Model-Driven Design of Object and Component Systems

Zhiming Liu[1(✉)] and Xiaohong Chen[2]

[1] Centre for Software Research and Innovation,
Southwest University, Chongqing, China
zhiming.liu88@gmail.com
[2] Singapore University of Technology and Design, Singapore, Singapore
xiaohong_chen@sutd.edu.sg

Abstract. The notion of software engineering implies that software design and production should be based on the types of theoretical foundations and practical disciplines that are established in the traditional branches of engineering. The goal is to make development of complex software systems more *predictable* and the systems developed more *trustworthy - safe, secure and dependable*. A number of theories have been well developed in the past half a century, including *Abstract Data Types*, *Hoare Logic*, *Process Calculi*, and *I/O automata*, and those alike. Based on them, techniques and tools have been developed for software specification, refinement and verification.

However, the theoretically sound techniques and tools have not been seamlessly integrated in practical software development, and their impact upon commonly-used software systems is still far from convincing to software engineering practitioners. This is clearly reflected by the challenges of their applications in engineering large-scale systems, including Cyber-Physical Systems (CPS), Networks of Things and Cloud-Based Systems, that have multi-dimensional complexities. Indeed, students are not often shown how the theories, and their underpinned techniques and tools, can better inform the software engineering they are traditionally taught. The purpose of this course to demonstrate such an effort.

We present a *model-driven design* framework for *component-based and object-oriented* software systems. We identify a set of UML notations and textual descriptions for representing different abstractions of software artefacts produced in different development stages. These abstractions, their relations and manipulations all have formalisations in the rCOS formal method of component and object systems. The aim is to allow the advantage of using precise models for development better appreciated. We organise the lecture notes into three chapters, each having a title page but all the references to literature are given at the end of Part III.

Keywords: Component-based architecture · Object-oriented design · Interfaces · Contracts · Design patterns · rCOS · UML

Z. Liu and Z. Zhang (Eds.): SETSS 2014, LNCS 9506, pp. 152–255, 2016.
DOI: 10.1007/978-3-319-29628-9_4

1 Part I: Introduction

1.1 Background and Organisation

This chapter is organised based on the materials that have been taught since 1998 at the University of Leicester, United-Nations University – International Institute for Software Technology (UNU-IIST[1], Macau) and Birmingham City University. The materials have also been adapted to and taught at training schools similar to the Summer School on Engineering Trustworthy Software Systems (SETSS) held in Chongqing in August 2014. Furthermore, these materials and the feedbacks from students have influenced the development of the *rCOS method*, which is a formal model-driven method of object and component systems. With the insight developed through research in the rCOS method, we taught the students how to prepare themselves for effective study and application of formal techniques and tools in software design and program verification.

Our aim is and has consistently been to show that in order to apply formal techniques, models and tools to software development projects, the requirements and the design, together with their models must first be developed. We demonstrate an informal process of requirements gathering and analysis as well as design patterns can be used to develop models that are formalisable. Thus they are a basis for reasoning about and verifying desired properties. We believe this will contribute to bridging the gap between formal techniques and their application in practical software development.

We focus on the requirements gathering and analysis, component-based architecture design and object-oriented design of components. The theme of the approach is model-driven development of component-based architectures, their interface-based decomposition and composition and detailed object-oriented design. Component-based architectures (or systems of systems architecture) with techniques and tools of interface-based composition, evolution and integration are seen as key to dealing with modern complex software systems, including cloud-based systems, internet of things (IoT), smart cities and cyber-physical systems (CPS).

Organisation. We divide this chapter into three parts in order for the reader to select different sections from each parts. Part I contains a brief introduction to the background and organisation in Sect. 1.1, a historical account of software engineering in Sect. 2, that is followed by a discussion of the basics of model driven development in Sect. 3. Though software development process models are not a focus topic of this chapter, the concepts of modelling, analysis and design activities and the artefacts they produce are useful for understanding the technical discussions, and thus they are introduced in Sect. 5. The technical discussion throughout the chapter is based on a case study described in Sect. 4.

Part II is on use case driven object-oriented requirements gathering, modelling and analysis. Section 6.1 discusses about use cases, their identification,

[1] It is now renamed to UNU-CS.

description and decomposition. Section 7 is about object-oriented modelling of the domain structure through identification of classes, their attributes and associations. Section 8 moves into understanding, modelling and analysis of functional behaviour of the requirements, followed by a summary of this part in Sect. 9.

Part III presents the techniques and models for component-based architecture design in Sect. 10.1, and for object-oriented design of the architecture components in Sect. 11. The component-based architecture model emphases on the contracts of the component interfaces, provides the basis of the object-oriented design of the components using design patterns for responsibility assignments to objects. Section 12 gives an overall summary of the chapter and discusses possible future developments.

At the end of each technical section, we relate the informal techniques to the rCOS formal method with references to publications. The materials in the textbook of Larman [42] are a major source of the knowledge and ideas in the discussion, developed through all the versions of the course notes, from the first version used at University of Leicester, through the tailored versions taught at the international schools, to the version used for the Software Design module at Birmingham City University.

2 Software Engineering

For a long time there was no authoritative account of when "software engineering" first appeared in the literature, but it is now widely accepted that the term was first coined by Anthony Oettinger in 1966, ACM President between 1966 and 1968, in his "letter to the ACM membership" [71], and then used by Hamilton [1,80] while working on the Apollo guidance software. The term was used in 1968 in the title for the world's first conference on software engineering, sponsored and facilitated by NATO [67]. The motivation for the conference was the so-called "software crisis", characterised by the symptoms of late delivery, over budget, product failing to meet specified requirements, and inadequate documentation [67]. The notion of software engineering was meant to imply that software design and production should be based on the types of theoretical foundations and practical disciplines that are established in the traditional branches of engineering. This meaning and aim of the term, though its content was yet to be defined, was clearly reflected in the discussions on development processes and cycles at the first conference, and the discussions on the notion of program correctness appeared as a key issue at a followup conference in 1969 [78].

2.1 Software Complexity

Though there are disputes about if there is a "software crisis", software development is hugely complex, and the source of the so-called crisis is just the inherent complexity of software development. Fundamental understanding of software complexity has contributed to the formation of major areas of software engineering and advances in these areas. In particular, complexity of software

development is characterised by the following four fundamental attributes of software [6,8,9]:

1. the complexity of the domain application,
2. the difficulty of managing the development process,
3. the flexibility possible to offer through software, and
4. the problem of characterising the behaviour of software systems.

Complex systems are open to total breakdowns [73], and consequences of system breakdowns are sometimes catastrophic and very costly, e.g., the famous Therac-25 Accident 1985–1987 [44], the Ariane-5 Explosion in 1996 [81], and the Wenzhou High Speed Train Collision[2] in 2011. Also the software complexity attributes are the main source of *unpredictability* in software development projects. Software projects fail due to our failure in mastering the complexity [35]. Advances in software engineering have been largely driven by understanding of and seeking solutions to handle the different attributes of software complexity.

The first attribute, the complexity of the domain application, is the main cause of the difficulty of capturing and specifying the requirements. Imagine the requirements of the Apollo guidance software in the 1960s [1,80], and the software systems used nowadays in air traffic control and hospital information systems. A major challenge comes from the fact that it is not realistic to expect a software engineer to understand the domain thoroughly, or a domain expert to come up with a design for the software. Solutions to this problem are in the scope of the sub-discipline of *(Software) Requirement Engineering*, which includes specification/modelling languages (together with their semantic theories), techniques and tools for requirements specification, validation and verification [69]. The aim is to master the complexity of requirements capture, definition, validation and documentation. These constitutes elements of *requirements specification and analysis* in a *software development process* [40].

The second attribute, the difficulty of managing the development process, concerns the difficulty to define and manage a development process that has to deal with complex and changing requirements and constraints. A software project typically involves a large team of software engineers and domain experts, possibly in different geographical places. The process has to identify the software technologies and tools that support collaboration of the team in working on shared software artefacts. Roughly speaking, a development process defines in the development when, who, does what work or tasks, uses what techniques and tools, and produces what artefacts. Tasks include work management and artefacts management, and techniques and tools should also support the ways of collaboration of the team. Research, education and practice of solutions to this challenge form the area of *Software Project Management* [86].

The third attribute, the flexibility possible to offer through software, is about the problem of making sound design decisions among a wide range of possibilities

[2] http://en.wikipedia.org/wiki/Wenzhou_train_collision.

that have conflicting features. This includes the design of the software architecture, and the design and reuse of software components, algorithms and communication networks. For the same requirements, different decisions lead to different software products. In particular, the decision making involves the best practice of the fundamental engineering principles of (a) *separation of concerns*, (b) *divide and conquer*, and (c) *use of abstraction* through information hiding (in different design stages). The notions of *modularity* and *interfaces* discussed at the 1968 NATO Software Engineering Conference [67], and the later developed *structured design* [84], *object-oriented design* [6], *component-based design* [25,28,61,88] and *service-oriented architecture* [5] all aim to support the practice of these three engineering principles. These three principles also apply to other software development activities including requirements analysis, verification and validation, and software project management. Model-driven architecture (MDA) [70], that recently has become a main stream approach, aims at a seamless integration of the above approaches in a unified development process, such as the Rational Unified Process (RUP) [40].

The final attribute, the problem of characterising the behaviour of software systems, pinpoints the difficulty in understanding and modelling the dynamic behaviour of software, for analysis, validation and verification for correctness, as well as reliability assurance. The dynamic behaviour of a program is defined in terms of all possible changes of states of the program. A *state* is a mapping from the program variables to their value space, representing the values that the variables take (stored in the memories allocated to the variables) at a point of the program execution. The variables include the program variables defined by the programmer and those which controls the program execution flow, such as program or process counters. A program is (functionally) correct if its dynamic behaviour conforms to its specification. For a large program with a big number of variables, especially a large scale concurrent and distributed software system, the dynamic behaviour has a great scale of complexity. This poses a great challenge for (a) writing the right requirements specification that identifies the correct state changes allowable by the application, and (b) verifying that the behaviour of the program is correct with respect to the specification. It is well-know that finding bugs in a program which may cause its behaviour to violate the program requirements is hard and costly. Seeking solutions to these challenges is the background motivation for *formal methods* of software development, that include mathematical theories of modelling and programming languages, including their syntaxes and semantics, techniques and tools for design (such as correctness preserving refinement), logical reasoning about and verification of program correctness.

2.2 Chronic Complexity of Modern Software

The characteristic attributes of software complexity still hold for modern and future software, but their extensions are becoming increasingly wider, due to the increasing power of computers, here we quote

> "The major cause of the software crisis is that the machines have become several orders of magnitude more powerful. To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."
>
> – Edsger Dijkstra
> The Humble Programmer, Communications of the ACM [20]

Now computers are everywhere and networked, executing programs anywhere and any time, which share data and communicate and collaborate with each other. New buzz-words are introduced for these different kinds of networked computing systems, *Cloud Computing*, *Internet of Thing* (IoT) [48], *Smart Cities* [83], and *Cyber-Physical Systems* (CPS) [43]. Application, control and monitoring programs [89] are being developed and integrated into these systems, which we see in our everyday life in transportation, health, banking and enterprise. These systems provide their users with a large variety of *services* and *features*. They are becoming increasingly *distributed*, *dynamic* and *mobile*. Their components are *deployed* over large networks of *heterogeneous platforms*. In addition to the complex functional structures and behaviours, modern software systems have complex aspects concerning *organisational structures* (i.e., *system topology*), *adaptability*, *interactions interoperability*, *safety*, *security*, *real-time* and *fault-tolerance*.

It is even more challenging when a networked system supports collaborative workflows involving many different kinds of stakeholders and end-users across different domains. The system is open to ever changing requirements during the development of the software and when it is in operation. Typical cases are healthcare applications, such as telemedicine, where chronic conditions of patients on homecare plans are monitored and tracked by different healthcare providers. The openness makes it much more difficult to do requirements modelling and analysis, software design, system validation and verification, and management of the development. Furthermore, it imposes challenges to *software maintenance*.

This chapter focuses on handling software complexity through component-based modelling, decomposition, refinement and verification. The next section summarises the state of the art of software engineering and motivate model-driven and component-based software engineering, and formal methods.

3 Model-Driven Software Engineering

For the discussion in this section and technical discussions in the later sections, some concepts related to software engineering were clarified.

3.1 Basic Concepts in Software Engineering

Textbooks and websites have numerous definitions for *computer software*. For examples

- a collection of *computer programs* and related *data* that provide the instructions for telling a *computer* what to do and how to do it.
- a set of *computer programs* or *procedures*, and associated documentation concerned with the operation of a *data processing system*.
- one or more *computer programs* and *data* held in the storage of the *computer* for some purposes.

For developing a systematic understanding, we first give a rather abstract description of the related concepts of *computation*, *computer*, a *data process system*, and *programs* as follows:

1. a *computer* carries out a *computation* by executing a *program*;
2. a *program* defines of a number of variables and a sequence of *commands*;
3. when a computer executes the program, the variables are allocated in memories (storage) of the computer to hold data values at any moment of time, called the *state* of the program execution at that moment of time;
4. at the beginning of the execution, the computer receives *input values* for the variables (to set up the *initial state* of the execution);
5. during the execution, the commands are carried out according to the flow of control defined by the sequence of commands and the execution of each command changes the current state to the next state defined by the *semantics* of the command; and
6. at the end of the execution (if the execution terminates), an *output* is produced that is determined by the final state of the program in the execution.

Note that the same program can be executed repeatedly with different inputs (i.e., initial states) to generate different outputs (i.e., final states). The fifth and sixth statements above imply that the semantics of a program command, thus that of a whole program can be mathematically defined as a *relation between program states* (see later in this Subsection for further clarification). This relational semantic model is the theoretical foundation of the method we study in this chapter.

The above discussion on computation, computers and programs is easy to comprehend when we think of sequential programs running on uni-processor computers. For example, given any initial value x_0 and y_0 to variable x and y, i.e., from the initial state $s_0 = \{(x, y) \mapsto (x_0, y_0)\}$, the execution of command (program) $x := x + y + 1$ changes the value of x from x_0 to the value $x_0 + y_0 + 1$, i.e., the execution changes from the initial state s_0 to final state $s_1 = \{(x, y) \mapsto (x_0 + y_0 + 1, y_0)\}$.

The semantics of the program, denoted by $[\![x := x + 1]\!]$ is defined to be the relation $\{(s_0, s_1) \mid x_0 \in T_x \wedge y_0 \in T_x\}$, where T_x and T_y is the value spaces of x and y, say the set of integers. The semantics of the composite commands can be defined using operations on relations, and recursive or iterative commands by fixed points of recursive equations in relational algebra [32]. The following examples give the flavour of the calculation of the semantics of composite commands:

$$
\begin{aligned}
&[\![x := x + y + 1; y := x]\!] \\
&= [\![x := x + y + 1]\!]; [\![y := x]\!] \\
&= \{((x, y) \mapsto (x_0, y_0), (x, y) \mapsto (x_0 + y_0 + 1, y_0)) \mid x_0 \in T_x \wedge y_0 \in T_y\}; \\
&\quad \{((x, y) \mapsto (x_1, y_1), (x, y) \mapsto (x_1, y_1)) \mid x_1 \in T_x \wedge y_1 \in T_y\} \\
&= \{((x, y) \mapsto (x_0, y_0), (x, y) \mapsto (x_0 + y_0 + 1, x_0 + y_0 + 1)) \mid x_0 \in T_x \wedge y_0 \in T_y\}
\end{aligned}
$$

where in the above formulas we overloaded ";" for both the sequential composition of program commands and for the composition of relations in relational algebra. Another example illustrates the conditional command, where $C_1 \lhd B \rhd C_2$ denotes the conditional choice **if** B **then** C_1 **else** C_2:

$$
\begin{aligned}
&[\![x := x + y + 1 \lhd x < y \rhd y := x + y + 1]\!] \\
&= [\![x := x + y + 1]\!] \cap [\![x < y]\!] \cup [\![\neg(x < y)]\!] \cap [\![y := x + y + 1]\!] \\
&= \{((x, y) \mapsto (x_0, y_0), (x, y) \mapsto (x_0 + y_0 + 1, y_0)) \mid x_0 \in T_x \wedge y_0 \in T_y \wedge x_0 < y_0\} \\
&\quad \{((x, y) \mapsto (x_0, y_0), (x, y) \mapsto (x_0, x_0 + y_0 + 1)) \mid x_0 \in T_x \wedge y_0 \in T_y \wedge y_0 \leq x_0\}
\end{aligned}
$$

where $[\![B]\!] = \{(s, s') \mid B \text{ holds in } s\}$, thus $x < y$ hold for state $(x, y) \mapsto (x_0, y_0)$ if $x_0 < y_0$. The semantics of an iterative command **while** B **do** C, algebraically denoted as $B * C$, is defined to be the "smallest" solution [32,68,87] to the recursive equation $[\![B * C]\!] = [\![C; B * C]\!] \lhd B \rhd$ **skip**, where **skip** is the identity relation that define the semantics of the program whose execution does not change the state.

The above discussions on computation, computers and programs can be generalised to models of modern computer systems, including networks of data processing systems (cf. the second definition of computer software), and other programming paradigms such as object-oriented programs, concurrent and distributed programs [32].

An extension of the notion of "computer software" is *software systems* which are a "collection of programs" and the associated "data", which are interrelated in an architecture and the 'work' together when being running on a computer. Here, "computer" refers to any device or system with the power of processing and transmitting data. We thus define a **software system** to consist of set of architected programs and data that tell a set interrelated computers what to do and how to it. Computers include all devices with programmable processing capacity, all kinds of "smart devices" as well as "computers", that now affects all aspects of daily life.

We take the general view that **software engineering** is about the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software systems, and the study of these approaches. This is to say software engineering is about the application of *engineering* to manufacture of software systems, and software engineering has a significant foundation in mathematics, computer science and has practice that have strong origins in engineering. In more concrete terms software engineering is about development, study and application of *theories, techniques and tools* for requirements analysis, design, implementation, correctness validation and verification (including testing and simulation), and maintenance of software systems.

