te medarbejders motivationsfaktorer og muligheder for faglig udvikling. Forudsat at projektlederen både kan motivere og har mulighed for at iværksætte en faglig udvikling, er grundlaget skabt for personaleudvikling.

De områder i en medarbejders stilling som projektlederen bør vurdere i udviklingsprocessen, er følgende fem nøgleområder:

1. Variation i færdigheder

Gives medarbejderen mulighed for at bruge varierende færdigheder? Udnyttes medarbejderens færdigheder fuldt ud, eller er der uudnyttede muligheder inden for det nuværende jobs rammer?

2. Arbejdets identitet

Kan medarbejderen identificere resultatet af sit arbejde? Udgør arbejdet for eksempel et samlet hele, der giver medarbejderen mulighed for at følge et job fra start til slut? Rent samlebåndsarbejde savner identitet.

3. Arbejdets betydning

Har arbejdet en betydning, der opleves som vigtig for andre i virksomheden eller i virksomhedens omverden? Projektlederen skal prøve at give alle del i det „vigtige“ arbejde.

4 . Selvstændighed

Gives medarbejderen mulighed for at tilrettelægge og gennemføre sine arbejdsopgaver selvstændigt? Det handler om indflydelse på egen arbejdssituation.

5. Målelig effektivitet

Er arbejdet af en karakter, der sætter medarbejderen i stand til at måle sin egen effektivitet? Har medarbejderen mulighed for at aflæse effektiviteten direkte som følge af arbejdsprocessen, eller gives der mulighed for feedback fra interesserne?

Personaleledelse på kort og på lang sigt

Mål for personaleledelse kan deles op i kortsigtede og langsigtede mål.

De kortsigtede er knyttet til den opgave som projektet løser her og nu. De langsigtede rækker ud over projektets tidshorisont og er knyttet til generelle mål og forhold i virksomheden.

En af projektformens ulemper er, at det er vanskeligere at drive effektiv

langsigtet personaleudvikling. Informationen, der er grundlaget for langsigtet planlægning, opstår i projektet. Projektlederen er i besiddelse af informationen, men har måske hverken uddannelse eller kompetence til at opstille langsigtede mål for personaleudvikling og til at planlægge et udannelsesforløb.

Det er personalefunktionens ansvar – eller linieledelsens, i mangel på en personalefunktion – at langsigtede mål defineres og nås, ligesom det er projektlederens ansvar at opstille og nå de kortsigtede mål.

Formålet med de kortsigtede mål er at gennemføre projektet med succes, herunder at have udnyttet medarbejdernes evner optimalt. De kortsigtede mål defineres på grundlag af opgavens karakter, og motivation på kortsigt er at få anerkendelse for arbejdsopgaverne udførelse. Forudsætningerne for succesrig kortsigtet personaleledelse er først og fremmest at projektlederen afsætter den nødvendige tid til det og har rådighed over de lige så nødvendige motivationsfaktorer, det være sig i form af løn, tilladelse til fleksibel arbejdstid eller anerkendelse af forskellig art.

De langsigtede mål for personaleledelse er en del af virksomhedens strategiske planlægning. De skal sikre at virksomhedens samlede sæt af viden, erfaringer, færdigheder og holdninger ændres i en sådan retning og med en sådan hastighed, at virksomhedens samlede strategiske planlægning kan realiseres.

Hvorfor skal en projektleder interessere sig for strategisk personaleudvikling? De langsigtede mål er jo netop defineret som mål, der rækker videre end til at nå projektets primære mål. Svaret er for det første at opstilling af langsigtede mål for den enkelte medarbejder, med en deraf følgende personlig udvikling, vil have en positivt afsmittende virkning i det daglige projektarbejde. For det andet at projektlederen, set som en del af virksomhedens ledelse, bør medvirke til at understøtte virksomhedens samlede personaleudvikling.

Forudsat at arbejdsdelingen respekteres og forudsætningerne opfyldes, kan personaleledelse tilgodeses alene ved en effektiv kommunikation mellem projektledere og den ansvarlige personalefunktion.

En indfaldsvinkel til at fastlægge, hvem der skal have ansvaret for den langsigtede personaleudvikling, er at fastlægge, hvem der gennemfører personaleudviklingssamtalen. Er det projektlederen, afdelingslederen eller personaleudviklingschefen, der gennemfører samtalen i praksis?

Projektdeltagernes færdigheder

Der er en klar udvikling i retning af større faglig kompetence hos den enkelte projektdeltager.

Konstruktører udfører mange aktiviteter i tillæg til dem, der naturligt følger af stillingsbetegnelsen. Det er ikke tilstrækkeligt at kunne programmere. Man skal beherske udviklings- og testværktøjer. Der er selvstændige dokumentationsværktøjer eller integrerede udviklingsværktøjer. Prototyping kan være et omfangsrigt arbejdsområde med selvstændige teknikker, værktøjer og samarbejdsformer. Der kan være brug for objektorienteret design og programmering.

Planlæggere skal kunne håndtere de forskellige værktøjer til virksomheds- og systemanalyse, og der kan være brug for at beherske „Information Engineering“ og/eller objektorienteret analyse.

Konsekvensen af den øgede kompetence er, at projektlederen oplever at projektet består af specialister.

Virksomheder har normalt flere niveauer af egentlige specialister, og hvis alle programmører eller alle planlæggere har nogenlunde samme færdigheder, er de ikke specialister i virksomhedens øjne. Men projektlederen oplever dem som specialister fordi:

- ▶ De planlægger og gennemfører selv deres faglige uddannelse. Medarbejderne identificerer og opfylder selv behovet.
- ▶ De nedbryder selv hovedaktiviteter til kalendersatte detailaktiviteter.
- ▶ Arbejdstilrettelæggelsen: Medarbejderne koordinerer selv med andre projektdeltagere, med andre afdelinger og med specialistfunktionerne i virksomheden.
- ▶ Værktøjsvalget: Udviklerne vælger selv værktøj, når der er flere lige gode muligheder.

Denne tendens i retning af større faglig selvstændighed i projektgruppen har nogle positive konsekvenser. Projektlederen får bedre mulighed for at koncentrere sig om ledelses- og styringsopgaver. Omvendt vokser behovet for fælles klare mål.

Projektkulturen

Kulturen i projektgruppen opstår som en syntese af alle de holdninger og regler, som projektdeltagerne demonstrerer og udtrykker.

Projektdeltagerne kan vælge at være tilfredse med den kultur, der opstår

af sig selv. Det kan enten betyde, at projektdeltagernes samlede sæt af holdninger ikke er indbyrdes modstridende, eller at kulturen er tolerant nok til at spænde over forskellene. Det er i sig selv et stærkt kulturtræk at spænde vidt.

Hvis projektdeltagerne derimod ikke er tilfredse med den rådende kultur, må den ændres eller tilpasses. Forudsætningerne for at ændre kulturen er at ændre de bagvedliggende holdninger og værdinormer. Det kan være vanskeligt at påvirke kulturen fordi det altid vil være enkelte menneskers personlige holdninger og meninger om, hvad der er bedst, der skal tilpasses.

Projektets kultur er nuanceret og omfattende, men det er dog muligt at beskrive generelle træk i et projekts kultur. Eksempler på sådanne træk er:

- ▶ Tolerance over for de forskellige faglige orienteringer. Udviklere accepterer brugerrepræsentanternes holdninger og værdisæt, og vice versa.
- ▶ Accept af at der begås fejl. At der skal bruges tid på at finde fejlene, men at selve det forhold, at mennesker begår fejl, accepteres.
- ▶ Åbenhed over for vurderinger af resultater fordi alle ved at kvalitetsvurderinger går på produktet og ikke på personen der fremstillede det.
- ▶ Aktiv løsning af konflikter ved at søge kompromiser der er tilfredsstillende for alle parter. Konflikter kommunikeres og behandles åbent og ligefremt.
- ▶ Trufne beslutninger bakes op af hele gruppen i holdning og handling.

Der kan formuleres lignende generelle træk i forbindelse med møder, koordinering, dokumentation og ledelse.

Eksemplerne er formuleret positivt, men kan også formuleres i negative vendinger. En beskrivelse af kulturen vil i praksis være nuanceret, i og med at den beskriver kendsgerninger. I modsætning til hensigtserklæringer.

Holdninger er ofte filosofiske i deres grundudtryk. Nedenstående filosofiske holdninger vil oftest være enkeltpersoners holdning og kun i sjældne tilfælde udtryk for en generel holdning. Hvis sådanne er fælles holdninger, vil man til gengæld opleve spændende og markante projektforløb:

„Erfaring er billet til det tog, der kørte i går.“

(Nytænkning og det ikke opdagede er i højsædet).

„Rigtige programmører sover aldrig; de besvimer en gang imellem hen over skærmen.“

(Arbejdstid sættes over fritid).