A **software engineering method** consists of a theory, a set of techniques developed based on the theory, and a suit of tools that support systematic applications of the techniques to requirements analysis, software design, implementation, validation and verification, and maintenance in software systems development. The systematic application of a method to software development relies on well-defined *software development processes*. Examples of software engineering methods include *structured software development*, *object-oriented design*, *component-based design*, and *model-driven development* (or *MDA - model-driven architecture*), that have overlapping theories, techniques and tools support. Software engineering has to help in mastering software complexity. Therefore, the theory, techniques and tools have to support effective handling of software complexity. This means they need to define mechanisms of separation of concerns, divide and conquer and information hiding for abstraction. A model-driven software engineering method represents the state of the art of software engineering methods with regard to these aspects.

3.2 Model-Driven Development

All well established engineering branches rely on the use of *models* to represent different *viewpoints* and *concerns* of the *artefacts* constructed at different stages of the engineering process. All models are representations of the views with details that are not relevant to the present concern being excluded. Model-driven software engineering methods [70,88,89], propose the same approach to engineering software systems. That is, a software system is manufactured through building system models in all stages of the development. A particular model-driven method is called *Model-Driven Architecture* (MDA), launched as a standard model-driven software engineering method by Object Management Group[3].

Example. Consider an application case in the context of smart cities. One can imagine a street lightening system has different stakeholders, each having different views and concerns. The city council is concerned about the conveniences of the citizens when walking in the night; the police office on the other hand has an interest in the relation of the street lightning with crimes; and further the electricity company is concerned about power consumption as readings on meters and bills. These different views are represented as models of the requirements, the interfaces of services to these stakeholders, and the program implementations of the services. These different models are at different levels of abstraction, but they are closely related and can all be built based on a common model of the configurations and dynamic behaviour of the lights.

MDA supports the principles of divide and conquer with its component-based architectures, in which an architectural component is hierarchical and can be divided into subcomponents. The architectural components inside a (composite) component interact and communicate with each other through their interfaces, according to explicitly specified *contracts*, and work together to realise the interface contract of the whole composite component. For example, a buffer of

[3] http://www.omg.org.

capacity one has an interface for a user to "put" a data item and for another user to "get" the data item stored in the buffer. Then a two-place buffer can be composed from two one-place buffers by employing a connector. The user puts a data item using the 'put' of one-place buffer and get a data item out using the "get" of the other. The connector moves the data item out of the first one-place buffer using its "get" and puts it into the second one-place buffer using its "put". This is shown in Fig. 1.

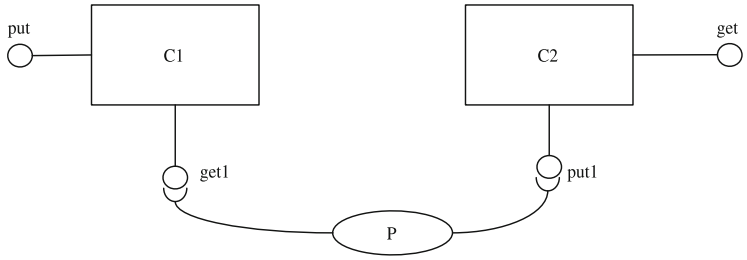


Fig. 1. Composite component

MDA supports *separation of concerns* by providing notations for representing different viewpoints of a component. These include the static component and class views using UML component and class diagrams, interaction views by UML sequence diagrams, and dynamic behavioural views by UML state machine diagrams. The models of different views of architectural models are important when defining and managing a development process [53]. However, a serious consistency problem arises in both of its theoretical foundation and its practical application in software system development, caused by the large number of UML models that are constructed in different notations possibly by different members of the project team.

The consistency problem is mainly due to first the lack of clear defined development process that defines the models to be sued and their relations, and secondly the omission of a unified semantics of these models for the project. Therefore, integration and transformation of models are mostly syntax-based, having no provable semantic correctness. Hence, the tools developed to support model integration and transformation cannot immediately be integrated with tools for verification of semantic correctness and correctness preserving transformations [60]. For MDA to support a seamless development process of model decomposition, integration, and transformation, there is a need of formal semantic relations between models of different viewpoints at the same level of abstraction, and between models of the same viewpoint at different levels of abstraction. The relations for the former are to deal with consistency among models of different views for correct integration [12,51,52], and the relations for the models at different levels are the refinement/abstraction relations [15,27,52,91]. We do not promote a single unified semantics for UML, but a unified semantics of the models used in a project should be defined.

Most MDA techniques and tools focus on transformations between models of the same view, such as class models or state-based behaviour, but built with different notations and tools. Also, there are plenty of techniques and tools for transformations between PSMs. There is, however, very little support for transformations between PIMs at different levels of abstraction, except for some design patterns directed model transformations [63]. This is actually the reason why MDA has yet to convincingly demonstrate its potential advantages of *separation of concerns*, *divide and conquer* and *incremental development*. This lack of semantic relations between models as well as the lack of techniques and tools for semantics-preserving model transformations is also an essential barrier for MDA to realise its full potential in improving safety and predictability of software systems. The research in rCOS and development of its tool support focus on filling these gaps [4,47,90,91].

3.3 Formal Methods of Software Development

The development of formal methods is a step towards placing software and hardware development on a sound engineering discipline so that appropriate mathematical analysis is possible [7]. A formal method is about the uses of a broad range of theoretical computer science fundamentals to solve problems in software and hardware specification and verification. These fundamentals include logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types. We say a *formal method of software engineering* consists of a semantics theory, a body of techniques and a suite of tools for the *specification*, *development*, and *verification* of software systems of a certain programming paradigm, such as procedural sequential programming or object-oriented programming, concurrent and distributed programming and component (or service) based programming. The semantic theory of a formal method is developed based on the fundamental theories of *denotational semantics* [87], *operational semantics* [75], or *axiomatic semantics* (including algebraic semantics) [19,30] of programming. As they are all used to define and reason about behaviour of programs, they are closely related [68], and indeed, they can be "unified" [27,32].

A specification is an abstract model of the program or the specification of desirable properties of the program in a formally defined notation. In the former case, the specification notation is also called a modelling language though a modelling language usually includes graphic notations (e.g., Petri Nets [74]). There are now a large number of well-known formal modelling/specification languages, including CSP [31], CCS [66], the Z-Notation [85], the B-Method [2,3], VDM [36], UNITY [10], and TLA+ [41]. In the latter case, desirable properties are defined on a computational model of the executions of the system and specified in a formal logic. Well-known examples include the *labelled transition systems* and the *linear temporal logic* (LTL) of Manna and Pnueli [65] and the

branching temporal logic (or CTL), which are also used in verification tools like Spin [33] and Uppaal.[4]

In the past half a century or so, a rich body of formal theories and techniques have been developed. They have made significant contributions to understanding the behaviour of programs. Recently there has been a growing effort in development of tool support for verification and reasoning. However, these techniques and tools, each of which each has its own community of researchers, have been mostly focusing on models of individual viewpoints. For examples, type systems are used for data structures, Hoare Logic for local and static functionality, process calculi (e.g., CSP and CSS) and I/O automata [64] for interaction and synchronisation protocols. While process calculi and I/O automata are similar from the perspective of describing the interaction behaviour of concurrent and distributed components. The former is based on the observation of the global behaviour of interaction sequences, and the latter on the observation of local state transitions caused by interaction events. Processes calculi emphasise algebraic reasoning, and automata are primarily used for algorithmic verification techniques model checking [18,77].

The impact of these theories, techniques and tools on the improvement of qualities of the daily used software systems is yet to become convincing to software engineering practitioners for their industry adoption. The gap between the development of formal methods and the advances in software technologies is not becoming narrower. More precisely, the relation between formal methods and software technologies is not well understood yet. This is clearly reflected by the challenges in engineering current large-scale systems, including CPS, IoT and cloud-based systems, that have multi-dimensional complexities. The experience, e.g., in [34], and investigation reports on software failures, such as those of the Therac-25 Accident in 1985–1987 [44] and the Ariane-5 Explosion in 1996 [81], show that a simple mistake can lead to catastrophic consequences. *Ad hoc* application of the above methods to specification and verification of programs will not be enough or feasible to detect and remove these causes. Different formal techniques that deal with different concerns more effectively have to be systematically and consistently used in all stages of a development process, along with safety analysis that identifies the risks, vulnerabilities, and the consequences of the risk incidents. There are applications that have concerns of extra functionalities, such as real-time and fault-tolerance constraints [57]. Studies show that models with these extra functionalities can be obtained and treated by model transformations of models without these concerns [22,56,57]. But this is yet to be better understood by software engineering practitioners.

4 Case Study: A Trading System

We will use this example to demonstrate the attributes of software complexity, motivate the problems in our discussion, as well as to understand the fundamental concepts and techniques of the model driven method.

[4] http://www.uppaal.org.

This case study describes a *Trading System*. It was used as the Common Component Modelling Example (CoCoME) in a comparative modelling exercise using different methods. The problem description and the solutions developed by the participants were presented at the two GI-Dagstuhl Research Seminars [79]. A team from UNU-IIST led by the first author used the rCOS method in the exercise [13]. The case study is an extension of the Point of Sale System (POST) described in the textbook of Larman [42].

This version of the Trading System focuses on the functionalities in terms of *use cases*, related to processing sales in a supermarket. This restriction obviously is related to the limited space. However, it also reflects a fundamental engineering principle that MDA, object-oriented design and component-based design provide effective support [6,27,91] for dealing with complexity. That is,

> *'start with a small and simple system and get it work; then let the system evolve by adding new features and/or functionalities'.*

The description of this case study plays the role of a client's requirements document as if it were provided to the software development team by a business company in the reality. Therefore, the description is potentially imprecise, incomplete and even inconsistent, as it has to go through the requirements analysis by the developer. The Trading System is used in handling sales in a supermarket including the processes at a Cash Desk (or a Point of Sale). For example

1. a Casher *scans* or *types in* the Products being purchased using a Bar Code Scanner or a Keyboard, then the Customer *pays* for the Sale by a Credit Card, or
2. by Cash and receive the Change.

Notice the italic verbs and capitalised nouns, which in later stages of analysis and specification might be formalised as "operations" and "objects" (or "classes"), respectively. These processes of a business task, e.g., handling sales, will be defined as *use case scenarios*, and all the use case scenarios of a task form an abstract *use case*. The Trading System is also used for various administrative tasks, such as reordering various products or generating reports. The following subsection gives a brief overview of the Trading System and its hardware parts. The required use cases will be presented in the subsections to follow. The description is based on Chap. 1 [29] and the rCOS solution [13] of the final workshop proceeding [79].

4.1 The Hardware Components

Software components run on or controls hardware components and their interactions. The Cash Desk is the place where a Sales Assistant/Cashier scans the product items which a customer wants to purchase and where the payments (either by credit card or cash) are carried out. Furthermore it is possible to switch to an express checkout mode which allows only costumers with a few

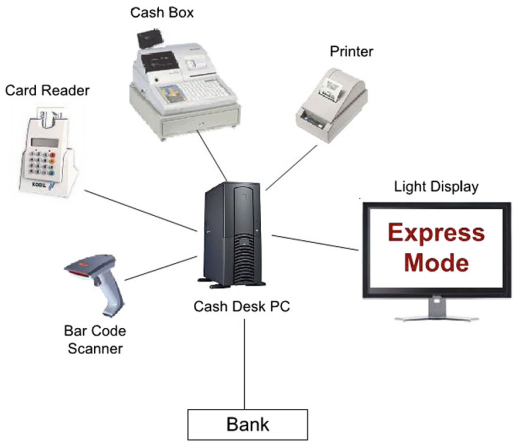


Fig. 2. Hardware components of a single Cash Desk.

items and cash payments to speed up the checkout process. To manage the processes at a Cash Desk a lot of hardware devices are necessary (See Fig. 2).

Using the Cash Box, which is available at each cash desk, a sale can be processed from the start to the end, *through the interactions between the Cashier (a human actor) and the customer (a human actor), and between the Casher and the system under design.* The cash payment process needs to involve the cash box. To handle payments by credit card, a Card Reader is used. In order to identify all product items the Customer are purchasing the Cashier uses the Bar Code Scanner. At the end of the process a receipt is produced using a Printer. Each cash desk is also equipped with a Display to let Customer know if this cash desk is in the express checkout mode or not. The central unit of each cash desk is the Cash Desk PC which interfaces to all the hardware components. This PC also runs the software which is responsible for handling the sale process as well as communication with the Bank for credit payment authorisation.

A single Cash Desk might be enough for the management of a small retail shop. In general, a larger Store has several cash desks in a Cash Desk Line. The cash desk line is connected to a Store Server which itself is also connected to a Store Client as shown in Fig. 3. The store client can be used by the manager of the store to view reports, order products or to change prices of goods. A Store Server is also needed to hold the inventory of the corresponding store.

The Trading System can have even more components. Consider an enterprise with a chain of supermarkets/stores. Then, the cash desk lines of the stores can be connected to a server, called Enterprise Server. With the assistance of an Enterprise Client the Enterprise Manager is able to generate several kinds of reports. The enterprise system of a Store Chain is shown in Fig. 4.

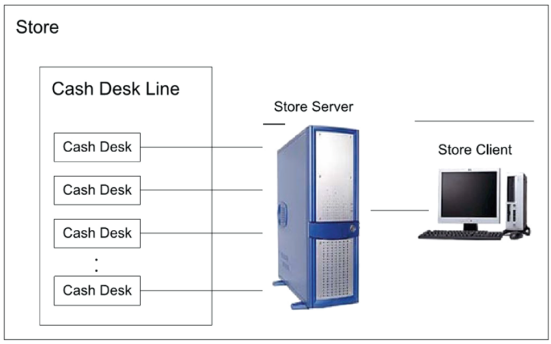


Fig. 3. An overview of a Cash Desk Line in a store.

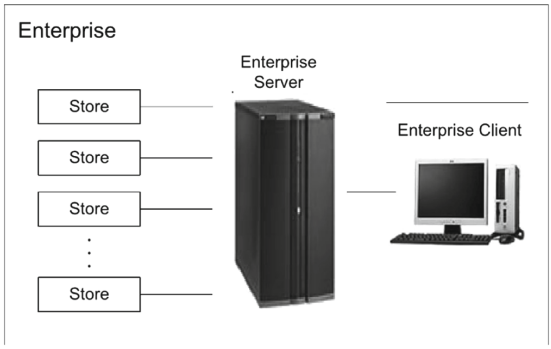


Fig. 4. The Enterprise of a Store Chain with an Enterprise Server and an Enterprise Client.

4.2 Functional Requirements and Use Cases

We describe the functional requirements using the very important notion of *use cases*, though their precise definitions and systematic discussions will be left to Sect. 6.1 in Part II. To show a moderate scale of complexity of the Trading System, we introduce five use cases[5] though we will not present the analysis and design of all of them. In what follows, we present brief and informal descriptions of these use cases.

UC 1 - Process Sale

Overview: A customer arrives at the Cash Desk with the product items to purchase. The payment - either by credit card or cash - is performed. Involved Actors includes Customer, Cashier, and Bank[6].

[5] CoCoME [29] has eight use cases.

[6] In the CoCoME problem description [29], the hardware devices, such as Printer, Card Reader, Cash Box, Bar Code Scanner, and Light Display, are also modelled as actors that the software has to interact with. In our approach, they are separated from the application logic and treated as a separate embedded system.

Process: The normal courses of interactions between the actors and the system are described as follows.

1. When a Customer comes to the Cash Desk with her items, the Cashier initiates a new sale.
2. The Cashier enters each item, either by scanning in the bar code or by some other means; if there is more than one of the same item, the Cashier can enter the quantity. The system records each item and its quantity and calculates the subtotal. When the Cash Desk is operating in express mode, only a predefined maximum number of items can be entered.
3. When there are no more items, the Cashier indicates the end of entry. The total of the sale is calculated. The Cashier tells the Customer the total and asks her to pay.
4. The Customer can pay by cash, check, or credit card. If by cash or check, the amount received is entered. The system records the cash payment amount or the check and calculates the change; operating the Cash Box to put cash or the check in and take cash out. If by a credit card, the card information is entered. The system sends the information to the Bank for authorisation. The payment succeeds only if a positive validation reply is received. In express mode, only cash payment is allowed. After the payment is made, the inventory of the store is updated and the completed sale is logged.

Alternative Courses of Events: There are exceptional or alternative courses of interactions, e.g., if the entered bar code is not known in the system, the Customer does not have enough money for a cash payment, or the authorisation reply is negative. A system needs to provide means of handling these exceptional cases, such as cancel the sale or change to another way of paying for the sale.

In many books and papers, e.g., the chapter [29], use case descriptions include *preconditions* and *postconditions*. However, no clear definitions are given to preconditions and postconditions, or the definitions are confusing, specially sometimes the preconditions are confused with the notion of "guards". Guards are conditions for control flow and synchronisation. In our method, we do not have preconditions and postconditions for use cases. Instead, we introduce notions of preconditions, postconditions and guards of *use case operations* in Sect. 8 of Part II. Examples of use case operations include "initiate new sale", "enter item", "make cash payment". We will give details on use case operations in Sect. 8 of Part II.

UC 2 - Manage Express Checkout Mode

Overview: The Cashier should be able to switch the checkout mode between "express mode" and "normal mode" by pressing a button at his Cash Desk. Involved Actors includes Cashier only.

Process: The normal courses of interactions between the actors and the system are informally described as follows.

1. Depending on the current mode of the check out, the Cashier switches the mode to "express" if it is currently "normal".

Notice this is a one-step process. Unlike the description given in [29] where automatic control of checking model is considered, we leave the decision of changing mode to the human actor, Cashier. To automatically control this operation, the human operator would be replaced by a digital operator triggered by a certain condition. In this chapter, automatic control of hardware devices is not considered.

UC 3 - Order Products

Overview: For the purpose of inventory control in a store, products should be ordered when the stock becomes low (determined by a threshold value). There are two actors involved: Store Manager and Product Suppliers.
Process: The normal process of interactions of Store Manager and the Trading System is as follows.