„Livet er at gå på line, resten er bare ventetid.“
(Risikovillighed).

„Når alle er enige, har middelmådigheden seized.“
(Uenighed vurderes positivt).

„Det er nemmere at bede om tilgivelse end tilladelse.“
(Tro på egne vurderinger).

„Initiativ er lig indflydelse.“
(Handlings- og beslutningsvillighed).

„Det er svært at være ydmyg, når man er de bedste.“
(Positiv selvopfattelse).

Kulturændringer skal gennemføres med stor omhu og med skridt, der enkeltvis kan overskues og kontrolleres af alle projektdeltagere. Proceduren kan have disse trin:

- Opnåelse af enighed i projektgruppen om at kulturen skal ændres i en eller anden form og grad.
- Afklaring af den eksisterende kultur. Der skal nås en fælles opfattelse, eller som minimum skal alle gøres bekendt med de øvriges opfattelse.
- Beskrivelse af den ønskede kultur. En kultur i sig selv er ikke så ligetil at beskrive, men reglerne, holdningerne og mulighederne, der ønskes fremmet, kan beskrives.
- Fastlæggelse af eventuelle aktiviteter der skal gennemføres for at nå en ønsket tilstand.

Denne procedure kan begynde med et ønske om mere samarbejde, øget kvalitetsbevidsthed, fælles mål etc. Og som et naturligt resultat ende med enighed om at øge mængden af reviews og inspektioner. Det hele går nemmere, når det er et led i kulturudviklingen. I modsætning til ordrer eller topstyrede opstramninger.

Kulturen skal vurderes, udvikles og vedligeholdes løbende. Den proces, der ligger bag fremstillingen af produktet, er væsentlig både for at nå et godt fagligt resultat og også fordi mennesker har krav på at se sammenhænge i hverdagen og have indflydelse på den.

Projektets image

Ved et projekts image forstås den opfattelse som omverdenen har af projektet. Image eksisterer altså, uanset om man ønsker det eller ej. Det er vigtigt at være opmærksom på, at det er omverdenens faktiske opfattelse, der er gældende. Omverdenen skal påvirkes hvis man af en eller anden grund ønsker at ændre image.

Hvorfor bør man interessere sig for projektets image? Hvad kan image bruges til? Svaret er, at et godt image kan bruges til at fremme samarbejdet med interesserterne, både i og uden for virksomheden. Et dårligt image vil tilsvarende påvirke samarbejdet i negativ retning og mindske de samlede muligheder for projektets succes.

Projekter opfattes både i generelle unuancerede vendinger og i indirekte, men ofte mere præcise karaktertræk.

Eksempler på generelle unuancerede opfattelser er at omverdenen oplever projektet som aggressivt - åbent - omhyggeligt - ubeslutsomt - ufleksibelt - konservativt - teknikorienteret - handlingsorienteret - velfungerende - kvalitetsbevidst.

Projekter vil dog ofte blive opfattet med indirekte imagetræk:

- „De bruger al deres tid i mødelokaler“
- „De er opmærksomme på kundens behov“
- „De er temmelig sociale“.

Pointen er at man selv kan påvirke image ved at lægge vægt på udvalgte aktiviteter. Såfremt man ønsker at blive opfattet som værende meget opmærksom på kundens behov, kan dette relativt nemt opnås. Dels ved at være det, og dels ved at fremhæve at man er det. Og for himlens skyld ikke kun det sidste. Så får man et helt andet image.

Proceduren, der bruges til at ændre image, er magen til den procedure, der bruges til at ændre kulturen. Der er blot den yderligere komplikation at analysen af det eksisterende image forudsætter at interesserterne involveres. Enten ved at interesserterne interviewes direkte eller ved at analysere interesserternes opfattelse indirekte.

Ansvar, opgaver og kompetence

Teoretisk er der ikke behov for ledelsesmæssige retningslinier for projekt-deltagerne. „Menige“ medarbejdere skal principielt blot udføre opgaverne ifølge de givne ordrer. Retten til at lede og fordele arbejdet ligger hos pro-

jektlederen. Men det holder ikke i praksis. Alene det forhold, at medarbejderne i stigende grad fungerer som specialister, er tilstrækkelig årsag. En fagligt kompetent specialist tilrettelægger selv sit arbejde – og det i et omfang, der gør eksistensen af ledelsesmæssige retningslinier nødvendige.

Desuden er det også et spørgsmål om produktivitet. Optimal udnyttelse af en medarbejder forudsætter delegering af ansvar og kompetence. Hvis medarbejderne ikke har mulighed for selv at tage stilling og træffe beslutninger, vil behovet for kommunikation mellem medarbejder og projektleder blive af betydelige dimensioner, og produktiviteten vil falde.

Ansvar

Medarbejderen har ansvaret for at aktiviteterne på projektplanen gennemføres. Det skal ske med den forventede kvalitet og inden for de aftalte tidsmæssige rammer. Medarbejderen er ansvarlig for at rapportere enhver forventet eller realiseret afvigelse. Ansvaret omfatter også at eventuelle konsekvenser for andre medarbejdernes aktiviteter bliver kommunikeret for at give andre medarbejdere mulighed for at leve op til deres ansvar. Enhver medarbejder er ansvarlig over for sin viden.

Opgaver

Medarbejderens hovedopgaver er:

- ▶ Nedbrydning af hovedaktiviteter til detailaktiviteter
- ▶ Estimering af detailaktiviteter
- ▶ Fremstilling af testgrundlag til egen test af komponenter
- ▶ Gennemførelse af detailaktiviteter
- ▶ Egen test af komponenter
- ▶ Rapportering af færdiggørelse.

Kompetence

Kompetencen skal understøtte ansvaret. Medarbejderen har fuld kompetence inden for aktivitetens indhold. Medarbejderen har kompetence til at tilrettelægge arbejdet, koordinere med andre, fastlægge eventuelt uddannelsesbehov og fastlægge behov for afprøvning og kvalitetsvurderinger.

Sidstnævnte er ofte et problematisk forhold. Men konsekvensen af at uddeleger en aktivitet er, at færdigmeldingen af arbejdet ligger hos den udførende. Inklusive at produktet er godkendt til aflevering videre i forløbet. Projektledere fristes lejlighedsvis, måske på grund af pres udefra, til

at erklære en aktivitet for slut. På trods af en medarbejdernes skøn. Det vil altid skabe konflikter og i sidste ende et utilfredsstillende fagligt resultat. Den korrekte løsning for et projekt i tidsnød, der presses til at holde de aftalte leveringsterminer, er at beskære indholdet i produktet, at reducere det funktionelle indhold og definere en senere levering med de resterende funktioner. En anden mulighed er at tage forbehold for kvaliteten af visse navngivne funktioner, således at den endelige klarmelding stadig ligger hos medarbejderen. Man ser dog også det modsatte: at deltagere afleverer et produkt for tidligt og med for ringe kvalitet. Men de skal fanges af de efterfølgende formelle test og afprøvninger.

„Common errors and pitfalls“

Almindelige fejl og fælder. Udtrykket stammer fra manualer. Man kan lejlighedsvis støde på et kapitel med dette navn. Kapitlet indeholder en liste over de mest almindelige fejl og fælder. I forbindelse med personaleledelse er følgende fejl og fælder de oftest forekommende:

- ▶ Kun at tænke i opgaver og ikke at supplere med at tænke i mennesker. Før man bliver leder, tænker man kun i opgaver, men som leder er det nødvendigt også at tænke i mennesker: „Hvad skal der gøres for at udnytte NN's evner optimalt?“ Det er en forudsætning for at skabe resultater igennem andre.
- ▶ At glemme etablering af fælles klare mål.
- ▶ At tage parti i forholdet mellem medarbejdere og virksomhedens ledelse. Udtalelser til ledelsen som: „I ved jo hvordan medarbejdere er,“ eller omvendt til medarbejdere: „I ved jo hvordan ledelsen er“, er helt overflødige. Projektlederen må erkende sit ledelsesansvar over for sig selv og acceptere den grad af organisatorisk ensomhed, der måtte følge med. At beskrive sig som „en lus mellem to negle“ er blot et udtryk for manglende evner.
- ▶ At give større opgaver eller øget ansvar uden at motivere tilsvarende.
- ▶ Kun at anvende en bestemt lederstil. Opgaver er forskellige, mennesker er forskellige, situationer er forskellige. Et projektforløb vil normalt kræve situationsbestemt ledelse.
- ▶ At den langsigtede personaleledelse forsømmes.
- ▶ At projektgruppen ikke involveres i pleje af kultur og image.
- ▶ At ansvar og kompetence ikke delegeres samtidig med opgaverne.