1. The Store Manager makes an Order (of Products).
2. The Store Manager enters the identities of the products, their amounts and the identities of the corresponding suppliers to the system, one product at a time.
3. The Order is saved in the system and the order is sent to Suppliers.

UC 4 - Receive Delivery of Ordered Products

Overview: When a delivery of ordered products arrives at the Store, the Store Manager checks the correctness and completeness (compared to the order), and then the inventory is updated. The Store Manager is the involved actor.

Process: The normal course of interactions is:

1. A Delivery of ordered products arrives at the Store.
2. The Store Manager checks the delivery against the order for correctness and completeness.
3. The Store Manager updates the inventory of all the received products.

Alternative Courses of Events:

- Step 1: if the delivery is not correct (either the delivery contains products not ordered, or larger amounts of some products delivered than ordered), exception handling is needed.
- Step 2: if the delivery is not complete (either there are omissions of products ordered or not enough amounts of some ordered products have delivered), exception handling is needed.

UC 5 - Product Exchange Among Stores. Consider an enterprise of a chain of stores. If a store runs out of certain products and these products are not available from their suppliers, it is possible for this store to ask the enterprise management to check whether those products are available in some other stores. If there is such a store, called a Providing Store, it will ship the requested products to the store that asked for them. Notice that this use case involves interactions among Requesting Server (i.e., the server of the requesting store), Enterprise Server and Providing Server (i.e., the server of the providing store).

Overview: The Requesting Store Manager makes a query (similar to a Product Order) at her Store Server. This Store Server sends the query to the Enterprise Sever. The Enterprise Server then looks for a Providing Store, through interaction with the other servers at the stores of the enterprise. Once a Providing Store is found, the product query is passed on to the Providing Store Server. The Providing Store Server generates a delivery according to the product query and send to the Requesting Store Server. The Providing Store ships the products to the Requesting Store.

Process: The normal course of interactions is described as follows.

1. The Requesting Store Manager makes a query.
2. The Requesting Store Manager enters the identities and amounts, one at a time, in the query.
3. After entering the last item, the Requesting Store Manager indicates the Requesting Store Server to save the query and sends it to Enterprise Server.
4. Enterprise Server looks for a Providing Store through interactions with the other Store Servers.
5. The Providing Store Server generates a delivery according to the query.
6. The Providing Store Server sends the delivery to the Requesting Store Server.

Alternative course of interactions:

- Step 2. If product identity is not known, raise exception handling.
- Step 3. If Enterprise Server is not available (in terms of communication and system failures), exception handling (keep trying for example), raise exception handling.
- Step 4. If no providing stores is available, exception handling, including communication failures in finding a providing store, raise exception handling.
- Step 5. The delivery fails to be received by the Requesting Store Server (communication error), raise exception handling.

Remarks

1. The enterprise server needs to realise a distributed algorithm to find a providing store. The algorithm should be efficient in terms of space and time complexity, and implements a strategy that this economically optimal.
2. There is a use case for each store server to be designed and implemented to receive the shipment of the goods against the delivery received from the providing store server.

3. The requirements description does not have to start with such a use case involving interactions between a number of subsystems. We could instead write use cases Prepare Query, Look for Providing Store, and Prepare for Delivery. Then we analyse and design them in separation. We can then compose (or integrate) them together with middleware.

In the CoCoME functional requirements description [29], three more use cases are introduced. They are *Show Stock Reports*, *Show Delivery Reports* and *Change Price*. These use cases are quite simple by themselves. However, they introduce significant complexity to system integration. We also notice the descriptions of the use cases involve hardware devices, such as printers, lights and cash boxes as actors. However, in our approach we treat these hardware devices as an embedded system separated from the application logic. They can be represented as variables to be changed by the use case operations, such as "switch light" and "print sale".

The discussion of use cases in this section is rather ad hoc. Thus, it is impossible to carry out systematic analysis and design. In the sections to follow, we will introduce a definition of use cases, their presentations, and compositions for requirements analysis.

5 Software Engineering Process in Brief

The model-driven approach we present combines object-oriented design and component-based design. Although its principles and techniques of modelling, analysis and design can be used in general software processes, this approach supports especially seamlessly the use case driven Rational Unified Process [40].

5.1 Software Development Processes

Recall the view of software engineering introduced in Sect. 2 as being concerned with the theories, methods and tools which are needed to develop software systems. Its aim is the production of dependable software, delivered on time and within budget, that meets the user's requirements. For all life cycles of the software, that is reliable installation, operation and maintenance as well for the development cycles, software development not only produces a working software system, but also documents such as those of the requirements specification, system design, user manual, and so on. For the development of a software system of a certain scale of complexity, there is a need for an *engineering process* which allows techniques and tools of software methods to be used effectively and systematically (See Sect. 2 about the 2nd attribute of software complexity).

All engineering is about how to produce products in a disciplined process. In general, an *engineering process* defines who is involved in the process, which products is being produced, what and when activities of the process happens, and which and when techniques and tools are being used for the production of what artefacts. A process to build a software system and its documents or to

enhance an existing one is called a **software development process**. A software development process is thus often described in terms of a set of activities needed to transform the user's (or client's) requirements into a software system. At the highest level of abstraction, a development process is in general an iterative of activities that can be depicted in Fig. 5.

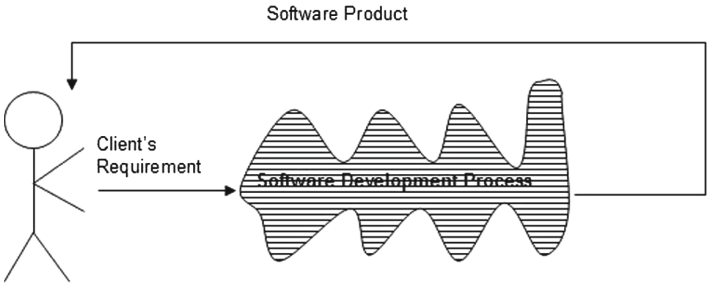


Fig. 5. Software development process

The *client's requirements* define the goal of the software development. They are prepared by the client (sometimes with the help from a software engineer) to set out the services that the system is expected to provide, e.g., the functional requirements of the Trading System described by the sue cases in Sect. 4. Apart from functional requirements, a client may also have non-functional constraints they would like to place on the system, such as the required response time or the use of a specific language standard. In this chapter, we are mainly concerned with functional requirements.

We must bear in mind about the following facts, as demonstrated in the description of the Trading System, which make the requirements capture and analysis very difficult:

- The client's requirements are often incomplete.
- The client's requirements are usually described in terms of concepts, objects and terminology that may not be directly understandable to software engineers (cf. first attribute of software complexity discussed in Sect. 2).
- The client's requirements are usually unstructured and they are not supposed to be rigorous, without redundancy, vagueness, and inconsistency.
- The client's requirements may not be feasible, as one cannot expect a client to know as well as software engineers, about theories of computability and computational complexity, and about state of the art of computer technologies.

Therefore, any development process must start with the activities of capturing and analysing (part of) the system requirements based on the client's requirements. These activities and the associated results form the first phase (or subprocess) of the process called *requirements gathering and analysis*.

5.2 Requirements Gathering and Analysis

This phase is to develop a good understanding of the application domain and capture the right requirements for the system to design. It is the first step aiming the development towards an adequate system[7]. The goal is to produce the artefact called the *requirements specification*. The whole scope of requirements capture and analysis forms *requirements engineering*. Here, we briefly discuss the main activities needed and the essential attributes of artefacts produced by the activities. We study object-oriented and component-based techniques for requirements capture and analysis in Sects. 6.1–8 in Part II.

First of all, the requirements specification will be used as

1. a fairly full model of the requirements for the system to design;
2. a contract agreed between the client and the system development organisation, also called the *project developer*;
3. a basis for requirements validation including prototyping, simulation, reasoning (for correctness and completeness), and verification of desired properties of the specification;
4. a basis for design and system verification:
   - test cases should be made against the specification, and
   - test cases should be designed to cover all the crucial services required;
5. a basis for system evolution.

To produce a requirements specification with the above attributes, requirements analysts need notations, techniques and tool support to carry out the following highly iterative interrelated activities, involving discussions and collaborations with the client, the application domain experts, and the potential system users.

- **Domain understanding.** The analysts must develop their understanding of the application domain. Therefore, the concepts are explored and the clients' requirements are elicited.
- **Requirements capturing.** The analyst must have a way of capturing the clients' needs so that they can be clearly communicated to everyone involved in the project. Skills of abstraction are important to capture the essence and ignore the accidental details.
- **Classification.** This activity takes the unstructured collection of requirements captured in the earlier phases and organises them into coherent clusters, and then prioritises the requirements according to their importance to the client and the users.
- **Validation.** This is to check if the requirements are consistent and complete, and to resolve conflicts between requirements.
- **Feasibility study.** This is to estimate whether the identified requirements may be satisfied using the software and hardware technologies, and to decide if the proposed system will be cost effective.

[7] There can be a number of adequate systems, as discussed in Sect. 2 about the 3rd attribute of software complexity.

There are no rigid rules on when requirements analysis is completed and the development process proceeds into the next phase. It is helpful, however, to ask the following questions before the development progresses moving into the next phase:

- Has the system requirements been understood by the clients, end-users, and the developers?
- Has a fairly complete model of the requirements been built? This model specifies what the system must do in terms of
  - available functions (or services),
  - inputs and outputs, and
  - necessary data.
- Are the functions and data correct and complete, and how are they related? To check this, fast prototyping for validation is often used.

Precise and systematic statements of these questions and development of their answers require notations for requirements description and techniques and tools for their analysis. In Sects. 6.1–8 of Part II, we will introduce these notations and techniques, and demonstrate how they are used in the requirements analysis of the Trading Systems. They are largely based on the notion of use cases and structures of conceptual classes, the UML notations and tool support. Rigorous requirements specification and analysis (i.e., mathematical based analysis) fall into the scope of *formal requirements specification and analysis*. In summary, we make two remarks on requirements analysis:

- The requirements specification is the official statement of what is required of the system to be developed. It is not a design document and it must state what to be done rather than how it is done. It must be in a form which can be taken as the starting point for the software development. For this, specification languages, including graphic notations, are often used to describe the requirements specification.
- The requirements capture and analysis phase is important, as an error which is not discovered at the requirements analysis phase is likely to remain undiscovered, and the later it is discovered, the more difficult and more expensive is it to fix.

5.3 System Design

After the requirements specification is produced through requirements analysis[8], it undergoes iterative cycles of *architectural design* and *detailed design*. In the architecture design of a cycle, part of the requirements are partitioned into interconnected *components* which are specified in terms of their *interfaces* and the *contracts of the interfaces*. This results in an *architectural design document*. Then each component undergoes its *detailed design*, to decide how each component does what it is required to do by the contracts of its interfaces. The

[8] It is not necessarily for the overall requirements analysis to be completed before design activities can start.

architectural design is in general an abstract and implementation independent high level platform independent (PIM), defining the functionality and interface of each component. The detailed design is a low level PIM and it is desirable that an implementation can be generated once the programming language and system platform are given.

In Sect. 10.1 of Part III, we show how the architecture design is derived from the use case analysis and decomposition. The artefact produced there is the architectural design document called an *architectural model*, consisting of some component-based diagrams and the models of component interface contracts. In Sect. 11 of Part III, each component is designed using five design principles of responsibility assignments to objects [42], resulting in a detailed design model. The model of the system at this level is a low level PIM and it can be seen as a template program modules.

Correctness of Detailed Design. Verification of the design of the architectural components should be in general done against the specification of the components in the system architectural model. There are few effective techniques and tools for this correctness assurance if the model of the architecture and the models of the detailed designed components are informal. In formal methods, the verification of a low level design model can to some extend be done by logic reasoning, and automatically checking the low level model satisfies specified properties of the high level model using *model checking* techniques and tools.

Implementation and Unit Testing. After the design of the system architecture and the (detailed) design of the architectural components, each of the designed components is realised as a program unit. Each unit then must be either tested against its specification obtained in the design stage - *unit testing*, or formally verified using techniques and tools of automatic *static analysis*, *dynamic analysis*, and model checking.

System Integration and System Testing. The individual program units representing the components of the system are combined and tested as a whole to ensure that the software requirements have been met. When the developers are satisfied with the product, it is then go through acceptance testing. This phase ends when the product is accepted by the client. System testing plan must be made and test cases must be designed according to the system requirements specification.

Operation and Maintenance. This phase starts with the system being installed for practical use, after the product is delivered to the client. It lasts till the beginning of system's retirement phase, which we are not concerned in this chapter. Maintenance includes all changes to the product once the client has agreed that it satisfied the specification document. Traditionally, corrective maintenance (or software repair) and enhancement (or software update) are the

main concerns. Corrective maintenance involves correcting errors which were not discovered in earlier stages of the development process while leaving the specification unchanged.

Modern software systems, however, are ever evolving, as their operation environments are constantly changing. Evolutionary changes include legacy components being upgraded or removed, and new components being integrated. These different components are often implemented and deployed on different platforms, and are interacting via different communication technologies and networks. Therefore, the maintenance process of modern systems now become different cycles of requirements analysis, design, implementation and integration. The method that we will introduce in later sections is characterised as use case driven and component-based design, with interfaces as a first concepts. This method provides effective support to the development of modern evolutionary software systems.

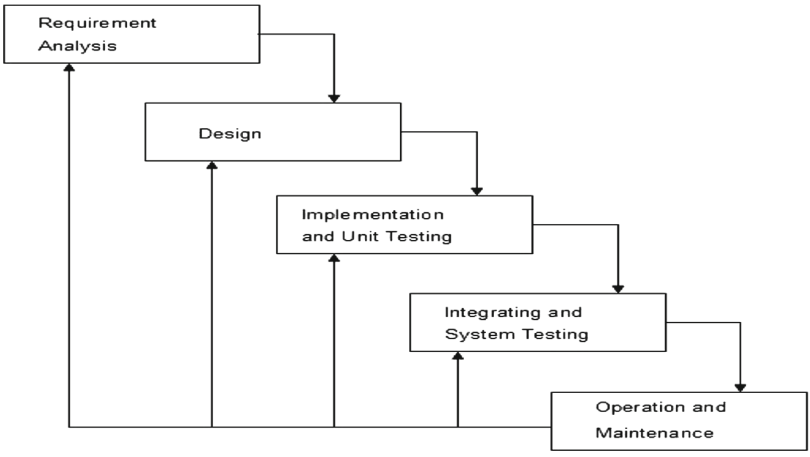


Fig. 6. Software development process

Waterfall Model of Development Process. According to the software development activities discussed in the earlier subsections, a development process is often organised into the so-called "waterfall model" depicted in Fig. 6. However, variants and extensions of the waterfall model are used in practical software system development, such that

– The stages in the waterfall model overlap and feed information to each other: during design, problems in requirements are identified; during coding, design problems are found and so on. Therefore, the development process is not a simple linear model but involves a sequence of iterations of the development activities.

– Sometimes it is quite difficult to partition the development activities in a project into these distinct stages.
– The waterfall model is often extended with validation and/verification activities/phases which are often inserted between two consecutive steps of the water fall model, and this extension model is called the *V-model* of development process [76].

The waterfall model can be used for some statistic studies about the software development. It mainly serves us for organisation and terminology of the discussions of the development activities and models of artefacts.

Our approach, being as the background motivation for and the informed practical approach by the development of the rCOS formal methods, supports the principle of *correct by construction* and improved assurance by validation and verification. Also, the construction identifies the obligations of validation and verification. In this chapter, we focus on model construction without the space for enough model validation and verification.

6 Part II: Requirements Modelling

6.1 Requirements Gathering and Analysis - Use Cases

From the perspective of model-driven development, software engineering transforms a real world application model to a model in the digital world that satisfies the application requirements. The first step is thus to build and analyse an adequate abstract model of the real world of the application. The application domain is usually described in terms of domain processes, also called business processes. These processes carry out operations on and involve interactions among objects so that objects are created, recorded, destroyed, changed and transmitted.

The operations and processes are carried out in an organisation or a structure. We will consider the structure at two levels of abstraction. The structure at the higher level of abstraction is formed by the processes and their relations, and this structure supports the interactions among the processes. The structure at the lower level supports the interaction among the objects involved in the execution of operations. The structures for processes interactions are component-based; and the structures for object interactions are object-oriented, are formed by objects and their relations.

We study an approach to build models for analysis, design and verification with these two kinds of structures and their relations. Thus, our approach is a combination of object-oriented and component-based modelling and analysis, driven by the domain processes. The main ideas are to model domain processes by use cases, and real world concepts and objects by classes and with relations between classes as associations. The classes and their associations will form a conceptual class model of the domain. The structure of the domain processes is represented by a component-based model of the architecture. The main points discussed in this section are: *use case descriptions* of domain processes, and *use case diagrams* for representing relations between use cases as well as *actors* and between use cases.

6.2 Use Cases

A key point in *object-oriented analysis* is decomposition by classification of the domain objects and their relations. An important technique to help in identifying and understanding domain concepts and objects and their relations is to inspect the domain processes. We use a *use case* for a narrative description of a domain process in terms of interactions between the users and the system. Domain processes that are required to be automated by the software must be done no matter which software requirements engineering approach is used in a software project. For example, the concept of use cases is similar to *viewpoints* in structured software requirements analysis.

Roughly speaking, a **use case** is a story board that tells how users carry out a business process (or a task). Here a 'system' does not have to be a digital system. It can be, for example, a paper based manual system. A 'user' does not have to be a human actor either. It can be, for example, a device or a digital system. More precisely, a **use case** describes possible *sequences of interactions* by *some types of users* using *some of the system functionality* to complete a process. Such a type of user is called an **actor**. An **interaction** is an happening of an *operation* by a user on the system or a *message* from the system to an actor.

A given user may have several roles and thus be different actors when interacting with the system. For example, a Store Server can act as a Requesting Store Server or as a Providing Store Server. On the other hand, several individual users may act as different instances of the same actor. For example, there can be many individual Cashiers. Therefore an actor represents a coherent role in using the system, rather than representing a particular individual or entity. In other words, an actor represents a type of users of the system. Actors are external to but interacting with the system. A use case is always initiated by some possible **initiator actor**. The actors that directly interact with the system are **direct actors**.

An actor interacts with the system by *sending messages* to and *receiving messages* from the system. For examples, a Cashier sends a message to record an item, and the system sends a request to actor Bank for authorising a credit payment. We make the following remarks on use cases.

- Use cases describe functional requirements of the system from the actors' perspective, stating what the actors do to use the system for realising application tasks.
- Each use case uses part of the functionalities of the system, providing a natural divide and conquer strategy and helping to identify system components.
- Actors form the *external environment* of the application; they define the boundary of the system under design. This offers a basis for *interfaced-oriented design of ever evolving systems*, such as cloud-based systems, IoT, and CPS [39,43].

These features show that the use case driven approach is consistent with David Parnas' Four Value Model[9], and share the similar philosophical thinking to Michael Jackson's Problem Frames[10].

Identifying Use Cases. It is not in general easy to capture the use cases for an application. This obviously needs close collaboration between the requirements analyst and the domain expert. There are two approaches to find use cases, and they are often used together in practice.

1. **Actor-based**
   (a) identify the actors related to the system in the organisation, i.e., look at which users in the organisation will use the application and which other systems must interact with it; and
   (b) for each actor, identify the processes they initiate or participate in by reviewing how the actor communicates/interacts with the application.
2. **Event-based**
   (a) identify the external events that a system must respond to, and
   (b) relate the events to actors and use cases.

For example, in the Trading Systems the five uses case are easily identified through the actors Customer, Cashier and Store Manager. On the other hand, a use case for automatic checking the inventory of products and generating an alert to Store Manager can be identified by the alert and then this event is related to a "timer" as the initiating actor.

To identify use cases, we need to read the existing requirements from an actor's perspective and interact with those who will act as actors. It is necessary to think and discuss questions like:

- what are the main tasks of the actor?
- which of these tasks can be automated, and with what added values?
- will the actor have to read/write/change any of the system information?
- will the actor have to inform the system about outside changes?
- does the actor wish to be informed about changes?

6.3 Incremental Use Case Analysis

Capturing, understanding and describing use cases go hand in hand, and in an incremental manner. Taking the example of the Trading System introduced in Sect. 4 in Part I. Customer and Cashier are obviously associated with an use case Process Sale. After the identification of these actors and this use case, we can write down our initial understanding of this process as an overview below.

Use Case: Process Sale
Actors: Customer, Cashier

[9] http://en.wikipedia.org/wiki/DavidParnas.
[10] https://en.wikipedia.org/wiki/Problem_frames_approach.

Process of Interactions:

1. Customer arrives at the Cash Desk with items to purchase.
2. Cashier records the purchase items and collects payment.
3. On completion, Customer leaves with the items.

This *overview* is a rather abstract description, but it contains significant information about what this use case does as well as the what actors are involved.

Consider, as another example, a software system used in a university library. An actor Librarian is easily identified in the application, and then another actor, Member (of the library) that defines the people who use the library. A possible use cease is to register a member. We write the following overview of the use case.

Use Case: Register Member
Actors: Member, Librarian
Process of Interactions:

1. Member arrives at the reception desk with identification.
2. Librarian records the personal details and issues a card.
3. On completion, the Customer leaves with the card.

In the same way, we can identify more use cases and their actors for both the Trading System. A use case is a *complete course of interaction events described from the users' perspective.* This is a very important property of use cases, avoiding writing partial business processes as use cases or arbitrarily combining use case.

To support incremental development of use case documentation, we use the following structured format proposed in Larman's textbook [42] for writing overviews of use cases.

Use case:	**Name of use case** (use a phrase starting with a verb)
Actors:	List of actors
Purpose:	Intention of the use case
Overview:	A brief description of sequence of events in the process
Cross References:	Relations to other use cases and artefacts for **traceability**

Example. As an example of incremental use case analysis, we first consider a simpler version of Process Sale of the Trading System, which handles sales with cash payment.

Use case:	**Process Sale with Cash Payment**
Actors:	Customer (initiator), Cashier (direct actor)
Purpose:	Capture a **sale** and its **cash payment**
Overview:	A **customer** arrives at the **cash desk** with **items** to purchase. The **cashier** *records* the purchase **items** and *collects* the **cash payment**. On completion, the Customer leaves with the items
Cross References:	Restricted case of use case Process Sale

Notice in the above representation, the nouns that represent important concepts and objects in the domain are written in bold so that they will be identified as classes later when we develop the *class model* of the domain. This approach is suggested to adopt in practical project development. Also, some important verbs are emphasised, such as *records* and *collects*. They indicate possible interaction events.

The above high level overview use case, with only a couple of significant concepts (nouns) and interactions (verbs), does not contain sufficient information for identifying enough interaction events, concepts and objects. Further analysis of a high level use case through meeting with domain experts and end users are needed to refine it to a detailed use case, called an **expanded use case** in [42]. The refinement focuses on details about the interaction actions between the actors and the system and information about:

- input data an actor provides to the system when performing an interaction,
- data that an actor receives after an interaction,
- what the system does when an actor performs an interaction action, e.g., what objects are created, data updated, checked or read[11],
- the main course of actions and the time when exceptions may occur and how to handle them, and
- invariant properties that are preserved by the actions, including specific safety properties of the business process.

For expanded use cases, we also follow the structured format for their documentation proposed in Larman's textbook [42], which extends a high-level use case with a section of typical course of events and a section of alternative courses of events (or exceptions), respectively. The typical course of events is presented in an style of a conversation between the (direct) actors and the system.

Expanded Use Case Process Sale with Cash Payment

Use case:	**Process Sale with Cash Payment**
Actors:	Customer, Cashier
Purpose:	Capture a **sale** and its **cash payment**
Overview:	A **customer** arrives at the **cash desk** with **items** to purchase. The **cashier** *records* the purchase **items** and *collects* the **cash payment**. On completion, the Customer leaves with the items
References:	Restricted case of use case Process Sale

[11] This is the basis for identification of postconditions of an operation in Sect. 8.

Typical Course of Events

	Actor Action		**System Response**
1.	This use case begins when the **Customer** arrives at **Cash Desk** with **items** to purchase.		
2.	**Cashier** starts a new sale.	**3.**	Creates a new sale.
4.	**Cashier** *records* the **identifier** for each **item**.	**5.**	Determines the **item price** and *adds* the **item information** to the **running sale transaction**.
	If there is more than one item, the Cashier may enter the **quantity** as well.		The **description** and **price** of the current **item** are presented.
6.	On completion of item entry, the Cashier ends item entry is completed	**7.**	Calculates and presents the *sale total*.
8.	Cashier tells the Customer the total.		
9.	Customer gives a **cash amount**, possibly greater than the sale total.		
10.	Cashier *records* the **cash amount received amount**.	**11.**	Shows the **balance** due back to the Customer. Generates a **receipt**.
12.	Cashier deposits the cash received and extracts the amount back. Cashier gives the balance owing and the printed receipt to the Customer.	**13.**	Logs the **completed sale**.
14.	Customer leaves with the items purchased.		

Alternative Courses

- Line 4: Invalid identifier entered. Indicate error.
- Line 9: Customer didn't have enough cash. Cancel sales transaction.

This is slightly different from the use case "Buy Items with Cash" in Larman's book [42]. There, there are no steps **2** and **3** to start a new sale. Instead, these

two steps and the following steps **4** and **5** are combined into the following step of interaction.

2. Cashier records the **identifier 3.** If it is a new sale creates a sale. for each **item**.	Determines the **item price** and *adds* the **item information** to the **running sale transaction**.

This shows that a business process can be represented by use cases with different sequences of interactions. The decision on which one should be selected in the requirements specification needs to be made by the client and the project enveloper together. For complex cases, validation, say by prototyping or scenario plays, should be conducted to help the decision making. Experience from teaching shows that our slightly longer use case description is easier to formulate, and there is a general pattern that a use case often has a 'start' operation by a direct actor. For example, 'start to register a new customer', and 'start to make a new order'. On the other hand, after one use case is described and understood, it may be changed into another acceptable by further decomposing interactions or combining interactions[12].

Remarks. A use case description always implies, explicitly or implicitly, assumptions on the functionalities of the use case. Significant assumptions are better to be stated clearly for the sake of further refinement. The above stated Process Sale with Cash Payment makes the following assumptions:

- there is only one cash desk;
- there is no inventory management;
- no tax calculations (that is needed in the U.S.) or coupons;
- no record maintained of customers (some important business analysis or analysis of customer's buying habits (related to big data);
- no control of the cashbox;
- name and address of store are not shown on the receipt; and
- Cashier ID and CashDesk ID are not shown on receipts.

The given simplified Process Sale with Cash Payment can be refined step by step by adding more details to remove these restrictions. This can either be done at the requirements gathering and analysis phase, or in another cycle after the design, or even after the implementation, of this simplified use case.

Writing a good use case description requires experience, but we do not have the luxury of space for more examples. Readers can practise on the other use cases of the Trading Systems, use cases of a Library System, or an ATM system. For example, one can identify and describe the use cases of Check Balance, Deposit Money, Withdraw Money, and Transfer Money. More examples of us ceases can be founded in textbooks on use-case driven development processes [40].

[12] Though the use case of Process Sale with Cash Payment and the use case of Buy Items with Cash [42] are both adequate, one is not a *refinement* of another by formal theory of refinement or process simulations.

6.4 Use Case Diagrams

A use-case diagram represents a set of use cases, actors, and their relationships. In a use case diagram, as the one shown in Fig. 7, an oval represents a use case, a stick figure represents an actor, a line between an actor and a use case represents that the actor initiates and/or participates in the process[13]. The diagram shows three simple use cases of the Trading System.

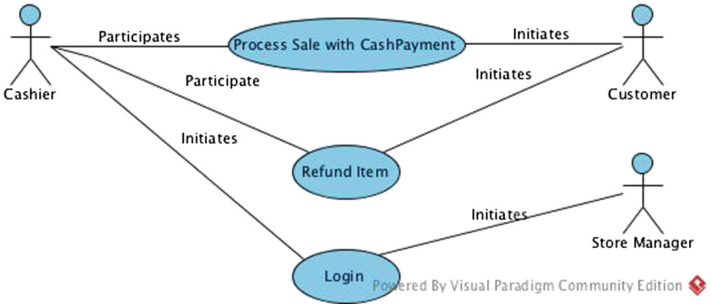


Fig. 7. An example of a use case diagram

We are not going to give the full definition of the syntax of the UML use case diagrams, as they can be found in many books and on many websites. We focus on the use of the modelling notation, thus we introduce the syntactic features that we need along with our discussion. Use case diagrams offer a means of organising use cases into groups such that the use cases in a group are logically related in the application domain. In Sect. 10.1 in Part III, each use case is modelled as a component that can be further decomposed into subcomponents. Also the components of the use cases in a use case diagram can be composed into a composite component.

6.5 Use Case Composition

Composition is essential in incremental development, just as decomposition is essential for divide and conquer. Now we show how to decompose large use cases into compositions of smaller ones. As an example, it is not difficult to see that the sequence of actions from steps **9** to **13** in the use case Process Sale with Cash Payment can be treated as a (sub-)use case, which we call Pay by Cash. In the same way as we identify and describe Pay by Cash, we can have two more use cases Pay by Credit and Pay by Check, both in addition to Cashier and

[13] The diagram, as many of the other UML diagrams in this chapter, is produced by using Visual Diagram www.visual-paradigm.com/features/uml-and-sysml-modeling.

Customer involving actor Bank for payment authorisation. We write the typical course of events and alternative courses of Pay by Credit as follows.

Typical Course of Events

Actor Action	System Response
1. Customer tells Cashier she wants to pay by credit.	
2. Cashier request for credit authorisation with card information and the total of the sale.	**3.** Sends Bank the credit authorisation request.
4. Bank sends back credit payment approval.	**5.** Logs the complete sale and generates the receipt.

Alternative Courses

– Line 4. If credit payment authorisation is denied raise exception.

This example also shows tricky decisions on the levels of abstraction. In the above description, the credit payment is decomposed into two phases of interactions: Cashier asks the system to enter a state for credit authorisation with the required input data, the system sends the authorisation request to Bank, Bank sends back the approval to the system and the system completes the payment and logs the sale. If PIN is required for the authorisation, Customer should also be a direct actor, and more interactions are required. One can also decide that the action of the sending the requests by the system to Bank for the credit authorisation will return a value that can be seen by Cashier. In this case, step **4** is changed to *Cashier indicates the system to complete the payment*. One can also decide to make the credit payment as one atomic step of interaction as follows.

Typical Course of Events

Actor Action	System Response
1. Customer tells Cashier she wants to pay by credit.	
2. Cashier records the credit payment information.	**3.** Sends the credit payment to Bank for authorisation. If the request is approved, it completes the payment. Logs the complete sale, and generates receipt.

Alternative Courses

– Line 2. If credit payment authorisation is denied, raise exception.

We will later see these different decisions also lead to different models of use case sequence diagrams, state machine diagrams, and contracts of the interaction operations. They also encounter theoretical problems. For example, the contract of the above 'big' operation of "records the credit payment" is hard to define as its execution involves calling for services, and the precondition relies on the result of the authorisation request from Bank.

Now we can use the three use cases for handling payments to form the general **Process Sale** use case, for which the typical course of actions is as follows.

Typical Course of Events

Actor Action	**System Response**
1. This use case begins with a Customer arrives at the Cash Desk checkout with items to purchase.	
2. Cashier indicates the system to start a new sale.	**3.** Creates a new sale.
4. Cashier records the identifier from each item. If there is more than one of the same item, Cashier can enter the quantity as well.	**5.** Determines the item price and adds the item information to the running sales transaction. The description and price of the current item are presented.
6. On completion of the item entry, Cashier indicates to the CashDesk that item entry is completed.	**7.** Calculates and presents the sale total.
8. Cashier tells Customer the total	
9. Customer chooses a payment method: (a) If cash payment, **initiate** *Pay by Cash*. (b) If credit payment, **initiate** *Pay by Credit*. (c) If cheque payment, **initiate** *Pay by Cheque*.	
	10. Logs the completed sale. And prints a receipt.
11. Cashier gives the printed receipt to the Customer.	
12. Customer leaves with the items purchased.	

A composite use case can be represented in a use-case diagram as shown in Fig. 8.

Remarks. About use case composition and decomposition.

1. In general, a use case may contain decision points. If one of these decision paths represents the overwhelming typical case, and the others alternatives

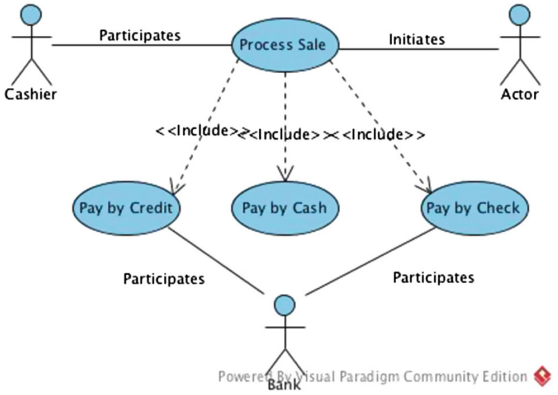


Fig. 8. Use case diagram for composite use cases

are rare and exceptional, then the typical case should be the only one in the Typical Course of events, and the alternatives should be in the Alternative Courses section. However, if the decision points represent alternatives which are all relatively common, then they appear as individual use cases in the main use case, such as Process Sale.

2. A use case represents a complete business process. It is not right to break a use case into arbitrary sequences of interactions. Neither is it right to compose arbitrary use cases to form a bigger one. For example one should not put Process Sale and Refund Item together to form a use cases.
3. Later, we will represent each use case as a *component* to provide services (interfaces) to it actors, and a composite use case corresponds to a component composed of the components of the sub-use cases. However, use case diagrams do not model interfaces of components. We introduce component diagrams in Sect. 10.1 of Part III.

6.6 Use Case Analysis in Development Process

The key steps for capturing and describing use cases are as follows.

1. Identify actors and use cases according to business (or domain) processes.
2. Carry out incremental use case analysis: from abstract high level to expanded versions with data to input, output, store and update.
3. Decompose and compose use cases.
4. Analyse and document the explicit and implicit assumptions made by each use case.
5. Rank use cases for project planning.
6. Present use case diagrams to group use cases belonging to business processes.

Write the expanded version most critical, influential and risky use cases to better understand and estate the nature and size of the problem.

The Significance of Use Cases Analysis

- Use cases answer the question what the system should do it in terms of what the system should do for each user/actor. They therefore identify the functionalities/services that the system provides to its users, and help to remove functionalities that do not have added values.
- Use cases identify relations among the system services/functionalities, as well as the required system services/functionalities. Therefore, services and functionalities are related in the processes to which they contribute.
- Use cases also relate functionalities and services to concepts and objects in the application domain. As illustrated in the next section, these concepts and objects are essential in modelling the structure of the system, and are later in the design realised as software classes and objects.
- Use cases are also used as "placeholders" for non-functional requirements (now widely called *extra functionality*), such as performance, availability, timing constraints, accuracy, and security requirements that are specific to a use case. Consider the use case Pay by Credit: It may be required that the authorisation result will be received in 30 s after the cashier makes the authorisation request. We will not discuss non-functional requirements in this chapter.

Relation to the rCOS Formal Method: The use case descriptions are purely textual and the use case diagrams are only a static organisation of the use cases. The analysis and description are not direct related to any formal techniques for requirements analysis. However, the conversational style presentations of the expanded use cases form the basis for obtaining the use case sequence diagrams in Sect. 8, and the use case diagrams will be further refined into component-based architecture models in Sect. 10.1. Both sequence diagrams and component-based model of architectures can be formalised in rCOS. However, we refer the reader to the paper [49,59] on a model of formal and use case-driven requirements analysis in the UML.