Projektledere skal være opmærksomme på de to helt forskellige behov for at fungere som enkeltpersoner og som hørende til en gruppe. Mennesker har behov for at fungere som personer sammen med andre personer. Det kan tilgodeses ved hjælp af samarbejde, det kan trænes med teambuilding, og projektlederen kan „coache“. Men der er også behov for at se et klart formål med gruppen. De fleste projekter er født med en berettigelse i form af opgaven, der skal løses. Det giver ofte den nødvendige identitet. Men hvis det er uklart hvorfor projektet kører, hvad der er den vigtige årsag til, at denne gruppe mennesker skal arbejde sammen, er det vigtigt for projektlederen at forklare det. Hvorfor er vi her, hvad er meningen? Og så videre. Det lyder meget filosofisk, men det kan i praksis betyde meget for gruppens produktivitet, at der er klare svar og indlysende sammenhænge.

RETNINGSLINIER FOR PROJEKTLEDELSE

4

- 4.1 Indledning
- 4.2 Formål
- 4.3 Indhold
- 4.4 Forudsætninger
- 4.5 Eksempel på beskrivelse
- 4.6 Kommentarer til eksemplet
- 4.7 Efterskrift vedrørende kompetence

Formål

Hvad er formålet? Er det orienterende, vejledende, instruerende, motiverende, til beslutning/godkendelse osv.?

Ansvarlig

Her dokumenteres hvem der er ansvarlig for at formidle informationen.

Hvordan?

Her dokumenteres hvordan formidlingen sker, fx interne meddelelser, interne publikationer, nyhedsbreve, møder, seminarer, email eller rapporter. Man skal også gøre sig nogle overvejelser om det er 'push' eller 'pull'. Altså om informationen skal sendes, eller interessenken selv skal hente den fra projekthåndbogen. Hvis man ønsker en behandling, og en reaktion, skal informationen sendes, eller der skal som minimum sendes en mail om at den er tilgængelig.

Hvornår?

Her dokumenteres hvornår informationen skal finde sted. „Timingen“ er vigtig for informationens effekt.

Eksempel

Eksemplet i figur 5.2 er en traditionel måde at dokumentere analysen på. Men netop informationsanalysen kan inspirere til at anvende visuelle og kreative former for dokumentation. Forløbet i informationsformidlingen, inklusive godkendelser og tilbagemeldinger, kan illustreres. Modellen kan

HVEM	HVAD	TYPE	HVORNÅR
Køber	Fremdrift Problemer Forbrug Funktionalitet	Til godkendelse	Ugentlig
Ledelse (internt)	Kvalitet Fremdrift Økonomi	Til godkendelse	14. dag
Marketing	Funktionalitet	Til orientering	Månedligt
QA	Kvalitet	Til godkendelse	Ved milepæle

Figur 5.2: Informationsmodel.

udbygges med symboler for informationsstrømme, organisatoriske enheder, beslutningsprocesser og tidsmæssige forskydninger, afhængigt af om det er nødvendigt at stille projektet i bero, eller om det er forsvarligt at opere med parallelitet i beslutningsprocesserne.

Forudsat at de visuelle hjælpermidler ikke tager overhånd på bekostning af det konkrete indhold i modellen, kan en kreativ model forenkle og samtidig nuancere og uddybe det væsentlige.

5.4 Risikoanalyse

Formål

Formålet med risikoanalyse er at afdække de risici som – hvis de indtræffer – kan have negativ virkning på projektet. Afdækningen giver mulighed for at overveje alternative løsninger og supplere planen med aktiviteter til at imødegå risiciene.

Ved at opfylde formålet yder risikoanalyse desuden et væsentligt bidrag til kvalitetsstyring: „at kunne forudsige forløbet og resultatet“.

Baggrund

Projektplanlægning er neutral, idet man gennem planlægningsforløbet har forsøgt at være så objektiv som muligt – uden at tage højde for alle tænkelige uheld. Men det er i sig selv et udtryk for en ideel verden.

Dokumentation

Risikoanalysen skal gennemføres og dokumenteres som et obligatorisk produkt. Som en del af fremstillingsforløbet kan projektlederen yderligere vælge at gennemføre en SWOT. Den uddyber årsagen til risici og afdækker yderligere muligheder for at imødegå dem. Der findes mange forskellige modeller til vurdering af sværhed, kompleksitet, risici, trusler og usikkerheder. Nogle er meget omfattende og til tider meget teoretisk eller matematisk funderede. Andre er løst formulerede rammer, hvor projektdeltagere selv definerer spørgsmålene i analysen.

Gennemførelse

Risikoanalysen foretages i forbindelse med planlægningen af projektet. Opstartsmødet bør desuden udnyttes – enten til at gennemføre den indledende analyse eller til at gennemgå resultatet af den. Der bør under alle

omstændigheder afsættes en del af opstartsmødet til at tænke i risici. På opstartsmødet kan opgaven formuleres direkte, for eksempel således:

„*Hvilke forhold, i form af uheld eller mulige trusler, kan vælte vores plan?*“

Dokumentér ved hjælp af minimumsmodellen „Risikoanalyse“. For hver mulig risiko udfyldes de øvrige kolonner således:

S = Sandsynlighed

- 1 = Det vil sandsynligvis ikke ske, men det er dog muligt
- 2 = Det er lige så sandsynligt, at det sker som at det ikke sker
- 3 = Det vil sandsynligvis ske

K = Konsekvens

- 1 = Konsekvensen kan opsuges med planens nuværende aktiviteter
- 3 = Der skal gøres noget for at afbøde konsekvensen
- 10 = Planen skrider hvis dette sker

P = Produkt af sandsynlighed og konsekvens

$$S * K$$

Mulige foranstaltninger

Hvad skal der gøres? Det kan enten være for at fjerne/reducere risici eller for at definere et beredskab.

Figur 5.3 viser en minimumsmodel med fem identificerede risici.

RISICI	S	K	P	MULIGE FORANSTALTNINGER
1. Ændringer til kravspecifikationen	2	10	20	Godkendende review efter hvert faseskift
2. Manglende ressourcer til test	2	10	20	Træf aftale med ekstern underleverandør om ressourcer
3. Udkiftning i projektgruppen	1	10	10	Videnspredning via reviews og workshops
4. Forsinket levering af hjælpetekster	3	3	9	Skriv formel aftale om underleverance, med tidspunkt for levering
5. Ændringer til dialogstandarden	1	3	3	Stram ændringsstyring når det sker

Figur 5.3: Risikomodel.

Behandling

Risici med 9 point eller mere giver umiddelbart anledning til handlinger. De øvrige vurderes. Handlinger er enten forebyggende eller beredskabsgivende. Projektlederen skal vælge forebyggende, når omkostningerne tillader det. Men da beredskabsgivende handlinger generelt er billigere, er der brug for at vurdere hver risiko. Vurdér hver pointstørrelse og tag hensyn til hvor tallet stammer fra:

- En høj sandsynlighed og høj konsekvens skal udmøntes i forebyggende (præventive) aktiviteter
- En lav sandsynlighed peger på behov for beredskab
- En høj sandsynlighed, men lav konsekvens, kan nøjes med beredskab.

De nødvendige handlinger er indlysende på visse kombinationer. Fx S = 3 og K = 10 siger at det sandsynligvis sker, og at planen skrider. Point på mindre end 9 udløser normalt kun beredskab, hvis nogen handling overhovedet.

Parathed

Når et uheld er sket, kan det ikke gøres usket. Men netop i den situation er der ingen grund til at tilføje rådvildhed. En beredskabsplan kan være guld værd. De beredskabsgivende aktiviteter definerer, hvad der skal gøres, når en risiko materialiserer sig. Det øger „paratheden“ i en situation, hvor det behøves mest.

Tilbageværende risici

Brug kriterier for om en risiko overhovedet skal imødegås:

- Lille sandsynlighed for at det indtræffer
- Begrænsede konsekvenser
- En tilstrækkelig accept blandt interesserne
- Dyrere at imødegå end at fjerne
- Kan det nås, eller vil tiden være forpasset?
- Vi evner det ikke
- Introducerer vi nye risici?

Projektet i IT-organisationen								
Risikoelement	Sandsynlighed			K	P	Risiko-styring	Hvem	Tid
	høj = 3	medium = 2	lav = 1	S				
Projektlederens kvalifikationer	* Det er projektlederens første projekt af denne type * Projektlederen har 1-2 lignende projekter bag sig * Projektlederen har en række lignende projekter bag sig	3 1 0	2 3 6	Kollegial coaching	Projekt-leader	xx.xx		
Projektlederens ansvar og kompetence	* Ansvar og kompetence er ikke klart beskrevet * Ansvar er beskrevet, men kompetencen er uklar * Klart beskrevet kompetence svarende til ansvaret	3 2 0	2 3 6	Møde m/styre-gruppen	Projekt-leader	xx.xx		
Styregruppe for projektet	* Der findes ingen styregruppe for projektet * Styregruppe eksisterer, men den er ikke repræsentativ * Repræsentativ styregruppe, lav beslutningshastighed * Repræsentativ og beslutningsdygtig styregruppe	3 2 1 0	2 3 6	Detaljeret model for information & rapport	Administrator	xx.xx		
Opgavens vigtighed for organisationen	* Opgaven er blot en blandt mange og har lav prioritet * Opgaven har almindelig prioritet * Opgavens løsning er højt prioriteret	3 1 0	0					
Fuldtids projektleder	* Projektleder er ikke fuldtids og har andre vigtige opgaver * Projektleder er ikke fuldtids, men projektet er hovedopg. * Projektlederen er fuldtids	3 1 0	0					
Projektgruppens kendskab til forretningsområdet	* Intet kendskab til det generelle eller specifikke område * Kun kendskab til det generelle forretningsområde * Grundigt kendskab til det specifikke forretningsområde	3 1 0	1 10 10	Workshop med brugere	Projekt-leader	Løbende		
Projektgruppens kendskab til udviklingsmetoden	* Intet kendskab til udviklingsmetoder * Gruppen har tidligere arbejdet med andre metoder * Projektgruppen har kendskab til den anvendte metode * Grundig erfaring med den anvendte metode	3 2 1 0	1 10 10	Arrangér seminarer med leverandøren	Projekt-gruppen	Løbende		
Motivation i projektgruppen	* Projektgruppen er demotiveret * Gruppen er ikke motiveret, men er åben for initiativer * Højt motiveret projektgruppe	3 2 0						
Referencegruppe	* Der eksisterer ingen referencegruppe for projektet * Referencegruppe eksisterer, men nytteværdien er lav * Gruppen eksisterer, men har ringe kendskab til projektet * Velfungerende referencegruppe	3 2 1 0						
Interne ressourcer versus konsulenter	* Projektet udelukkende bemandet med konsulenter * Konsulenter benyttes kun til at 'lukke huller' * Projektet bemandet med interne ressourcer	3 1 0						

Figur 5.4: Eksempel på en færdig model til et generelt nyudviklingsprojekt.

Antal involverede afdelinger	* Mere end 5 afdelinger involveret i projektet * 2-4 afdelinger involveret * Kun en afdeling involveret	3 2 0				
Antal eksterne underleverandører	* Mere end 3 eksterne underleverandører * 1-2 underleverandører * Ingen eksterne underleverancer	3 2 0				
Support fra underleverandører	* Underleverandører har ingen eller kun ringe kendskab til det leverede produkt * Underleverandører har nogen kendskab til produktet, men kan ikke klare komplikerede problemer * Solidt kendskab til produktet hos underleverandører	3 2 0				
Afhængighed af andre IT-projekter	* Afhængighed af et eller flere projekter med høj risiko * Afhængighed, men risiko i projektet forventes at være lav * Ingen afhængighed	3 2 0				

Projektplanlægning og -opfølgning

Risikoelement	Sandsynlighed	K	P	Risiko-styring	Hvem	Tid
	høj = 3 medium = 2 lav = 1					
Målbeskrivelse	* Ingen målbeskrivelse. Opgaven er blot defineret som leveringen af et system * Uklar målbeskrivelse eller målbeskrivelse uden accept * Klar målbeskrivelse med kvantificerbare mål	3 2 0				
Projektstørrelse	* Over 10.000 mandtimer * Projektstørrelse på 1000 - 10.000 mandtimer * Under 1000 mandtimer	3 2 0				
Antal underprojekter	* Tre eller flere underprojekter * To underprojekter * Ingen underprojekter	3 2 0				
Tidspunkt for estimat	* Før foranalyse * Før kravspecifikation * Før design * Efter design er godkendt	3 2 1 0				
Hvor meget er skønnet	* Både omfang og produktivitet er skønnet * Omfanget er stabilt, produktivitet er skønnet * Omfang er stabilt, produktivitet er kendt	3 2 0				
Metode anvendt til estimering	* Ingen metode anvendt * Metodeelementer anvendt * Metode anvendt som tager højde for usikkerheder	3 2 0				
Usikkerhed ved estimerater	* Usikkerhed større end 25% * Usikkerhed på 15-25% * Usikkerhed på 5-15% * Usikkerhed under 5%	3 2 1 0				

Projektplanlægning og -opfølgning

Risikoelement	Sandsynlighed			K	P	Risiko-styring	Hvem	Tid
	høj = 3	medium = 2	lav = 1	S				
Accept af projektplan	* Planen er ikke blevet publiceret og kendes kun af få * Kun delvis accept af planen * Planen er kendt og accepteret af alle interesserter	3 2 0						
Beskrivelse af aktiviteter	* Aktiviteter eksisterer kun som overskrifter * Kun stikordsagtige beskrivelser af aktiviteter * Detaljeret beskrivelse af aktiviteter	3 2 0						
Nedbrydningsgrad i planen	* Kun grovplan eksisterer * Planen er nedbrudt enkelte steder * Systematisk nedbrydning af aktiviteter	3 2 0						
Review af plan	* Planen er kun gennemarbejdet af projektlederen * Uformelt review af projektplanen er foretaget * Formelt review foretaget med kvalificerede kolleger	3 2 0						
Kvalitetsmål og kvalitetsplaner	* Kvalitetsmål er ikke kendt; ingen kvalitetsplan * Spredte kvalitetsaktiviteter mod vase kvalitetsmål * Klare kvalitetsmål og kvalitetsaktiviteter som grundlag for at nå målene	3 2 0						
Procedurer for projekt-opfølgning	* Ingen klare procedurer for opfølgning på projektet * Uformelle procedurer for opfølgning * Klare procedurer som er afprøvet i organisationen	3 2 0						
Procedurer for ændringsønsker	* Ingen procedure etableret for projektet * Procedure findes, men bruges ikke konsekvent * Procedure er godkendt af interesserter og følges	3 2 0						
Stabilitet af systemkrav	* Kravene er ustabile og ændrer sig meget * Nogen ændringer til kravene * Stabile krav	3 2 0						

Teknik og løsning

Risikoelement	Sandsynlighed			K	P	Risiko-styring	Hvem	Tid
	høj = 3	medium = 2	lav = 1	S				
Antal snitflader til andre systemer	* Mere end 3 snitflader til andre systemer * 2 snitflader * Ingen snitflader til andre systemer	3 2 0						
Erstatning versus nyt system	* Det første system på området * Systemet bygger på tidligere komponenter * Systemet er en direkte erstatning for gammelt system	3 2 0						

Grad af genbrug	* Intet genbrug muligt * Komponenter kan genbruges, men omfang ikke vurderet * Høj grad af genbrug er vurderet eller afprøvet	3 2 0				
Svartidskrav	* Der er skrappe krav til svartider * Der forventes svagt forbedrede svartider * Svartider er uvæsentlige for systemet	3 2 0				
Systemets kompleksitet	* Det mest komplekse systemorganisationen har lavet * Komplekst system, men organisationen har tidligere lavet lignende systemer med succes * Simpelt system	3 2 0				
Krav om nyt hardware og software	* Ukendt platform skal installeres som forudsætning * Platform er delvist ukendt * Enkelte nye H/W og/eller S/W komponenter * Ingen nye komponenter	3 2 1 0				
Kendskab til programme-ringssprog	* Projektgruppen har ringe kendskab til anvendte sprog * Enkelte medlemmer har godt kendskab * Grundigt kendskab til programmeringssprog	3 1 0				
Udviklings-platform	* Platformen skal udvikles som en del af projektet * Udviklingsplatformen eksisterer, men er ikke afprøvet * Afprøvet platform er til rådighed	3 2 0				

Implementering

Risikoelement	Sandsynlighed	K	P	Risiko-styring	Hvem	Tid
	høj = 3 medium = 2 lav = 1					
Antal berørte	* En stor del af organisationen vil blive berørt * En mindre del af de ansatte vil blive berørt * Få personer vil blive påvirket af det nye system	3 2 0				
Reduktion i arbejdsstyrke	* Kraftig reduktion i arbejdsstyrke * Få reduktioner, personale kan overføres til andet arbejde * Ingen personalereduktioner	3 1 0				
Geografiske lokationer	* Mere end 3 forskellige geografiske lokationer * 2-3 lokationer * Kun 1 lokation	3 2 0				
Grad af ændring af organisationen	* Vidtgående ændringer på tværs af afdelinger * Nogen ændringer inden for den bestående organisation * Ingen ændringer i organisationen påkrævet	3 1 0				
Strategi for implementeringen	* Big Bang implementering på alle lokationer * Big Bang implementering på enkelt lokation * Gradvis implementering på alle lokationer * Gradvis implementering på enkelt lokation	3 2 2 0				