7 Requirements Modelling and Analysis – Conceptual Class Model

An important activity in object-oriented requirement analysis is to identify the domain concepts and their relations to create a *conceptual class model* of the domain[14]. The term "domain" covers the application area, e.g., the supermarkets in the Trading System case study. This section introduces the principles and techniques for identifying concepts which are meaningful in the problem domain, and the notation for representing them as a conceptual class model. The main objectives of this section are to

[14] A "conceptual model" is also called a "domain model". A conceptual model can also be seen as an *ontology model* does in information systems, but it is the part of the ontology model that contains only the concepts and relations relevant for the requirements of the system under design.

- understand the concepts of classes, associations and attributes;
- identify classes, their attributes and associations in the problem domain;
- understand the defining properties of objects that are important for distinguish objects from attributes of objects;
- define class diagrams and their use;
- understand the relevance and consistency relation between the conceptual model and the use cases model of a project.

The artefact that domain concept analysis produces is the conceptual class model mainly formed by packages of *conceptual class diagrams*.

7.1 Concepts in Conceptual Models

Object-orientation supports divide and conquer by dividing the problem domain in terms of interrelated individual concepts and their objects. These concepts and objects are meaningful in the application domain and they are relevant to the required business process.

A concept is an idea, a thing, or a set of objects. More formally, a **concept** is considered in terms of three defining elements: **symbol**, **intension** and **extension**[15]:

- A **symbol**, in the form of words (or images) is used to refer to the concept when communicating and discussing about the concept.
- The **intension** is the definition of the concepts, this is, what the concept intents to define.
- The **extension** consists of the individual examples or instances to which the definition of concept applies.

From now on, we use **instance** or **object** to refer an individual example of a concept. We also use *use cases* and *business process* as synonyms, though the latter is more often used in the context of the domain discussion.

Considering the realisation of "business processes" involves "objects" in the application domain and their "interactions", the above understanding of the term "concept" is fundamental to object-oriented modelling and analysis. For example, in the descriptions of Process Sale with Cash Payment, the concepts of "Store", "Sale", "Product Item", and "Cash Payment" are used. This implies that some instances of these concepts are involved in the execution of the process represented by this use case. The concepts which have instances involved in the realisation of a business process are said to be *relevant* to the use case. For another example, in the student management system of a university, the symbol Module is used to refer to the concept that has

- the intension to "represent a course offered as part of a degree in that university" with a code, title, and number of credits;
- the extension of all the individual modules being offered in the university.

[15] This is similar to *intentional* and *extensional* definitions of sets in mathematics.

Consider the concepts of "university" and "degree". Instances of "module" are parts of instances of "degree", and instances of "degree" are parts of instances of "university". Concepts and objects are defined in terms of other concepts, thus objects have hierarchical structures. Concepts also have **attributes** and concrete values of an attribute define a property of an individual instance. For examples, "Student" has name and age, and Module has code, title and credit; and a student with name John Smith and age 19. Actions in a use case may create (say when a student is enrolled) or destroy (say if a student is expelled from the university) instances of relevant concepts, as well as changing properties of existing instances.

Also, concepts are related to each other so that their instances can interact with each other in a use case execution. In the student management of a university, for example, Student and Module are related by a relation "Student Takes Module". The relation between concepts also defines a means for navigating from one object to the other. For example from a student instance, one can find about the modules that the student "takes". The attributes of relevant concepts of a use case are called *relevant attributes* to the use case if they are maintained, manipulated and transmitted in the use case, and relations among the relevant concepts that supports the interactions in the use case execution are the *relevant relations*.

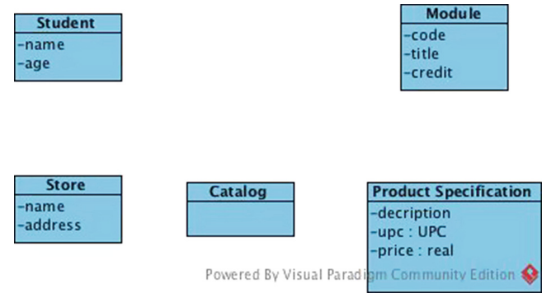
Concepts can have numerous attributes and relations, abstraction is about to only include the relevant concepts, attributes and relations, excluding anything irrelevant. This is a general object-oriented principle of modelling and design known as "the need to know principle".

In modelling requirements and designs, it is important to differ *attributes* of classes from *associations* between classes, and *properties* (values of attributes) of objects from *relations* (links of associations) between objects models. Attributes in general take simple values, instead of relating complex domain objects. It is common in object-oriented programming languages that associations are represented as attributes by pointers or references that have object types. However, associations do not have to be implemented by pointers or references, e.g., in relational databases.

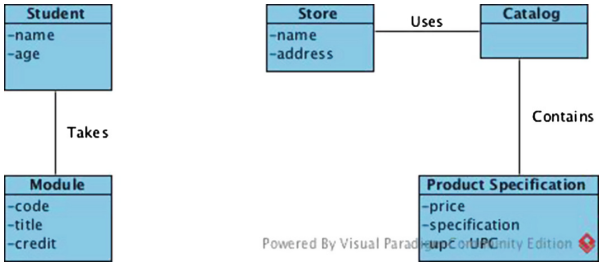
7.2 Classes and Class Diagrams in the UML

UML proposes a graphic modelling notation for **class diagrams** that models concepts as **classes**, e.g., in Fig. 9a, their conceptual attributes as **class attributes** (also called **members** in object-oriented programming languages), and relations between concepts as **associations**, e.g., in Fig. 9b.

Objects. Each instance in the extension of a concept is modelled as an **object** of the class that models the concept. The notions of classes and objects are interwoven as any object belongs to a class. The differences are: an object is a concrete entity exists in space and time (persistence property of objects); a class



(a) Classes in UML



(b) Associations in UML

Fig. 9. UML classes, attributes and associations

is a model or type for a set of objects. In the UML, an object is represent[16], as in Fig. 10.

The UML definition of a class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. This is the notion of **type correctness** of an object with respect to its class, covering classes used at all stages in an object-oriented development process, including classes in object-oriented programs. For requirements analysis, we focus on the classes, their attributes and relations for modelling the conceptual structure of the domain. The operations of the classes, which are more closely related to the functionalities of the use cases, are designed latter after further analysis of the behaviour of the use cases.

Fig. 10. Objects in UML

[16] The UML Visual Paradigm tool does not support object-diagrams in the way that we want to use them. Here we abuse the Visual Paradigm class diagram by prefixing the class name with (optionally) a name followed a colon ':' to represented the fact that the instance has type of the class, and giving values to the attributes.

The following characteristics of objects are important for identifying concepts and objects, and for distinguishing between concepts and attributes.

- **Object identity:** Every object is distinguishable from every other object. This is true even if two objects have exactly the same properties. For example, two instances of Student may have the same name, age, doing the same degree, in the same year, taking the same courses, etc. This means "complete properties" of an object are not enough to identify an object.
- **Object persistence:** Every object has a life-time from it is created to it is destroyed, and this characteristic implies the static nature of the system.
- **Object's behaviour:** An object has dynamic behaviour and may act on other objects and/or may be acted on by other objects. That is, the properties of an object keep changing during its life-time. For example, a Sale object can be created, it creates SaleLineItems objects, and it also creates its related Payment object.
- **Object state:** An object may be in different state at different time and may behave differently in different states. For example, the Sale instance can be paid only when it is in the state "complete".

An object in the real world can have many kinds of behaviour. For example, a car can be made, repaired, driven, and stolen; carry passengers and carry objects. However, only the behaviour that is exhibited in the required use cases is relevant.

From the defining characteristics of objects we can see that, in the Trading Systems for example, Product Item is a class but its attribute "price" is not, as the properties of identity does not apply to prices (e.g., identical prices are not distinguishable in the conceptual domain). Similar, one can identify the concept of "Receipt". It is subtle to decide if it should be modelled as an object or an attribute. It should not be an object if it is not required in any use case to be checked and/or changed. It should, however, be modelled as an object if receipts carry details of information to be checked and changed, for example when they are used for a Refund Item use case.

Identifying Concepts from the Application Domain: A central difference between object-oriented analysis and structured analysis is decomposition by concepts (objects) rather than by functions. Concepts and objects are gathered from the client requirements documents and most importantly from the use case descriptions. In concept and object identification, a useful and simple technique is to identify the *nouns* and *noun phrases* in the textual descriptions of a problem domain, and consider them as candidate concepts or attributes. Larman suggests in his book [42] to use category lists of concepts to identify classes. This is quite effective for experienced analysts. However, care must be applied when these methods are used; mechanical noun-to-concept mapping is not possible. Different nouns can represent the same concept, and different occurrences of a noun in different context may on the other hand refer to the same concepts (a fundamental interoperability issue in information systems). Also, nouns can be

about attributes, events, or operations, none of which should be modelled as classes. The defining characteristics of objects are always important in analysis for deciding on concept candidates, and these should be considered in relation to the use cases. This is the general principle of "*the need to know*" to follow in modelling requirements, so as to identify the relevant concepts.

Concept Candidates in the Trading System: Following the principles of conceptual modelling discussed above, the table of concept candidates in Fig. 11, are (very likely) related to Process Sale with Cash Payment.

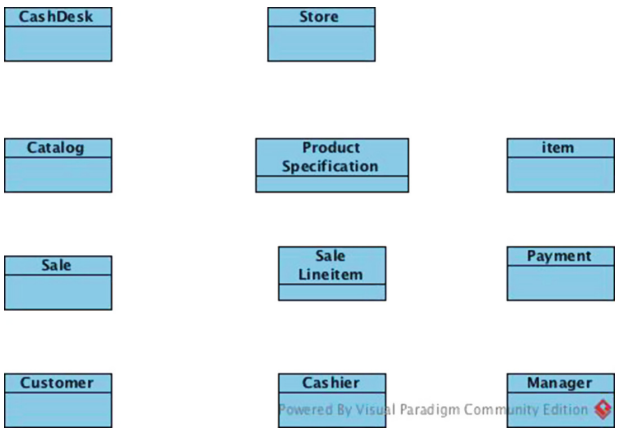


Fig. 11. Concept candidates for Process Sale with Cash Payment

The initial set of candidates may be incomplete and redundant. In later stages of analysis and design of use case behaviours, these candidates of concepts and the conceptual class models obtained need to go through a number of iteration steps of refinement to add and remove classes, attributes and associations. This relation between conceptual class models and use cases is formalised in rCOS [27,55]. This relation is in principle the same as the "completeness of program variable declarations with respect to the program body", where omission of variables could be detected by the compiler.

7.3 Associations

As shown earlier in Fig. 9, concepts are related, and object needs to be related so as to interact with each other when involving in computations. A conceptual model with only totally independent classes is useless.

In UML, an association is a relation between two classes that specifies how instances of the classes can be linked to work together. Associations and classes are at the type level of abstraction while, like objects being instances of classes,

links are at the level of instances classes. An instance of an association is a **link** between objects of the two associated classes. Therefore, objects in the same class share the same relationships. For example, in Fig. 9, the association Takes relates classes Student and Module. An individual student, whose name is for example John Smith is linked with a particular module, which has the code MC206 if this student John Smith takes Module MC206. This is represented by an object diagram as shown in Fig. 12.

Fig. 12. Link between objects

The notion of **type correctness of a link** for an association can be defined as for the type correctness of an object with respect to a class.

Multiplicities of Associations. For an association between classes, an important information is about how many objects of one class can be associated with one object of the other. We use **multiplicity** to represent this information. Figure 13 shows the most often used multiplicity expressions and their meanings.

Determining multiplicities often exposes hidden constraints built into the model. For example, whether Works-for association between Person and Company in Fig. 14 is one-to-many or many-to-many depends on the application. A tax collection application would model the case when a person works for multiple companies. On the other hand, a workers' union maintaining member records may consider second jobs irrelevant. Therefore, multiplicities represent part of the business rules of the application. More examples of associations are shown in Fig. 15.

There can be more than one association between two classes, and there can be associations on a class itself. Whether an association is useful or not depends on whether it provides a means for providing the objects with links to interact with each other in the computations relevant to the use cases, i.e., "the need to know principle". In general, a link denotes the specific association through which one object (the client) calls for the services of another object (the supplier). Figure 16 shows a partial class diagram for Process Sale with Cash Payment.

Roles of Associations. In a class diagram, each association connects two classes, one at each end. They are called the **roles** of the association, which may have **role names**. Naming a role in a conceptual model is sometimes useful, especially for an association on a class itself. As shown in Fig. 17 for example, the role names 'boss' and 'worker' distinguish two employees who work for a

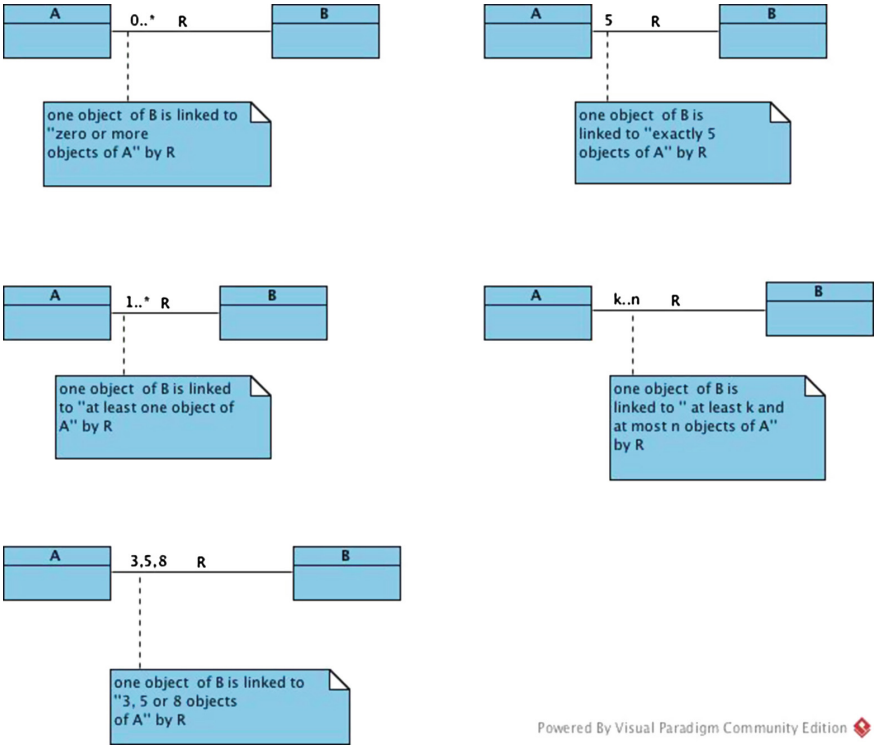


Fig. 13. Multiplicities

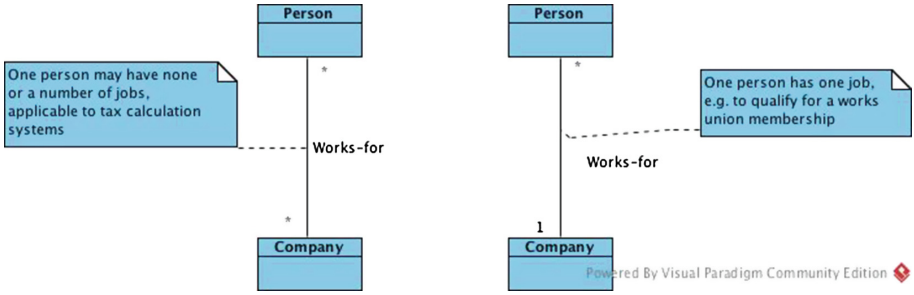


Fig. 14. Multiplicities depend on applications

company and participate in the Manages association. When we come to the design and implementation of the system, roles provide a way of viewing an association as a traversal from one object to the set of associated objects. In Java, a role name is the reference to the instance of the class, as shown below for role 'employer' of 'Company' in the association 'Works-for' of Fig. 17:

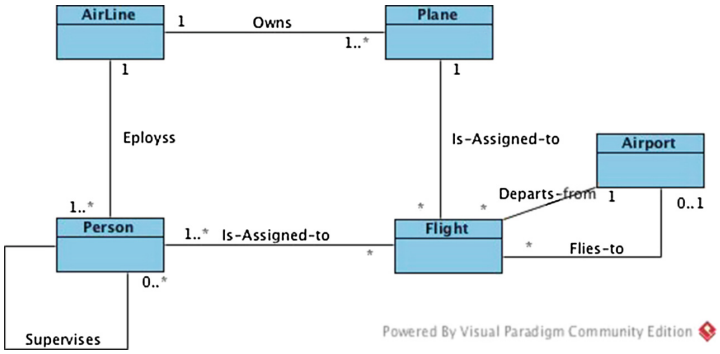


Fig. 15. Example of class diagram

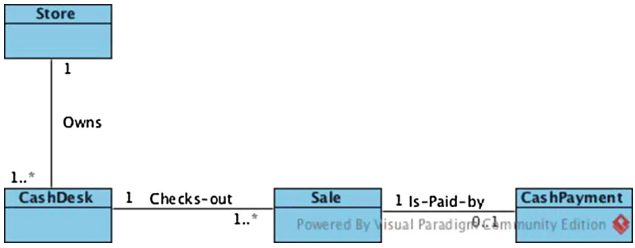


Fig. 16. Partial conceptual class diagram for Process Sale with Cash Payment

Fig. 17. Roles of an association

```
Class Person {
      Company employer;
      Person boss;
       }
```

Through identification of concepts, associations and their multiplicities, roles and using the UML notations, we build an initial conceptual class diagram for the use case Process Sale with Cash Payment shown in Fig. 18. The associations and multiplicities all represent assumptions on functionalities. For example, the one-one association Is-Used-by between Catalog and Store rules out the possibility of a Catalog being shared among stores. The completeness and correctness will be further analysed along with the behaviour analysis of the use cases. Formalisation of the semantics of conceptual class diagrams are important for rigorous checking [27,55].