Forståelse af implementeringsprocessen	* Ingen forståelse for implementeringsprocessen * Ringe forståelse, kun få implementeringaktiviteter * Aktiviteter igangsat, men ingen opfølgning * God forståelse, aktiviteter igangsat og opfølgning på alle påkrævede aspekter	3 2 2 0				
Afhængighed af andre projekter i brugerens organisation	* Afhængighed af projekter med høj risiko * Afhængighed af et projekt, men med lav risiko * Ingen afhængighed	3 1 0				
Brugerorganisationen						
Risikoelement	Sandsynlighed	K høj = 3 medium = 2 lav = 1	P S	Risiko-styring	Hvem	Tid
Brugerrepræsentanter	* Ingen brugerrepræsentanter udpeget * Ressourcer er allokeret, men har andre opgaver * Repræsentanter er udpeget og udfylder deres opgave	3 2 0				
Superbrugrens mål med projektet	* Superbrugeren har intet specifikt mål med projektet * Superbrugeren har andre mål end projektet * Mål er uklart for superbrugeren * Klar opfattelse af målet i overensstemmelse med brugerorganisationens mål	3 3 2 0				
Superbrugrens kendskab til forretningsområdet	* Intet kendskab til forretningsområdet * Kun generelt kendskab til forretningsområdet * Detaljeret kendskab til forretningsområdet	3 1 0				
Superbrugrens ledelsesmæssige kompetence	* Superbrugeren har ingen ledelsesmæssig kompetence * Superbrugeren har begrænset ledelsesmæssig kompetence * Superbrugeren har fuld ledelsesmæssig kompetence	3 2 0				
Holdning til IT-organisationen/leverbændøren	* Skepsis over for IT-organisationens evne til at kunne levere til tiden * Skepsis over for IT-organisationen generelt * Godt tillidsfuldt forhold mellem brugerorganisationen og IT-organisationen	3 1 0				
Brugerorganisationens forståelse af mål	* Kun ledelsen forstår målene med projektet * Forståelse hos ledelse og mellemledere * Bred forståelse i hele organisationen	3 1 0				
Modstand mod projektet i brugerorganisationen	* Projektet er kun støttet af topledelsen; resten af organisationen er mod projektets gennemførelse * Projektet får også støtte fra mellemledelsen * Bred opbakning om projektet	3 1 0				

Udvidet model

En udvidet model har flere kolonner for hver risiko. Projektledere kan udvide, som der er behov for. Den her beskrevne model er et eksempel, som indeholder de supplerende kolonner:

Sandsynlige årsager

Når man går dybere ind i en analyse af de mulige årsager til et forventet problem, bliver det lettere at definere virkningsfuld forebyggelse. Denne del af analysen kan være af interesse for alle i organisationen. Ved at finde og fjerne de mulige årsager, kan risikoen fjernes permanent.

Separate „forebyggende“ og „beredskabsgivende“ aktiviteter

En rent praktisk opdeling.

Hjem handler?

Det kan enten være de udførende på aktiviteterne, eller formålet kan være at dokumentere, hvem der har påtaget sig ansvaret for håndtering.

Hvornår?

Udfyldes når der er brug for „senest hvornår der skal gøres noget“.

Risici for planskred

De følgende 14 punkter er hentet fra en undersøgelse blandt projektledere om hvilke områder der, hvis der opstod problemer på dem, gav risiko for planskred. De er ment som inspiration uden at være en egentlig checkliste. Afhængigt af det konkrete projekt kan et punkt rubriceres som „svaghed“ eller som „trussel“.

Følgende problemtypes giver risiko for planskred:

1. Problemer med at få detaljeret kravene tilstrækkeligt
2. Ringe deltagelse af projektdeltagerne i planlægningen
3. Problemer med at få udpeget ressourcer og bemandet projektet
4. Ingen klar godkendelse af projektplanen i ledelsen
5. Ingen procedure for behandling af ændringsønsker
6. For svag håndtering af konflikter/kriser
7. Svært at vurdere risici
8. Projektet organisation står ikke klart for alle
9. Ikke klar rolle og ansvar for driftsrepræsentanter

- 10.** For hurtigt organiseret opstartsmøde
- 11.** Svært at måle projektets fremdrift objektivt
- 12.** Dårlig kommunikation med ledelse
- 13.** Dårlig kommunikation med bruger/sponsor
- 14.** Uerfaren projektledelse.

Udbyg/ajourfør/rapportér/følg op

Analysen skal ajourføres og udbygges i hele projektets levetid. Ændringer, herunder også blot i „sandsynligheder“ eller „konsekvenser“, giver projektlederen anledning til at overveje om en revideret analyse skal distribueres til interesserterne. Beslut på opstartsmødet om det gøres uanset ændringens størrelse, eller om der kræves en vis størrelsесændring af risikobillet. Følg op ved milepæle, og følg op ved projektafslutning. Vurdér om den præventive imødegåelse var for ringe eller for overdreven, og om beredskabet var godt nok.

Handlerier giver aktiviteter på planen

Både de forebyggende og de beredskabsgivende aktiviteter udløser nye eller ændrede aktiviteter på planen. Aktiviteterne skal placeres i tid, de skal bemandes og indgå i en milestensopfølgning. De ekstra aktiviteter på planen er den investering, projektet må foretage, for at få den til at holde.

Fordele ved risikoanalyse

De primære årsager ligger i formålet, dvs. at overholde planen og at kunne forudsige projektets forløb. Men der er yderligere afledte fordele:

- Projektlederens udsyn forbedres
- Ledelsens ansvar belyses
- Forbedrer grundlaget for at vælge de rigtige strategier
- Mindre afhængighed af held
- Bedre rapportering til interesserter og dermed forbedring af deres beslutningsgrundlag.

Model med forberedte spørgsmål

Resultatet i figur 5.3 bygger på at deltagerne i risikoanalysen selv definerer de mulige risici. Man kan som alternativ benytte sig af en model med faste risikoelementer, og med forslag til hvornår et element skal vurderes som risikabelt. Modelerne findes i mange udgaver, og der vælges en model der

passer til projektets type. For eksempel om det er et generelt udviklingsprojekt, et anskaffelsesprojekt, udvikling af et standardsystem, et integrationsprojekt, en brancheløsning, eller en ny release til et eksisterende produkt.

I figur 5.4 er der vist et eksempel på en færdig model til et generelt nyudviklingsprojekt. Dvs. at der opereres med et projekt i en IT-organisation og en levering til en kendt brugerorganisation. Modellen indeholder fem hovedområder:

- ▶ Projektet i IT-organisationen
- ▶ Projektplanlægning og -opfølgning
- ▶ Teknik og løsning
- ▶ Implementering
- ▶ Brugerorganisationen

Hvert hovedområde indeholder faste risikoelementer der skal vurderes i risikoanalysen. Der er forslag til hvornår et element skal vurderes som risikabelt, og dermed hvilken værdi der skal sættes i S (sandsynlighed). Der er derimod ingen vejledning i hvordan kolonnen for konsekvens skal udfyldes, det er alene projektgruppens opfattelse, der er afgørende.

Sandsynlighed kan udfyldes valgfrit mellem 0 og 3. Konsekvens udfyldes med 1,3 eller 10 i lighed med modellen i figur 5.3. I figur 5.4 er hovedområdet 'Projektet i IT-organisationen' udfyldt for et tænkt projekt.

Sådanne modeller har fordele og ulemper. Det er en stor fordel, at der tages stilling til en stribe områder, uanset om der er en synlig risiko. I modellens første spørgsmål tages der stilling til projektlederens kvalifikationer, og der er sat et 0. Alligevel er det en fordel at elementet gennemgås, da diskussionen typisk kan lede ind på mange andre områder. Processen bliver mere omfattende. Ulempen ved denne type færdige model er at, det er lettere at holde sig til spørgsmålene og glemme at tænke selv. Det forhindres ved at starte med en uformel brainstorm, før de forberedte spørgsmål gennemgås. Man kan begynde med et tomt skema og selv definere hovedområder og risikoelementer.

Der findes mange af disse modeller, og de er lette at oprette i et regneark, eller i en database, hvis der er brug for at gemme dem, regne videre på tallene eller føre sig frem med lidt grafik.

5.5 SWOT

– Strengths, Weaknesses, Opportunities, Threats

Når risikoanalysen udvides til en SWOT, giver det mulighed både for at identificere flere trusler og for at identificere flere handlinger for at imødegå dem.

SW-delen beskæftiger sig med interne faktorer i projektgruppen. De varierer i forhold til projektgruppens sammensætning eller valgte arbejdsmåde.

OT-delen handler om ydre faktorer, der er invariante i forhold til ethvert projekt, der måtte blive sammensat.

Imødegå „svagheder“ og „trusler“ med „styrker“ og „muligheder“

SWOT'en bygger på at i modsætning til at lære nye metoder, teknikker eller værktøjer, er det mere effektivt at bruge eksisterende muligheder i projektet eller i dets nære omverden.

Indholdet i en SWOT

Interne faktorer	
Strengths (styrker)	Weaknesses (svagheder)
Høj motivation	Ringe kendskab til metoder og teknikker
Erfarne brugere	Ikke direkte adgang til brugere i projektgruppen
Faglige færdigheder i top	Ringe viden om teknisk platform
Velfungerende organisation	Manuel dokumentation
Intet behov for uddannelse	Uerfarne udviklere
Stabile krav	Urealistiske planer
Formel kvalitetsstyring	
Eksterne faktorer	
Opportunities (Muligheder)	Threats (Trusler)
Nye versioner af værktøjer	Høj kompleksitet af nuværende systemer
Afslutning af andet projekt	Begrænsede maskinressourcer
Implementering af andet produkt	Interessenternes engagement er lavt
Erfaren ledelse	Urealistiske tidskrav
Stabil hardware	

5.6 Projektopstartsseminar

Formål

Formålet med seminaret er at sikre en effektiv opstart.

Seminarets indhold består af aktiviteter der under alle omstændigheder er nødvendige at gennemføre i projektet. Seminarets formål er at gennemføre dem effektivt, på det rigtige tidspunkt og med de rigtige ressourcer.

Seminaret kan gennemføres når projektets mål er beskrevet i et projektforslag eller et kravstudie.

Opstartsseminaret er en måde at systematisere brugen af en række andre værktøjer og teknikker på. Interessentanalyse, informationsanalyse, kvalitetsstyring og tidsestimering er velegnede emner til en opstart. Emnerne bearbejdes før seminaret, og resultatet gennemgås af deltagerne. Der vil også være en afsluttende bearbejdning og dokumentation efter seminaret.

Gennemførelse

Den praktiske gennemførelse består i at samle projekt deltagerne, en eventuel projektadministrator, styregruppen og nøgleinteressenterne. Tidsrummet kan være mellem tre timer og tre dage. Afhængigt af projektets karakter kan medlemmer af referencegruppen også være relevante.

Seminaret afvikles af hensyn til arbejdsroen bedst uden for virksomheden. En dagsorden svarende til forløbet af seminaret kan se således ud:

1 Godkendelse af dagsordenen

Projektlederen har på forhånd opstillet forslag til dagsorden, der som første punkt har en godkendelse af dagsordenen.

2 Oprettelse af projekthåndbog

Strukturen i projekthåndbogen præsenteres. Opret et site på nettet, således at der er let adgang til indholdet. Der skal blot være adgangskontrol, der svarer til den information, der lægges ud.

3 Gennemgang af projektets organisation

Organisationen gennemgås, og de indtil nu kendte medlemmer præsenteres ved navn. Det enkelte elements opgaver, ansvar og kompetence slås fast.

Aktører:

- Køberen stiller ressourcer til accept- og brugertest
- Leverandøren stiller ressourcer til systemtest
- Der udpeges en uafhængig testleder, der koordinerer planlægning og gennemførelse af testen, opsamling og behandling af fejlrapporter, godkendelse af gentaget test og rapportering af testens fremdrift.

Teknologi:

- Testværktøjer til Capture/Replay
- Testværktøjer til load/performance-test
- Reviews og inspektioner
- Kategorisering af den enkelte fejl samt måling af antallet pr. Function Points.

Organisation:

- Testleder refererer til styregruppen.

Implementeringsstrategi

Strategiske mål og overvejelser

Første leverance må koste én dags instruktørstøttet undervisning pr. brugssted. Resten skal klares af den indbyggede hjælp og støtte. De efterfølgende leverancer må koste 1/2 dag hver. Den tekniske strategi bygger på at både system- og brugertest kan gennemføres under forhold der ligner virkeligheden. Desuden arrangeres en betatest på 2 måneder på 4 brugssteder hvor systemet bruges både teknisk og juridisk. Det gamle system bruges kun til paralleltest. Den brugsmæssige implementering gennemføres i et særskilt projekt, der forankres i køberens organisation. De deraf afgtede krav til brugergrænsefladen, indbygget hjælp og støtte, indbyggede undervisningsfaciliteter og lignende skal indgå i analysefasen.

Taktiske tiltag

Opgaver:

- Arranger betatest
- Beskriv den mulige paralleltest.

Aktører:

- Brugere fra købers organisation
- Projektdeltagere.

Teknologi:

- Reviews og inspektioner.

Organisation:

- Separat projekt i købers organisation.

6.4 Aktivitetsplanlægning

Planlægning består i at identificere og dokumentere de aktiviteter der skal udføres for at realisere målene. Aktiviteterne skal desuden støtte og realisere de valgte strategier. Resultatet er en opgaveliste, evt. i første omgang uden tidsestimater og ressourcenavne. Det skal senest være på plads, når de fysiske planer fremstilles.

Hvorfor er planlægning så vigtig?

– fordi oversete aktiviteter vælter den gode planlægning, og glemte aktiviteter river planen i stykker. Overvej selv: Hvor lang tid varer en aktivitet? Den kan for eksempel være på 5 dage. Hvis den glemmes i planlægningen, og det senere viser sig at den skal gennemføres, har projektet en forsinkelse på 5 dage. Oversete aktiviteter svarer altså til 100 % underestimering. Med 2-3 oversete aktiviteter fra start er der indbygget et skred i planen før projektet er kommet i gang.

Planlægning er også vigtig fordi den kaster lys over opgaven og problemerne. Man „løser“ til en vis grad opgaven ved at planlægge den detaljeret.

Planlæg test og kvalitetsstyring

Læg vægt på en synlig kvalitetsstyring ved at specificere konkrete test- og reviewaktiviteter. Vurdér om testen er planlagt detaljeret nok, og om projektet har de nødvendige reviews og inspektioner. Vis hvert review som en selvstændig bemandet aktivitet. Det øger sandsynligheden for at det senere bliver afholdt. Stil et par gode spørgsmål:

- Hvilke aktiviteter skal med for at udvikle det rigtige produkt til aftalt tid og pris?
- Er der tilstrækkelig kontrol af de produkter projektet leverer?

De to spørgsmål går på henholdsvis udviklings- og testaktiviteter.

Afhængigheder mellem aktiviteter

Når alle aktiviteter er identificerede, kan afhængighederne imellem dem

defineres. Det kan udskydes, til man skal fremstille de fysiske planer. Motivet til evt. at tage fat på det nu er, at der kan identificeres afledte aktiviteter.

Standardaktivitetslister

En standardaktivitetsliste er uvurderlig for planlægningsprocessen. Den kan være købt eller selvudviklet, og det betyder mindre om den hedder: „Task List“ eller WBS (Work Breakdown Structure) eller er i form af en detaljeret projektmodel. Hovedsagen er at der findes standardaktivitetslister til de mest almindelige typer af projekter, for eksempel:

- Et traditionelt udviklingsprojekt
- Objektorienterede projekter
- Webløsninger
- Anskaffelse af standardprogrammel
- Integrationsprojekter
- Videreudviklingsprojekter

Der er normalt brug for at hente aktiviteter fra flere af listerne til et konkret projekt.

Andre metoder til at identificere aktiviteter

Aktivitetslisten leverer den store mængde af aktiviteter til planen. Resten fås fra en række forskellige kilder og ved hjælp af forskellige teknikker. Her er nogle gode kilder:

- Kravspecifikationen. Det er klart en mulighed at læse den igennem for at få styr på de opgavespecifikke aktiviteter.
- Strategiens taktiske tiltag skal ende som operationelle aktiviteter.
- Egen erfaring. Tænk tilbage på evt. tidligere glemsomhed.
- Andres erfaring ved at trække på projektdeltagerne og andre projektledere.
- Interessenterne er gode kilder i forhold til deres egne behov. De er involverede i opgaven og har en interesse i at få et roligt forløb med planlagte aktiviteter.
- Andre projekter der ligner dette. Se i databasen over afsluttede projekter.
- MindMap. Det er ganske vist en teknik og ikke en kilde. Men den kan bruges til at nedbryde hovedaktiviteter til detailaktiviteter.

- Analyse af sammensatte ord er et supplement til at detaljere kravspecifikationen. Fx „Styringsgrundlag for likviditetsudjævning“. Split det op og spørg:
 - Styring af hvad?
 - Grundlag for hvad?
 - Hvordan defineres likviditet?
 - Hvad forstås der ved „udjævning“?

Ved at nedbryde komplekse krav kan planen gøres tilsvarende mindre kompleks.