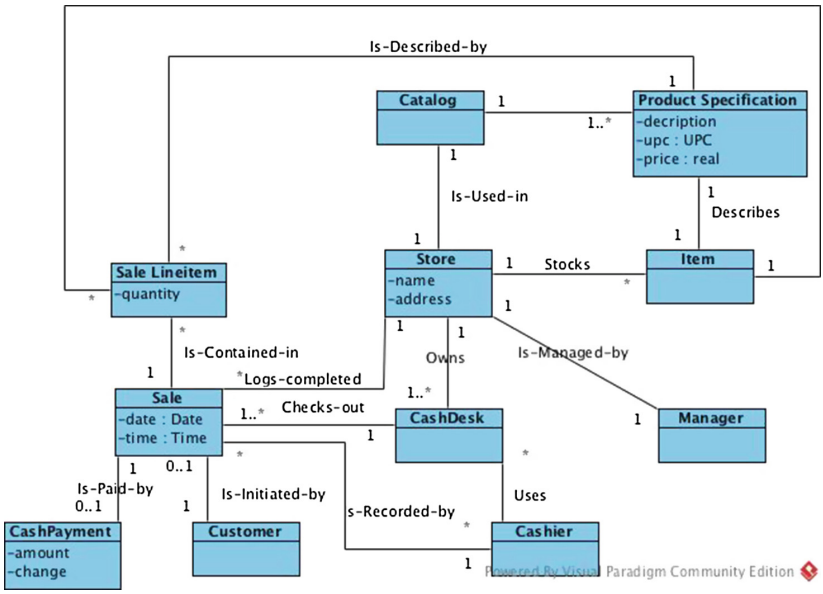


Fig. 18. Initial conceptual diagram for Process Sale with Cash Payment

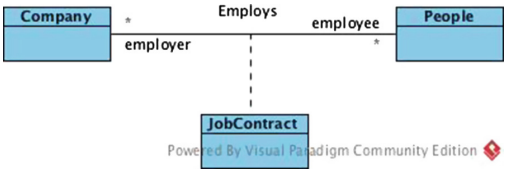
7.4 Association Classes

A one-one association or a one-to-many association is easily t implemented in a programming language by a pointer (C++) or a reference (Java) to an object or a container object, respective. However, it is rather difficult to implement a many-many association, such as the one in Fig. 19a. One solution is to use an **association class**. This is shown in Fig. 19b. But the semantics of association classes is rather difficult to define in the same way as for the other classes. We thus propose to decompose a many-many association into a one-to-many association and a many-to-one association as shown in Fig. 19c. Another example is that we can decompose the many-many association Student Take Module into a one-to-many association Student Takes Registration' and a many-to-one Registration Is-on Module. This is analogous to normalisation in relational data bases.

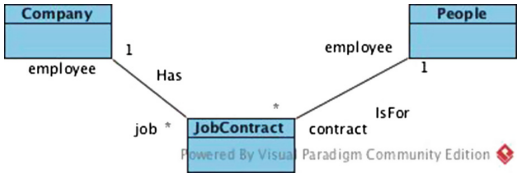
7.5 Aggregation Association

A special kind of relation between objects is the "part-of" relation, which share important common properties. We call a "part-of" relation between two classes an **aggregation**. For example, a "finger" is part of a "hand", thus class Hand aggregates class Finger. Similarly, a bicycle has as parts a frame, two wheels and up to two lights. For most aggregations, the multiplicity at the composite end may be at most one, and in this case the aggregation is called a **composition**. In the UML, an aggregation is represented with a "diamond" at the end of the aggregating class.

(a) A many-many association



(b) An association class



(c) Decomposition of a many-many association

Fig. 19. Many-many association and association class

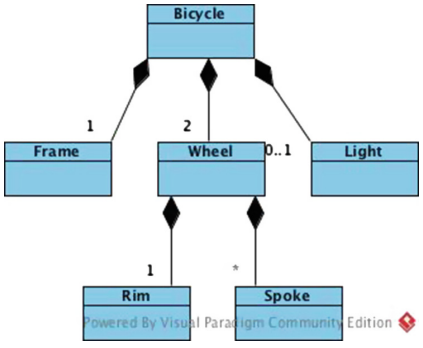


Fig. 20. Examples of composition

In some textbooks, **shared aggregation** is also introduced to model an aggregation where the multiplicity at the composite end may be more than one. Shared aggregation seldom (if ever) exists in the real world. We suggest to use general association to represent shared aggression if encountered in a project.

It is important to note two important properties of aggregation, which are useful for identifying and designing aggressions:

– *Antisymmetry:* states that if an object o_1 is related to an object o_2 by an aggregation, it is not possible for o_2 to be related to o_1 by the same aggregation. That is, if o_2 is a part of o_1 then o_1 cannot be a part of o_2;

– *Transitivity:* states that if o_1 s related to o_2 by an aggregation link, and o_2 is related to o_3 by the same aggregation, then o_1 is also linked to o_3.

These properties imply that the composition hierarchies are organised in forms of trees as shown in Fig. 20. The following requirements analysis patterns are useful when identifying aggregations in models of requirements.

– The lifetime of the part is within the lifetime of the whole - there is a create-delete dependency of the part on the whole.
– There is an obvious whole-part physical or logical assembly.
– The whole is a collection of the parts.

Using these patterns, we can polish the conceptual model in Fig. 18 by replacing the Is-Contained-in association between SalesLineItem and Sale, and the Contains association between Catalog and Product Specification by aggregations as shown in Fig. 21. If not sure when to use an aggregation, one can always use a plain association. Most of the benefits of discovering and showing aggregation relate to the phases of designing the software solution.

Fig. 21. Compositions in the initial conceptual diagram for Process Sale with Cash Payment

7.6 Generalisation-Specialisation Between Classes

Associations and aggregations represent relations among classes. The corresponding links are dynamic and changeable during executions of use cases. For example, a SaleLineItem instance only becomes part of a Sale instance after the item is entered by the Cashier, and a Sale instance is only linked through Is-Paid-by a CashPayment instance only after the Sale is completed and the CashPayment is handled and recorded by the Cashier.

There is another kind of relations between classes. Such a relation is called a **generalisation-specialisation** or **is-a** relation. We say a class A is a **generalisation** (or a **superclass**) of a class B if every object of B *is an* object of class A. In this case, class B is also called a **specialisation** (or a **subclass**) of A. A generalisation-specialisation structure gathers the common properties and behavioural patterns of different classes into a more general class, and specialisations partition a class into subclasses which share some common properties or behavioural patterns. Therefore, instances in a subclass **inherits** all the properties of its super class, but each of them **extends** the superclass with possible more properties. A generalisation-specialisation relation is a static relation between two classes that is not changed by operations of use cases.

Generalisation-specialisation provides a mechanism of **reuse** through inherence, that properties modelled in a supperclass can be reused, and *abstraction* by partitioning a general class into a number of subclasses. In Java, a specialisation (or sub-class) A of a class B is written as

```
Class A extends B {
     T x; U:u; V v;
    }
```

In this, certain attributes and methods, called *protected attributes* and *protected methods* of A are inherited by B, but B declares the addition attributes x, u, v (similar methods can also be declared).

However, in a programming language, specialisation introduces **polymorphism** when properties and methods of the superclass are redefined in the subclasses. Polymorphism provides flexibility of reusing attributes and methods of the super class, but it can be troublesome for verification of program correctness. A subclass is **refinement** of a superclass when all the properties and functionality of the superclass are preserved by the subclasses, and in this case the specialisation is also called **subtyping**. Specialisations identified from the application domain are usually subtyping (or refinement).

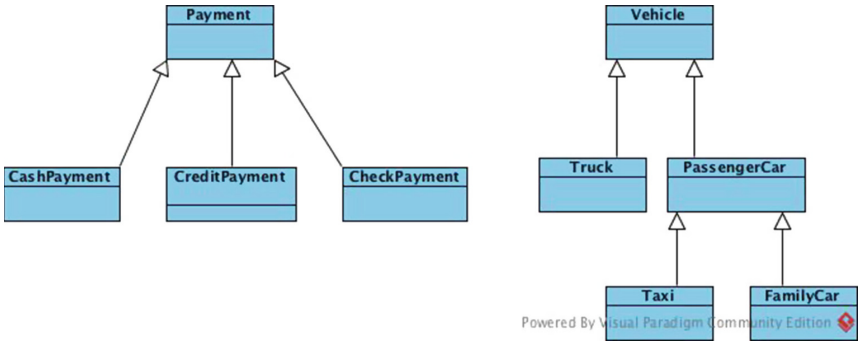


Fig. 22. Generalisation-specialisation hierarchy

An example of generalisation-specialisation in the Trading Systems is shown in the diagram on the left in Fig. 22. When all the instances of the superclass in a generalisation-specialisation relation is fully partitioned as the union of the instances of its subclasses, the superclass is called an **abstract class** and its identifier is signified in *italic* in UML. For example Payment will be an abstract class if only CashPayment, CreditPayment and CheckPayment are allowed in the Trading System, and in that case we would use "*Payment*" in the diagram. In Java, an abstract class is declared, for example, as follows

```
public abstract class Payment {
  }
Class CashPayment extends Payment {
  }
Class CreditPayment extends Payment {
  }
Class CheckPayment extends Payment {
  }
```

When a class is a specialisation of two or more different general classes, we say *multiple inheritance* occurs. For example, Mammal and Winged Animal are both specialisations of Animal, and Bat is a specialisation of both Mammal and Winged Animal. Java does not allow multiple inheritance from classes, but C++ do. Multiple inheritance introduces troubles when defining the semantics of the models, and programming languages. We do not exclude multiple inheritance in our models, though we do not have more examples of this.

7.7 Comments in Diagrams for Additional Constraints

We have discussed how multiplicities of associations are used for specification constraints, representing domain properties and business rules. However, a diagrammatic notation always has limited expressive power. UML provides "comments" for describing additional constraints on a model.

For example, Fig. 23 shows a partial conceptual class model. In this model, a library member who wants to borrow a Publication can make a Reservation on a Publication if no Copy of the Publication is available. When a Copy of the Publication is returned, the Copy is then held for a Reservation of the Publication. It is required that a Copy held for a Reservation is a Copy of the Publication reserved. This constraint cannot be represented by associations and multiplicities only. Thus, a comment is made in a comments box.

Constraints made in the UML diagrams are rather scattered and difficult to understand and manage. They can be formally specified using set theory and relation algebra, respectively, if an association is defined to be a relation between two sets, each defining a class [55].

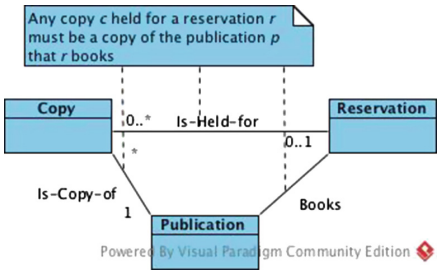


Fig. 23. Comment

$\forall c \in Copy \forall r \in Reservation \cdot (c\ Is\text{-}Held\text{-}for\ r \Rightarrow \exists p \cdot (r\ Books\ p \wedge c\ Is\text{-}Copy\text{-}of\ p))$
$Is\text{-}Held\text{-}for \circ Books \subseteq Is\text{-}Copy\text{-}of$

We thus define a **conceptual class model** to be a UML class diagram that may contain textual comments. Comments are part of the UML language definition, and they can be specified using the Object Constraint Language (OCL) analogous to the predicate above.

7.8 Relating Conceptual Class Models and the Use Case Models

As what we said earlier, the conceptual class model defines the domain structure for the use cases. A conceptual model is **adequate for a use case** if its classes, attributes and associations, together with the multiplicities and constraints imposed by the comments, support the behaviour specified by the use case. A conceptual class model is **adequate** for a use case model if it is adequate for all the use cases in the use case model.

Consider the two class diagrams in Fig. 14, the model on the left in the figure is adequate for all use cases that the class diagram on right of the figure is adequate for. Consider the two conceptual models in Fig. 24 for banking applications, that we call Small Bank and Big Bank, respectively. The only difference in the two diagrams is the multiplicities of the association "Has". The small bank only allows a Customer to have one Account and no Account is shared among Customers. On the other hand, the big bank allows one Customer to have up to 5 Accounts, and up to 3 Customers can share one Account. The small bank can support use cases "open an account", "deposit money", "check balance", and "withdraw money". The big bank on the other hand can support "transfer money" between two Accounts of the same customer as well.

(a) Small bank (b) Big bank

Fig. 24. Conceptual class diagrams

We say a conceptual class model $\mathcal{C}$ is a **structural refinement** of $\mathcal{C}_1$ if $\mathcal{C}$ is adequate for any use case that $\mathcal{C}_1$ is adequate for. Intuitively we have the proposition *that refactoring a conceptual model* $\mathcal{C}_1$ *in each of the following way generates a refinement of* $\mathcal{C}_1$:

- adding a class,
- adding an attribute to a class,
- adding an association between two classes,
- increasing the multiplicity of a role of an association (that is equivalent to adding attributes at the level of program code),

– promoting an attribute of a subclasses to its superclass, and
– promoting an association of a subclass to its superclass.

Obviously, structural refinement is reflexive and transitive. We say two class models are **equivalent** if they refinement each other.

At the current level of informal descriptions of use cases, we are not able to give rigorous definitions and checks the adequacy of a conceptual class model with respect to a use case model. We will come back to this topic in the next section when the specification of the behaviour of use cases is given in terms of contracts of use case operations. There, the contracts are defined by the change of the objects states of the class model.

7.9 Relation to the rCOS Formal Method

The conceptual class model focuses on what domain concepts and objects are, what structure the objects and concepts have, and what attributes the concepts have, for executing the required use cases. The use case description is very important for the incremental development of a good conceptual class model. The classes may not be complete for the execution of the use cases, and the missing classes will be discovered in later stage analysis and design. The class model may also contain classes that are not needed later, but they are useful for the domain understanding. For example, class Customer in the Trading System will not be implemented as a software class if information about customers does not need to be maintained or transmitted in the system.

Formal semantics of conceptual class models and their relations to use cases, and the design models (discussed in later sections) are give in our rCOS related publications, e.g., [27,49,55]. Incrementally adding classes, associations and attributes to a class model is also formalised in the sound and complete rCOS object-oriented refinement calculus [27,91]. Therefore, though the process of creating these models cannot be formalised, the models created are formalisable for verification.

8 Behaviour Modelling and Analysis

The process of identifying and describing the use cases and the activities of finding the concepts and building the conceptual class model is not in a linear order. Instead, they are interleaved, incremental and iterative, feeding back to each other. The conceptual model can be formalised, but the use case descriptions and use case diagrams remain informal. It is impossible to check formally the completeness and consistency of the conceptual class model with respect to the realisation of the described use cases. To this end, further modelling and analysis need to transform them, step by step, to models in the programming world. We will see that the closer to the programming world a model is, the more symbolic and precise it is. For this, we introduce more precise modelling notations:

1. the possible interaction sequences between the actors and the system in the execution of use cases, and
2. the state changes of objects caused by these interactions.

The first are modelled by UML *use case sequence diagrams*, and the latter by *contracts* of *use case operations* and UML *state machine diagrams* (a version *automata*). From these models, we define the concept of *components* and their *interfaces* in Sect. 10.1 in Part III. There, each use case is modelled as a component, and all use cases are combined to form a *component-based architecture*.

The key concepts of this section include *use case operations*, *use case sequence diagrams*, *object diagrams*, *object states*, *contracts of operations*, and *use case state machine diagrams*. The artefacts produced by this phase of analysis include the *use case sequence diagrams*, and use case state machine diagrams, *use case operations* and their *contracts*.

8.1 Use Case Operations

Recall that each expanded use case describes the patterns of interactions of its actors and the system under design, and interactions (communications) among actors. Each use case involves only a part of the functionalities required for the whole system, though different use cases may use some common functionalities. The first modelling decision we make is to treat a use case as a system *component*. The interactions among actors in the use case decryption are for requirements understanding, but they are not part of the functionalities required to design. Our second modelling is to eliminate the actors that do not directly interact with the system under design, such as Customer in use case Process Sale[17], from the model of interactions. Looking at the typical and the alternative courses of interactions, a use case describes the interactions of the direct actors and the component in the following way.

- **Input operations.** An actor generates an **input event** to the component to request an operation to be carried out. An input event may have input parameters to pass values to the component and return parameters for receiving values from the component. In the Process Sale use case for example, Casher generates *startSale*() to start handling a new Sale; *enterItem(upc)* to record an Item, where *upc* is an input parameter for the Universal Product Code (UPC) of the item; *finishSale*() to indicate the end of item input; and *makeCashPay(amount)* to record the CashPayment.
- **Output operations.** The component generates an **output event** to an actor to require a service from the actor. This is usually the execution of an operation in response to an input operation, but sometimes a component may also actively trigger an actor for the purposes of control and coordination. An

[17] In a further system evolution, more software components can be developed to automate these interactions so some of the interactions among actors are not needed anymore. For example, if online shopping is to be supported a customer would be a direct actor for the Make Order use case.

output event may have input parameters carrying values to be passed to an actor and return parameters to receive values from the actor. For example, the component for the Process Sale use case generates *authoriseCredit*() to the actor Bank for authorising credit card payment. By convention, the return value, including prompting message and signal, of an input operation or an output operation is part of the result of the interaction, but not an interaction event.

- **Repetitive interactions.** The interaction process may repeat a sequence of interaction operations, just as iterative statements in programming languages. Process Sale needs to repeatedly use *enterItem*() to record all the items that the customer is purchasing, until the last item is entered.
- **Branches.** Alternative branches of interactions may happen either controlled by an actor or by the component. The former is an external choices and the latter is an internal choices, both studied in communicating process description languages, such as CSP [31,82]. Process Sale may go through Process Sale with Cash Payment, Credit Payment or Check Payment; and also it can go into exception handling if the identity of an item, such as its Universal Product Code (UPC), is not recognised when executing *enterItem*(*upc*).

8.2 Use Case Sequence Diagrams

For precise modelling of the interaction patterns of a use case, we first introduce symbolic names to represent the input and output operations for each use cases. An input operation is called by actors and an output operation is called by the component as response to an actor. The patterns of the interactions of the actors and the use case are modelled as a UML sequence diagram. The diagram defines all the possible sequences of interaction events of all possible execution instances of the use case. Each instance execution of the use case is called a *use case scenarios*. For example, the sequence of events *startSale*(), *enterItem*(01000, 2), *enterItem*(01001, 3), *endSale*(), and *makeCashPay*(30) is a particular scenario. The understanding and analysis of a use case can start from the understanding of its significant scenarios.

We informally explain the syntactic elements of use case sequence diagrams, leaving the study of the syntax and semantics of use case sequence diagrams out of this chapter. A sequence diagram is formed with

- the use case components and actors life lines;
- messages that represent (a) input operations sent from actors to the use case component, and (b) output operations sent from the use case component to actors;
- the temporal order of messages is defined by the order in which they occur downwards along the life lines;
- "loop combined fragment" representing repeating sequences of interactions; and
- "alt. combined fragment" representing branching among sequences of interactions.

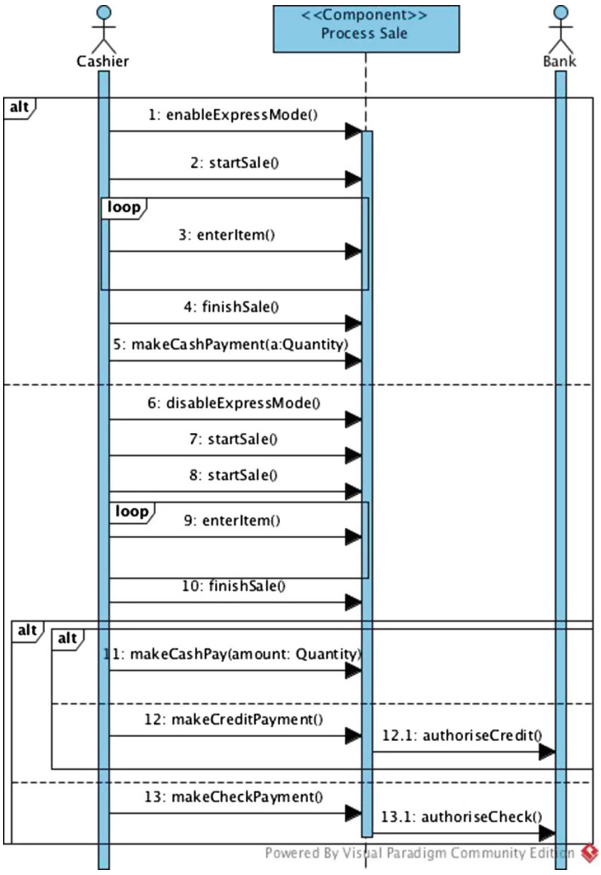


Fig. 25. Sequence diagram of Process Sale use case

Example. To illustrate the expressive power of use case sequence diagrams, we present the sequence diagram of the full use case of Process Sale in Fig. 25. Notice the nested choices, and the output operations of Process Sale. The sequence diagram becomes much simpler if express mode is disregarded, then only the bottom part starting from message 7:*startSale*() will be included. If we consider cash payment only, the component becomes closed and the sequence diagram will become the part from message 2:*startSale*() to message 5:*makeCashPayment*().

The interactions 11:*makeCreditPayment*() and 11.1:*authoriseCredit*() starts the sequence diagram for the Pay by Credit use case, but with the modelling decision that *makeCreditPayment*() treated as one atomic action. The use case sequence diagram for a different decision as discussed in Subsect. 6.5 is different, as shown in Fig. 26. Use case Pay by Check has the same design options to consider.

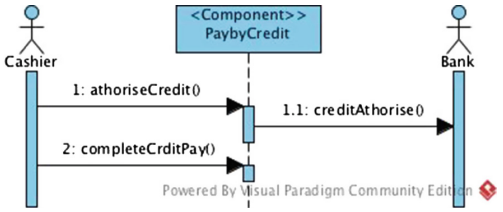


Fig. 26. Sequence diagram of an alternative model of **Pay by Credit**

Most UML modelling tools offer means of referring to sub-diagrams in complex diagrams. For formal analysis, the UML sequence diagrams can be automatically translated into a formal model, such as CSP processes [16,31]. In this chapter, we use two kinds of sequence diagrams. They are **component sequence diagrams** and **object sequences diagrams**, that are formally defined in rCOS [15,37,50]. A use case sequence diagram is a component sequence diagram, also called a **use case sequence diagram**. General component sequence diagrams will be introduced in Sect. 10.1 and object sequence diagrams in Sect. 11 in Part III.

8.3 Use Case State Machine Diagrams

The use case sequence diagram has a corresponding *state machine diagram* which is defined by a Statechart [24] and closely related to I/O automata [64]. If we ignore the express mode of Process Sale, we have the state diagram in Fig. 27 for Process Sale. The state machine diagram models the behaviour of the use case for verification (e.g., using model checking) of application dependent properties, such as safety and liveness properties. It can also be used for automatic generation of the interface control program. The states of a use case state machine diagram represents the conditions on the flow of control and synchronisation of the interaction processes when use case are being executed, thus they are called **control states**. A use case state machine diagram has a starting state, identifiable with a filled in circle. Most use case state machine diagrams also have final states, denoted by a circle with a black bullet inside, from which operations may stop. For example, an execution of Process Sale may stop after a sale is completed and paid, but it can also restart the execution to process a new sale.

The state machine diagram in Fig. 27 is consistent with the use case sequence diagram in Fig. 25. The transitions from state completeSale to the final state by the operations of *makeCreditPayment*(), *makeCreditPayment*()/*authoriseCredit*(), and *makeCheckPayment*(), *makeCheckPayment*()/*authoriseCheck*(), respectively. This models the input operations as single atomic operations with a *run to completion semantics* of method invocations. If we consider alternative models of these two input operations, such as the one shown in Fig. 26, the corresponding state machine diagram would be the one shown in Fig. 28.

However, different state diagrams can also model the same (observable) interaction behaviours. For example, the state machine diagram in Fig. 28 is

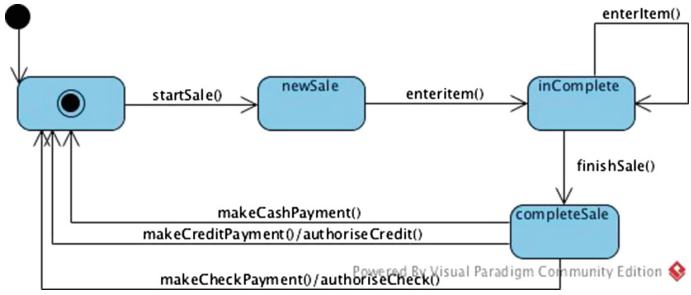


Fig. 27. State machine diagram of Process Sale

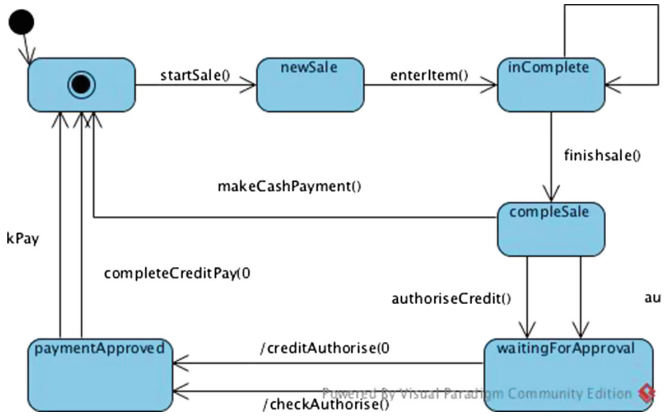


Fig. 28. An alternative state machine diagram of Process Sale

"equivalent" to the state machine diagram in Fig. 29. Consistency between the use case sequence diagram and the state machine diagram of a use cases can be formalised and automatically checked in rCOS. Equivalence between use case sequence diagrams and equivalence between state machine diagrams are studied in the formal theories of process refinement and simulation, such as CSP and CCS, and in theories of I/O automata and Statecharts.

8.4 Object Diagrams and Object States

Use case sequence diagrams do not have the concept of states and they only model interaction protocols. The states in a state machine diagram are symbolic and their names are insignificant, i.e., changing names of the states resulting in equivalent state machine diagrams. To design and implement a use case as a program component, we need to analyse and specify the *static functionality* of the interaction operations in terms of which *object state* changes they perform.

An **object state** of a component (or a use case or a system) is a snapshot at a particular time instant of the component execution, and it consists of the

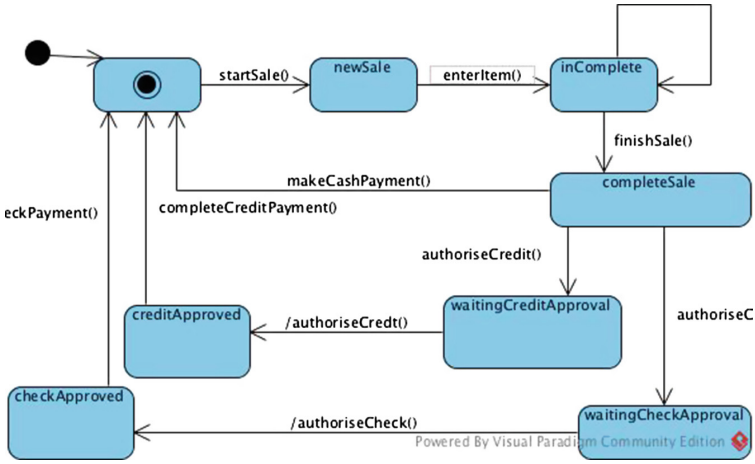


Fig. 29. Equivalent state machine diagram

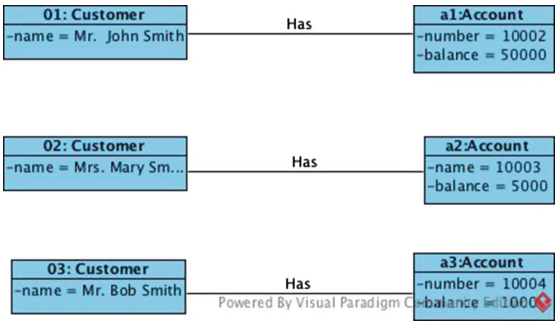
existing *objects*, *values of attributes* of the objects, and *links* between the objects. It is essential to understand the relation between the states of a component and the conceptual class model of the component. A possible state in the execution of the component must be **type correct** with respect to the class model. This means, the objects and links between objects must be instances of classes and associations of the class diagram.

We now define an **object diagram** Γ of a class diagram Θ as an instance of the class diagram, consisting of

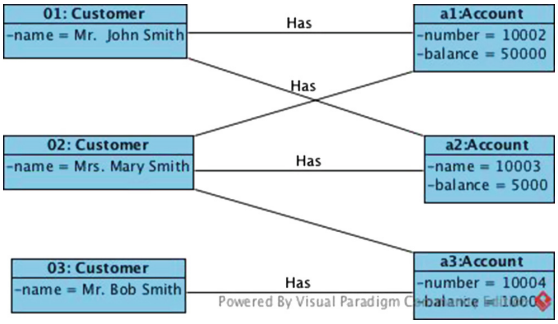
- a set of instances O, called **objects**, of some classes $\mathcal{C}$ in the class diagram,
- a set of instances L of some associations of the classes $\mathcal{C}$, which are **links** among the objects O, that satisfy the multiplicity and other (including the comments) constraints in class diagram Θ.

Example of Object Diagrams. Consider the conceptual class models of the small bank and big bank in Fig. 24. Figure 30(a) is an object diagram of both the small bank and big bank, but the object diagram in Fig. 30(b) is valid only for the conceptual class diagram of the big bank.

We call an object diagram of the class diagram of a component (or use case) an **object state** of the component. The **behaviour** of a component is the set of all possible sequences of state changes caused by the sequence of interactions between the actors and the component. We can equivalently define the behaviour of a component by the sequences of state changes caused by the operations defined by the state machine diagram. For example, consider a state in which the accounts related to customer Mrs. Mary Smith are shown in Fig. 30(b). Now Mrs. Mary Smith requests the big bank system to transfer 2000 GBP from her account $a2$ to account $a3$ that she shares with her son Mr. Bob Smith. After the execution of the transfer operation, denoted by *transfer*(), the system changes from the



(a) Small bank



(b) Big bank

Fig. 30. Object diagrams

pre-state shown in Fig. 30(b) to the **post-state** shown in Fig. 31: the balance of $a2$, $a.balance$, is reduced by 2000 GBP, and the balance of $a3$, $a3.balance$, is increased by 2000 GBP. Notice that an operation, such as *transfer*(), only changes part of the state of the component, keeping the other accounts unchanged, for example.

In general, a use case operation can change a state in the following ways:

- create new objects (object creation), such as opening an account in a bank application;
- remove (destroy) existing objects in the current state (object deletion), such as closing an account;
- form new links between both existing and newly created objects (link formation), such as an operation (in the Big Bank) of registering Mr. John Smith on account $a3$ so as to allow him to access that account too;
- remove exiting links from the current state (link removal), such as deregistering Mrs. Mary Smith from account $a3$;
- modifies values of object attributes (attribute change), such as *transfer*().

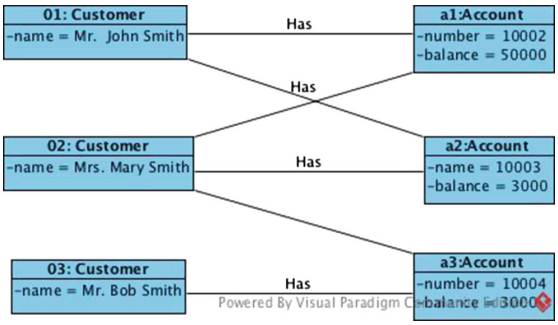


Fig. 31. Post-state of transfer operation

We describe the state change caused by an operation in an abstract way by stating what changes are made without saying how changes are made. A good analogy of a state change by an operation is a stage play performance. A snapshot on the of play on the stage shows the actors and actresses playing different roles, props, and background scenery, and their relations. Then the curtain is closed, changes are made behind the curtain (the execution of an operation) to change actors and actress, props, back ground scenery and their relations, then a new state is shown when the curtain is opened again.

8.5 Contracts of Operations

For systematic design of use cases, we need a clear and precise specification of the functionality of each use case operation, where the specification should define all the possible changes of states of the system. The specification of an operation $m()$ is called the **contract** of the $m()$, and it is formally defined to be a triple of the form

$$\{Pre\text{-}condition\}\ m()\ \{Post\text{-}condition\}$$

where

- *Pre-condition* is the condition that the states are assumed to satisfy before the execution of operation $m()$,
- *Post-condition* is the condition that the states have to satisfy when the execution of the operation terminates (if it terminates).

These triples are formalised in the well-known Hoare Logic [30] that is the foundation for formal program analysis and verification, and it is extended to object-oriented programs in rCOS [27,38,92]. Instead of introducing the formal specification of operation contracts in rCOS, we present a practical approach for informally stating these contracts. The advantage is that one does not need to have a background in formal logic, and these informal statements are yet formalisable for those who have the background. Also, the informal description of the contracts is a step that must be taken before further analysis, including formal analysis. We adopt the following format proposed by Larman [42].

Contract	
Name:	Name of operation, and parameters.
Responsibilities:	An informal description of the responsibility this operation must fulfil.
Cross References:	Such as use cases.
Note:	Design notes, algorithms, and so on.
Exceptions:	Exceptional cases.
Pre-conditions:	As defined.
Post-conditions:	As defined.

Examples. We now write the contracts of the operations of Process Sale with Cash Payment, which are largely from Larman's book [42].

Contract	
Name:	*startSale*().
Responsibilities:	Create a new sale and start the process.
Cross References:	Use case Process Sale with Cash Payment.
Exceptions:	If any of the precondition does not hold, indicate error.
Pre-conditions:	The objects Store, CaskDesk, Catalog exist and linked.
Post-conditions:	1. A new Sale was created. 2. The new Sale was associated with the CashDesk.
Contract	
Name:	*enterItem*(*upc:UPC, quantity:Integer*).
Responsibilities:	Enter an item and add it to the sale. Display item description and price.
Cross References:	Use case Process Sale with Cash Payment.
Note:	Use superfast database access.
Exceptions:	If *upc* is invalid, indicate an error.
Pre-conditions:	*upc* is valid, and Sale exists.
Post-conditions:	1. A LineItem was created. 2. The LineItem.quantity was set to quantity. 3. The LineItem was associated the Sale. 4. The LineItem was associated with the Product Specification.
Contract	
Name:	*finishSale*().
Responsibilities:	Indicates the end of item entry and get ready for make payment.
Cross References:	Use case Process Sale with CashPayment.
Pre-conditions:	The sale exists.
Post-conditions:	Attribute *isComplete* of Sale object was set to *true*.

Contract	
Name:	*makeCashPayment*(n : *Quantity*).
Responsibilities:	Record the payment and associate it to the sale, and log the completed sale.
Cross References:	Use case Process Sale with Cash Payment.
Note:	Use superfast database access.
Exceptions:	If the Sale is not completed.
Pre-conditions:	The Sale is complete.
Post-conditions:	1. A CashPayment object was created. 2. The attribute *balance* of the CashPayment was set to the input value n. 3. A the CashPayment object was associated to the Sale. 4. The Sale was associated to the Store to log the completed Sale.

In informal specification of the contracts, it is hard to make sure the preconditions and postconditions are complete or consistent. The preconditions, in particular, are hard to assume complete and they also imply implicit assumptions. For example, the condition "*upc* is valid" implies the existence of Catalog, and "Sale exists" implies the existence of the Store and CashDesk. Some other conditions implied are even harder to see. For example, the existence of the link between CashDesk and Store, and link between Store and Catalog, etc. From this discussion, we see the need for logic formulation and reasoning.

8.6 Guarded Contracts

Another issue is related to the precondition "the Sale is complete" of operation *makeCashPayment*(). Preconditions need to be checked when an operation is executed, and if a precondition does not hold an exception should be thrown. For this, class Sale should have a boolean attribute, say *isComplete*. Attributes of this kind would be very hard to identify in an early stage of requirements analysis.