- *Nedbryd til kendte opgaver.* Fortsæt aktivitetsnedbrydningen til aktiviteterne kan overskues eller genkendes.
- *Analysér produkter.* Mangler planen aktiviteter i forhold til fremstille leverancer og mellemresultater.
- *Analysér milepæle.* Der er ofte særlige aktiviteter i forbindelse med milepæle.
- *Analysér projektets metodikker.* Skal der bruges OO, webservices, workshops, præventiv test etc. Det kan alt sammen give anledning til særlige aktiviteter.

Husk: Det er de glemte aktiviteter der riller en ellers god plan i stykker. Erfaringer viser desuden at oversete udviklingsaktiviteter skaber forsinkelser, mens oversete test- og kvalitetsstyringsaktiviteter både skaber forsinkelser og er årsag til leverancer med fejl og mangler.

Fordele af god aktivitetsplanlægning

- Bedre overblik fra start
- Enklere og mere præcis opfølgning
- Afvigelser bliver mere synlige, herunder deres konsekvenser
- Hurtigere reaktion på afvigelser
- Korrekt placering af milepæle
- Gennemarbejdet kvalitetsstyring.

Med disse fordele in mente må det betegnes som en misforståelse at betragte detaljeret planlægning som bureaukrati eller på anden måde en belastning for projektet. Tag igen test- og kvalitetsstyring som eksempel. Med nogle klare målsætninger som grundlag og en sammenhængende

strategi for at realisere dem er det direkte produktivitetsfremmende og kvalitetssøgende at identificere de konkrete aktiviteter til test, reviews og inspektioner.

6.5 Tidsestimering

Generelt

Et estimat er et skøn der har lige stor sandsynlighed for at ligge over som under det endelige resultat. Usikkerheden skal altså være lige stor til begge sider. Når man ser på alle de realiserede tidsregistreringer i en organisation, skal der være mange estimer der rammer plet. Dernæst skal der være lige mange der rammer 10 % under og 10 % over, lige mange der rammer 20 % under og 20 % over etc.

Man kan nøjes med at kvalitetsstyre de kritiske estimer eller at fokusere på en vis størrelse. En 50-pct.'s overskridelse på et estimat på 2 dage, rammer knap så hårdt som en 50-pct.'s overskridelse på 3 uger.

Alle estimer skal sammenlignes med de senere realiserede tal. Både på den enkelte aktivitet, på hovedaktiviteter, fx „analyse“, „design“, „test“, på hele projektet, på alle estimer i en gruppe, i en afdeling og i hele organisationen. Dels fordi det indgår i projektevalueringen, dels for at kunne ajourføre sine historiske tal.

Hvornår er estimatet lige i plet?

Det er almindeligt at regne estimer der ligger inden for +/- 10 % af det realiserede, for pletsud. Men under forudsætning af at der er lige meget under- og overestimering set over alle estimer – ellers taber man penge, anseelse og troværdighed. Det virker uprofessionelt at ligge konstant 10 % under. Det er almindeligt at udvide denne +/- 10 %-regel til:

Ni ud af ti estimer skal ligge inden for +/- 10 % afvigelse.

Det åbner plads til vildskuddene, samtidig med at den store mængde dettailestimaters fejlskøn udligner hinanden. Man rammer totalt set plet ved at tjene det ind på karrusellerne der er sat til på gyngerne.

Usikkerheden

Et estimat er meget lidt værd uden viden om usikkerheden. Nyttens og bru-

gen af estimatet er bestemt af om der er en 10-pct.'s eller en 50-pct.'s usikkerhed. Projektlederen vil typisk arbejde med stor usikkerhed i starten af projektet, mindske den hen ad vejen og på et tidspunkt kende det endelige resultat med stor sikkerhed. Men uanset usikkerhedens størrelse er det helt afgørende at kende den. Usikkerheden kan være skønnet af projektlederen eller beregnet i den estimeringsteknik man bruger. I det successive princip er det almindeligt at bruge standardafvigelsen som usikkerhed.

Forudsætninger

Angående forudsætninger kan man være lige så kontant som ved usikkerheder:

Estimator er intet værd uden viden om de forudsætninger der ligger til grund for dem.

Det er faktisk værre end intet fordi et manglende estimat kan give en vis tilbageholdenhed – i modsætning til at kende et estimat, men ikke dets forudsætninger, da det sjældent holder nogen tilbage fra at referere til estimatorne. Men er der for eksempel forudsat:

- ▶ Erfarne/uerfarne projektdeltagere?
- ▶ Stabil/ustabil kravspecifikation?
- ▶ Et stabilt/ustabilt udviklingsmiljø?
- ▶ Gode/ingen testmuligheder?

Uden viden om forudsætningerne er estimatet sårbart over for ændringer i disse.

Estimeringsteknikker

Estimeringsteknikker inddeltes i grupper:

- ▶ *Algoritmiske.* Primært optælling af funktioner og/eller data.
- ▶ *Egen erfaring.* Primært mandtid.
- ▶ *Ekspertvurdering.* Primært som „second guess“ og review.
- ▶ *Andre projekter (kopiering).* Kræver ensartethed, ensartethed, ensartethed (og en database).
- ▶ *Fasers, hovedaktivitetters og aktivitetters indbyrdes størrelse.* Særdeles effektiv efter analysefasen, men kræver også ensartethed.

som dette projekt skal betjene sig af. Lån dem som instruktør i et par dage, og bed dem om at vise hvordan de ville løse nogle af de aktuelle opgaver. Uddannelse er også en mulighed for at undgå spild og fejl. Det handler ikke altid om at lære sig det mest smarte. Den høje produktivitet og kvalitet opstår i mange tilfælde når tiden bruges på at producere brugbare ting første gang. Arrangér inspektioner med en indbygget læreproces, og sorg for at der er tid nok til at diskutere årsag til fejl og mangler.

6.7 Fysiske planer

De fysiske planer dokumenterer aktivitetsplanlægning, tidsestimering og ressourceallokering. Planerne bruges eksternt som dokumentation til interesserterne og internt i projektgruppen som arbejdsplaner.

Afhængigheder

Før planerne endeligt kan udarbejdes, skal afhængighederne på plads. Nogle er fundet i aktivitetsplanlægningen, andre er blot kommet til undervejs. Rigtig mange er blevet synlige i forbindelse med ressourceallokering, fordi konkrete personer, antallet af dem og deres evner også giver afhængigheder.

De faglige afhængigheder er de ægte. De opstår når en aktivitet først kan starte, efter at en anden er slut. Den ene aktivitet bruger den andens resultat som grundlag. Disse afhængigheder kan være dokumenteret allerede i en WBS (Work Breakdown Structure) eller i en projektmodel. De ægte afhængigheder er synlige fra start.

Ressourcerne antal giver andre bindinger. Hvis der er to ressourcer i stedet for 10, er der flere aktiviteter, der først kan starte, når en ressource bliver ledig. Andre bindinger opstår på grund af enkelpersoners evner og færdigheder. Når en bestemt færdighed kun kan udføres af visse personer, nyttet det ikke at lade aktiviteten starte teoretisk korrekt, hvis der ikke er en kompetent ledig person.

Det er værd at skelne mellem disse typer af afhængigheder, fordi nogle af dem kan fjernes ved at ændre på ressourcerne antal eller tidspunktet for deres deltagelse i projektet. Uddannelse er også en mulighed hvis behovet opfanges tidligt nok.

Grov- og detailplaner

Grovplanen viser hele projektets forløb. Den er især beregnet på at informere projektets omverden. Afsnittet om plankrav opstiller retningslinier for grovplanens indhold.

Grovplanen afspejler fysisk de valgte strategier. Den viser, om der er valgt delleveringer, versioner, RAD osv. Teststrategien afspejles lige så tydeligt, hvad enten det er en traditionel, en V-model eller en o-fejl.

Plankrav indeholder også anbefalinger for detailplanens udseende og indhold. Den bruges internt i projektgruppen til at styre det daglige arbejde. Detailplanen kan udarbejdes, når ressourcernes antal, erfaring og tilgængelighed er kendt.

Detailplanen tager afsæt i, at alle aktiviteter er kendte. Hvis der er store mangler, hjælper det ikke at resten er nok så detaljeret. Det bliver til et stramajbroderi med store, bare pletter. Det kunne gå i estimeringen, forudsat at det var velkendte aktiviteter. Detailplanens formål rækker længere, da den skal fungere som den daglige arbejdsplan, der skal kunne følges op på.

Samme plan i forskellige fysiske former

Med et projektstyringsværktøj, for eksempel MS Project, eller Planbee, kan den fysiske planlægning dokumenteres på forskellig vis. De mest almindelige er Gantt-skemaer, netplaner og ressourceoversigter.

Gantt-skemaet

Et Gantt-skema viser aktiviteterne, deres ressourcer og deres plads i kalenderen. Det dokumenterer planlægningen i et kalenderudseende. Gantt-skemaet bruges til at dokumentere både grov- og detailplaner. Det indgår i projektets eksterne dokumentation og skal derfor være læseligt for interessenterne.

Netplanen

Netplanen viser aktiviteternes indbyrdes afhængigheder. Den viser rækkefølgen som de kan og skal gennemføres i. Det er muligt at „gætte“ ved at se på Gantt-skemaet, men vished fås kun ved hjælp af en netplan. Den viser for hver aktivitet, hvilke andre, der skal være afsluttet, før den selv kan starte, hvilke der kan udføres samtidig, og hvilke der først kan starte når denne er færdig. Netplaner har fået en renæssance med de automatiserede værktøjer. De letter fremstilling og frem for alt vedligeholdelse af

netplaner. Det er ikke for meget sagt at et automatiseret værktøj er en forudsætning for at arbejde med netplaner.

Netplaner fremstilles når projektlederen anser det for sandsynligt at der sker mange ændringer af det planlagte. Det kan være ændringer af projektets indhold, risiko for forsinkelser, udskiftning af projektdeltagere eller hundredvis af andre årsager. Der er brug for at vide om ændringen kan opsluges i planen, eller om den rykker projektets slutdato. Netplanen viser den kritiske vej igennem projektet, forstået som de aktiviteter der i deres indbyrdes afhængighed udgør den længste vej. Hvis en ændring har konsekvens for den kritiske vej, viser netplanen det øjeblikkeligt.

Det giver som hovedregel kun mening at udarbejde netplaner for detaljplaner.

I projekter med ganske få afhængigheder kan de tegnes ind på Gantt-skemaet som pile mellem aktiviteterne. Netplanen er først besværet værd at fremstille når der er en vis mængde afhængigheder der skal analyseres.

Ressourceoversigter

For at kende og styre tidsplanlægning, ledighed og overbelastning i forhold til den enkelte ressource er der også brug for planer, der har ressourcer som indgang.

Milepæle

En milepæl repræsenterer et sted i projektets forløb, hvor mange enkelte aktiviteter udgør et logisk målepunkt. Detailplaner viser alle milepæle, men grovplaner viser kun milepæle med ekstern betydning. Projektlederen bruger milepæle til at vurdere om projektets fremdrift og færdiggørelsesgrad er som planlagt. Milepæle kan for eksempel være:

- ▶ Opstartsmødet er gennemført
- ▶ Grovplanen er godkendt
- ▶ Objekt-/data-/funktions-/hændelsesmodel er godkendt
- ▶ Brugervejledningen er godkendt
- ▶ Use Cases er skrevet og godkendt
- ▶ Testdata til brugertesten er skrevet og godkendt
- ▶ Designreviewet er gennemført
- ▶ Accepttestens indhold er godkendt
- ▶ Testplanen er udarbejdet
- ▶ Brugertesten er gennemført.

Listen kan gøres kortere eller længere, afhængigt af projektets størrelse og kompleksitet. Milepæle kan standardiseres for alle projekter, alternativt kan de skræddersys til det enkelte – i så fald typisk med en endelig liste efter opstartsmødet.

Slæk

Skal der slæk (dvs. luft) i planen til uventede aktiviteter eller forsinkelser? Det er lidt et spørgsmål om lederstil. Nogle projektledere mener at slæk i planen sænker produktiviteten, fordi et slæk altid vil blive brugt. Andre mener at der både er brug for slæk på udvalgte aktiviteter, imellem aktiviteterne og i hver fase af arbejdet. Rent teknisk kan man sige, at enten må man identificere alle aktiviteter og estimere dem med lige stor sandsynlighed for, at de tager kortere eller længere tid, eller også må man operere med slæk i planen. Og beslutningen skal tages, før der kan skrives fysiske planer.

Hvad er en god plan og en god aktivitet?

Der er nogle grundlæggende karakteristika for gode planer og aktiviteter, som bør ligge til grund for den fysiske planlægning.

Karakteristika for en god aktivitet på detailplanen:

- Kort: mellem 5 og 10 arbejdssage
- Afsluttes med et veldefineret fysisk produkt
- Uafhængig af andre aktiviteter (eller afhængig af færrest mulige)
- Høj værdi for planen i sit bidrag til projektets primære mål
- Forståelig og endda velkendt for projektdeltagerne.

Karakteristika for en god detailplan:

- Indeholder „gode“ aktiviteter
- Viser den kritiske vej
- Kendt og anerkendt af projektdeltagerne
- Realistisk (overflødigt at påpege?)
- Forståelig/overskuelig for planens målgruppe.

Plankrav

Dette afsnit opsummerer krav til grov- og detailplaner. Det er forslag, men blot det at tage stilling til dem forbedrer grebet om projektstyringen.

Generelt skal der tages stilling til brugen af work-packages og iterationer. Hver for sig har de indflydelse på den fysiske planlægning.

En work-package, der ikke har fået noget dansk navn, er en samling af aktiviteter der udgør et logisk hele. Man udfører jo ikke nødvendigvis hele dataanalysen efterfulgt af hele funktionsanalyesen. Man arbejder måske på en del af systemet, baseret på en række aktiviteter fra aktivitetslisten (WBS'en).

En work-package beskrives ved sit navn, arbejdet der skal udføres, forudsætningerne for at starte, den estimerede varighed, produkterne der leveres, kriterier for at godkende produkterne, evt. en SWOT for den konkrete work-package, og endelig ved de nødvendige ressourcer i form af antal og type af personer, maskiner, værktøjer, rejser og beslutninger.

En work-package kan være på et højt niveau i form af en levering på grovplanen (en hel dellevering). Den kan også optræde på detailplanen som en intern dellevering.

Iterationer er nødvendige i mange projekter. Det kan være i form af delleveringer, stærkt overlappende faser eller en egentlig „spiralstrategi“, hvor dele af aktivitetslisten gennemløbes flere gange. Det iterative skal overvejes grundigt, både i form og i omfang. Og det skal dokumenteres i planlægningen. Iterationer har konsekvens for flere discipliner: Planerne skal vise dem, estimaterne skal tillade dem, og opfølgningen skal tage højde for dem.

Plankrav – grovplaner

Følgende krav er fornuftige at stille til grovplaner:

- ▶ Viser alle faser i projektet med start- og sluttidspunkter.
- ▶ Når projektet leverer funktionaliteten i delleveringer, viser grovplanen den enkelte levering.
- ▶ Viser alle milepæle der involverer interesserter.
- ▶ Viser reviews der involverer interesserter, ofte sammenfaldende med milepæle.
- ▶ Grovplanen godkendes af styregruppen og af de involverede interesserter.
- ▶ Den anvendte estimeringsmetode skal dokumenteres.
- ▶ Estimaternes usikkerhed skal dokumenteres.
- ▶ Viser eksterne forudsætninger, herunder underleverancer og godkendelser.
- ▶ Viser de hovedopgaver i en fase, der kræver koordinering med interesserter. For eksempel:
 - vurdering af prototyper

- planlægning/udførelse af accepttest
- implementeringsaktiviteter.

Plankrav – detailplaner

Følgende krav er fornuftige at stille til detailplaner:

- Detaljerer projektets aktuelle fase, således at alle anvendte timer kan henføres til aktiviteter.
- Hvis den bygger på en standardaktivitetsliste, skal fravigelser motiveres.
- Indeholder alle aktiviteter, ikke kun projektgruppens, men også interesserternes.
- Alle aktiviteter er bemandede.
- Indeholder projektlederens egne aktiviteter selvom det kan være svært. For eksempel kan motivering af projektgruppen og teambuilding ikke vises som faste, afgrænsede aktiviteter, hvad derimod planlægning, opfølgning og de flest mulige øvrige aktiviteter både kan og skal.
- Viser underleverancer.
- Viser projektgruppens egne aktiviteter og underleverancer, for eksempel:
 - Specifikation af indholdet
 - Reviews
 - Modtagelse og endelig godkendelse.
- Viser alle reviews som bemandede aktiviteter.
- Detailplanen skal være godkendt af underleverandører og involverede interesserter.
- Forudsætningerne for detailplanens succes skal dokumenteres.
- Detailplanen skal vise et evt. iterativt forløb.
- Indeholder veldefinerede aktiviteter i forhold til SWOT, dvs.:
 - Aktiviteter der imødegår Weaknesses og Threats
 - Aktiviteter der udspringer af Strengths og Opportunities
- Viser den kritiske vej.
- Viser evt. work-packages.

Planer som aftalegrundlag

Den fysiske plan fungerer som en aftale mellem projektlederen og interesserterne. Hvis planens forudsætninger holder vand, således at de ressourcer, der planlægges med, også er til rådighed, står projektlederen inde for at planen er realistisk. Det samme gælder mellem projektlederen og projektdeltagerne. Hvis forudsætningerne opfyldes, er alle enige om at planen er realistisk.