In fact this issue relates to more advanced modelling theory. Consider the above contracts of operations in relation to the state machine diagram Fig. 27, we notice that

- *startSale*() can only be carried out in the starting state;
- operation *enterItem*() can only take place in states *newSale* and *inComplete*;
- *finishSale*() can only take place in state *inComplete*; and
- *makeCasPayment*(), *makeCreditPayment*() and *makeCheckPayment*() can only take place in state *completeSale*.

We introduce a boolean variable for each of the control states in the use case state machine diagram, for example, @*start*, @*newSale*, @*inComplete*, and @*completeSale*, which take the value true when and only when the state machine diagram is in the corresponding state. We then add to each contract a section of

conditions called **guard conditions**. An operation can be executed only when all the gaud conditions hold. An attempt to execute (or call to) the operation is refused when the guard condition does not hold. An example of a **guarded contract** is given below, where the guard is emphasised.

Guarded Contract	
Name:	*enterItem*(*upc:UPC, quantity:Integer*).
Responsibilities:	Enter an item and add it to the sale. Display item description and price.
Cross References:	Use case Process Sale with Cash Payment.
Note:	Use superfast database access.
Exceptions:	If *upc* is invalid, indicate an error.
Guarded conditions:	*The system is in a state such that* @*newSale* holds or @*inComplete holds.*
Pre-conditions:	*upc* is valid, and Sale exists.
Post-conditions:	1. A LineItem was created. 2. The LineItem.quantity was set to quantity. 3. The LineItem was associated the Sale. 4. The LineItem was associated with the Product Specification. 5. *If in state* newSale *change to state* inComplete.

Note that the postconditions of a guarded contract may also change the control state. An error message (but not an exception) can be given when an operation is attempted when a guard is false.

8.7 Start Up Use Case

Before carrying any of the business processes of the application, the basic business infrastructure needs to be set up. This, for example, including the Store, the Cash Desk, and the Catalog. This is the operation *StartUp*(), that most system have. The contract of *StartUp*() for the Trading System is with on Store and one Cash Desk can be specified by the following postconditions:

1. Store, CashDesk, Catalog and ProductSpecification were created (instance creation).
2. ProductSpecification was associated with Catalog.
3. Catalog was associated with Store.
4. CashDesk was associated with Store.
5. Catalog was associated with CashDesk.

The creation of use case sequence diagrams, state machine diagrams and the specification of the contracts of the use case operations have been developed from the use case description and identification of conceptual class diagrams, and they together form a clear model of what the system under design is required to do. From these models, the development process is ready to move into the design phase.

8.8 Consistency Among the Models

The use case sequence diagrams, state machine diagrams, contracts of operations and the conceptual class model are models of the same system from different view-points. Their integration is the whole model of the applications requirements. Therefore, they must be consistent. The sequence diagram and state machine diagram of a use must be consistent so the sequence diagrams defines the same set of possible interaction sequences as those which are accepted by the state machine diagram.

The conceptual class model is required to be **adequate** for the definition of the contracts of the operations. This means that a conceptual class model is adequate for the operations of the use case model if it defines all the object states that are specified in the responsibilities, the preconditions and postconditions of all the operations. Clearly, this definition of the adequacy of a class model is a revised version of the adequacy of the conceptual class model for the use cases defined in Sect. 7. If a conceptual class model is adequate for an operation, so is its structural refinement.

Now we have a clearer and more precise definition of structural refinement: a conceptual model $\mathcal{C}$ is a **structural refinement** of $\mathcal{C}_1$ if each type correct object diagram of $\mathcal{C}_1$ is also a type correct object diagram of $\mathcal{C}$. For example, the conceptual class model on the left of Fig. 14 is a structural refinement of the one on the right; and the conceptual model Big Bank in Fig. 24 is a structural refinement of the Small Bank.

8.9 Relation to the rCOS Formal Method

The development of rCOS [31,82] shows that both use case sequence diagrams and state diagrams can be formally defined and analysed. In particular, they can be both translated to CSP process expressions for checking their consistency and for formal analysis and verification [14–17]. The informal descriptions of operation contracts are formalised in the rCOS extension [27] to UTP [32]; guarded contracts are presented in [11,26]. The development of a complete and sound refinement calculus and its applications can be found in [15,38,91].

9 Summary of Requirements Modelling and Analysis

Sections 6.1 and 8 focus on notations, activities and models of requirements modelling and analysis. The emphasis of these is on understanding of the requirements, concepts, and operations related to the domain business processes for which automation by software is required. The investigation and analysis are often characterised as focusing on questions of what - what are the processes, concepts and objects, associations, attributes, and operations? We have the following models and descriptions for the answers to these questions with different levels of details and precision.

1. Use cases: high level and expanded descriptions and use case diagrams. They are from the application domain and for understanding the functional requirements (use cases can also used for non-functional requirements analysis) from the domain perspectives.
2. Conceptual class model: the concepts, objects and their relations that form the structure for the realisation of the use cases.
3. Use case sequence diagrams: identification of the interactions between actors and the system under design for the realisation of the use cases.
4. Use case operations and their contracts: specification the functionality of the interactions, that is what changes to the domain structure, that are modelled by changes of object states that interaction smay cause.

We use UML diagrams for the representation of the above models: UML use case diagrams, class diagrams, and use case sequence diagrams state machine diagrams, plus textual description of contracts of use case operations. The contracts are described in terms of pre-conditions and postconditions, that can be formalised in Object Constrain Language - defined as part of the UML. Our formal method of object-oriented and component-based design offers formal specification of contracts of methods [15,27,37] and formal semantics and relations of different UML models for requirements models and design [12,25,52,55]. For the formal techniques and tools to be applied, the pre-formal activities should be carried out first to create the models with intuitive understanding of their meanings.

Through the above discussed modelling and analysis activities, and the creation of the models, the development team and the clients building a thorough understanding of the requirements. The UML models and the textual descriptions as a whole give a fairly clear model of requirements, including the architecture, the data model, and the static functionality. Further formal techniques and tools can also be applied for formal validation and verification. Therefore, it is fairly justifiable that after these activities and with the models built, the development can move into the next phase, that is the design phase to be discussed two sections to follow.

10 Part III: Component-Based and Object-Oriented Design

10.1 Component-Based Architecture Design

In Sect. 7 of Part II, the structure of the application domain is expressed in terms of conceptual class models and use case diagrams. It is not easy in these models to relate the functionalities of use case operations. For example, different use cases may perform some same operations, but these cannot be clearly represented. This section introduces the component-based modelling concepts of *components*, *interfaces* and *compositions*.

10.2 Components

A **component** C is a program unit with *encapsulated data states* and *explicitly specified* **interfaces**. In this presentation, operations are method invocations, and therefore a component in our framework can have two kinds of interfaces, **provided interfaces** for operations on data in the component and **required interfaces** for operations on other components. Syntactically an **interface** declares a set of method signatures of the form $m(in; out)$ with a method name m, a list of input parameters in and possibly an output parameter out.

In Java, interface are defined as shown in the following examples.

```
Interface pInterface_Process_Sale {
    startSale();
    enterItem(upc: UPC);
    finishSale();
    makeCashPayment(amount: Quantity);
    makeCreditPayment(cardInfo,amount);
    makeCheckPayment(cehckInfo,amount)
 }

Interface rInterface_Process_Sale {
    authoriseCredit(cardInfo, amount);
    authoriseCheck(cardInfo, amount)
 }
```

An interface is implemented by a class. For example,

```
 Class CaskDesk implements pInterface_Process_Sale {
        startSale() {the contract of the operation to be coded};
        enterItem() {the contract of the operation to be coded};
        finishSale() {body to be designed};
        makeCashPayment(amount: Quantity) {the contract of
                              the operation to be coded};
        makeCreditPayment(cardInfo,amount) {the contract of
                               the operation to be coded};
        makeCheckPayment(cehckInfo,amount){the contract of
                                the operation to be coded}
       }
```

We use the UML graphic representations of interfaces, as for classes, and relate interfaces with class by the relation "a class implements an interface".

A component is **closed** if it does not have required interfaces, and it is **open** otherwise. We use the UML notation to represent classed and open components, as shown in Fig. 32, respectively. In the figure, C is a closed component that has a provided interface *pIFC* and D is an open components with a provided interface *pIFD* and a required interface *rIFD*.

Visual Paradigm does not show the encapsulated state variables and the declared methods of interfaces (except for those which only have a single method

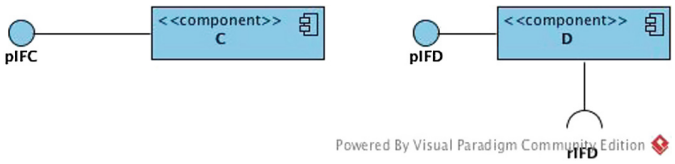


Fig. 32. Closed and open components

signature) of a component in the figure, but they can be specified with the textual editor and generated in the textual version. We define a component abstractly as a tuple $C = \langle X, pIF, rIF \rangle$, where X is the set of state variables, pIF and rIF are the provided interface and required interface, respectively. Each interface I is a set of method signatures, and we allow multiple provided interfaces and multiple required interfaces in diagrams but their unions are the provided interface and required interface in the abstract definition. The following examples show how use cases are modelled as components, and these are the initial components defined in the development.

Example. Consider the use cases Make Cash Payment, Process Sale with Cash Payment, Make Credit Payment, and Make CheckPayment, that can all be modelled as components in Fig. 33. The first two are closed components and the other two are open.

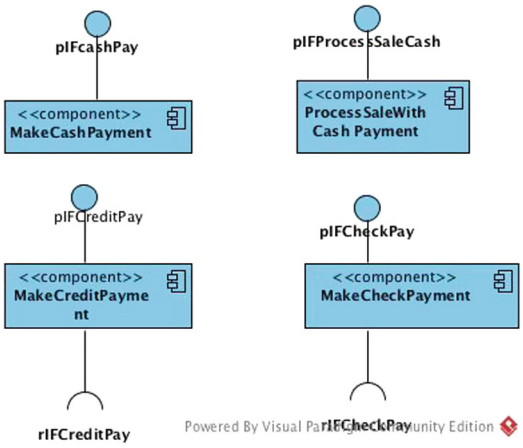


Fig. 33. Components of Process Sale

The input operations of a use case are declared in the provided interface of the component, and the output operations of the use case are declared in the required interface of the component. Provided interface *pIFCashPayment*

only declares one operation *makeCashPayment*(), while provided interface *pIFProcessSaleCash* declares the operations,

$$\{startSale(), enterItem(), finish\ sale(), makeCashPayment()\}.$$

Provided interface *pIFCreditPay*() declares the operation *makeCreditPayment*() and required interface *rIFCreditPay*() declares *authoriseCredit*(); and provided interface *pIFCheckPay*() declares *makeCheckPayment*() and required interface *rIFCheckPay* declares *authoriseCheck*(). The state variables of these components are the variables containing the Store object, Catalog object, CashDesk object, Sale objects and Payment objects, etc.

Neither a component diagram nor a tuple $C = \langle X, pIF, rIF \rangle$ specifies the functional behaviour of the component. We divide the specification of the behaviour of a component into two parts. First, the **static functional contract** of a component C is specified by the contracts of the operations declared in the provided interface *pIF*. The preconditions are assertions on the states variables, the input parameters of the operation; and the postconditions specify how the object states of the component are changed and what the value of the return parameter is. The contracts of the operations in *pIFProcessSaleCash* are the same as those given in the contracts of the input operations of the use case. For an example of contracts of operations involving required operations, the contract of *makeCreditPayment*() is

Pre-conditions:	Sale.*isComplete* is *true* the credit payment is authorised, i.e., *authoriseCredit*() return is positive.
Post-condition:	1. A CreditPayment object was created. 2. The CreditPayment object was associated with the Sale. 3. The Sale was associated with the Store to log the completed Sale.

The meaning of a contract of an interface operation is the same as that of a use case operation. If a precondition does not hold, exception handling needs to be designed. The static functional contracts of a provided interface are used to program the bodies of the methods in the class that implements the interface. The contract of an operation in a required interface of a component C is tricky in theory. In practice, however, it usually makes little change to the object state of C, and it usually sets some preconditions for some operations in the provided interfaces by changing the control state of C. For example, the condition that "the credit payment is authorised" in the above contract is set by the required operation *authoriseCredit*(). We leave further discussion of contracts of required operations out of this chapter.

In addition to the static functional contracts of the interfaces of the component, the flow of control for the protocol in which the interface operations can be executed should also be specified. This is called the **dynamic behaviour contract of the component** of the component, and it is modelled by a state machine diagram. For example, the state machine diagram in Fig. 27 of Part II is the dynamic

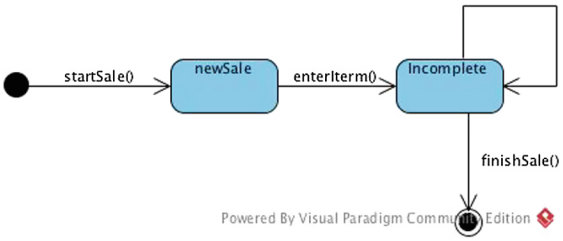


Fig. 34. State machine diagram of a finite component

behaviour contract of the component for Process Sale. Note that, the dynamic behaviour contract of a component can also be modelled by a sequence diagram, such as the sequence diagrams in Fig. 25 of Part II. The state machine diagrams, however, are easier to be associated with the static functional contracts of the component interfaces. We can associate assertions about object states to the control states of the state machine diagrams, for example, by adding comment boxes on the control states. The combination of the static functional contract and dynamic behaviour contract of a component are specified by the **guarded contract** of the interface operations, as defined in the previous section.

10.3 Composing Components

There is a need of mechanisms for components to be combined to form an architecture. For this purpose, the combinators known from concurrent and distributed program constructions, compositions for *sequencing*, *choices*, *parallel composition*, *service hiding* and *looping* (or *recursion*). Except for "plugging" to "link" provided operations to required operations, there are little UML tool support for these compositions. Here these compositions are defined as abstract operators, discuss their informal semantics, and then represented as special components called **connectors** in UML diagrams. Extending a UML tool to provide better support of component compositions, both their syntax and semantics could be an interesting student project.

Sequencing. Sequential composition is defined for "procedural components only", that is components that have terminating behaviour. For two components C_1 and C_2 whose required interfaces are disjoint with their provided interfaces, the sequentially composed components $C_1; C_2$ is defined if C_1 is terminating (or finite). When $C_1; C_2$ is defined, the provided and required interfaces of $C_1; C_2$ are the unions of the provided and required interacts, repetitively. The interaction behaviour of C_2 can start after that of C_1 terminates, i.e., the state diagram of C_1 enters a final state. Here C_1 is finite if its state machine diagram has a final state. Consider the behaviour from *startSale*() to *finishSale*() of the sequence diagram in Fig. 25 of Part II. We can model this part of Process Sale as a component

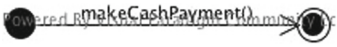


Fig. 35. State machine diagram of *PayByCash*

denoted by CashDesk (please note this is not a use case). The state machine diagram of CashDesk is given in Fig. 34.

The state machine diagram of the composite $C_1; C_2$ is to combine the final state of C_1 with the initial state of the C_2 into a new state so that the two state machine diagrams are composed. Consider the state machine diagram for component PayByCash for Pay by Cash use case as shown in Fig. 35. The state machine diagram of CashDesk; PayByCash is given in Fig. 36.

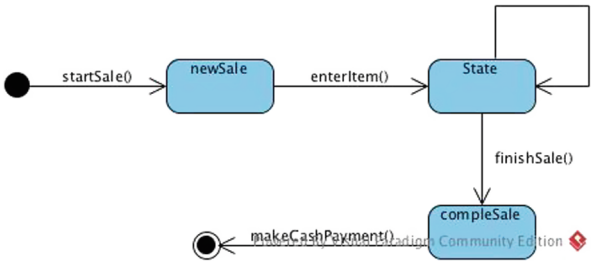


Fig. 36. Sequential composition of diagrams in Figs. 34 and 36

In a semantic theory, restrictions for $C_1; C_2$ to be defined are not needed. If C_1 and C_2 do not satisfy these conditions, it just make the behaviour of the composite component $C_1; C_2$ very complicated or even divergent. In practice, we would like to avoid these complicated cases when building our models. We use the **sequencing connector** component to represent the sequential composition. The sequencing connectors is shown in Fig. 37. It is easy to use the sequencing connector to realise CashDesk; PayByCash and this is the one iteration of Process Sale with Cash Payment.

Repeating. Similarly, the repeating composition is also defined for procedural components only. The **repeating operator** $*C$ is defined if C is finite and its behaviour is to repeat the behaviour of C as many times as possible (or as many times as the actors like). The state machine diagram of $*C$ is to combine the final state of the state machine diagram of C with its initial state.

The repeating connector is realised by the **repeating connector** component as shown by the diagram on the top of Fig. 38, and the combination use of the sequencing connector and repeating connector in forming the component ProcessSaleWithCashPayment, shown in the diagram at the bottom of Fig. 38. The state machine diagram of the composite component in Fig. 38 is given in Fig. 39.

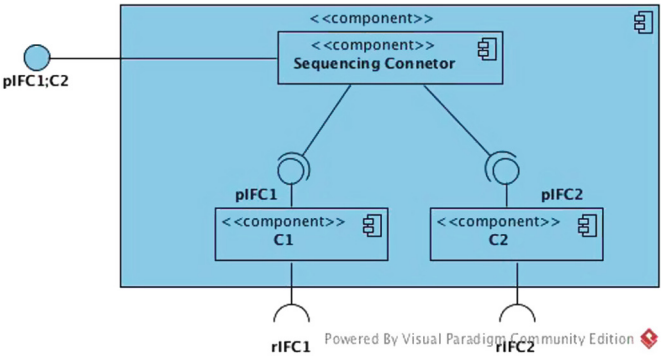


Fig. 37. Sequencing connector

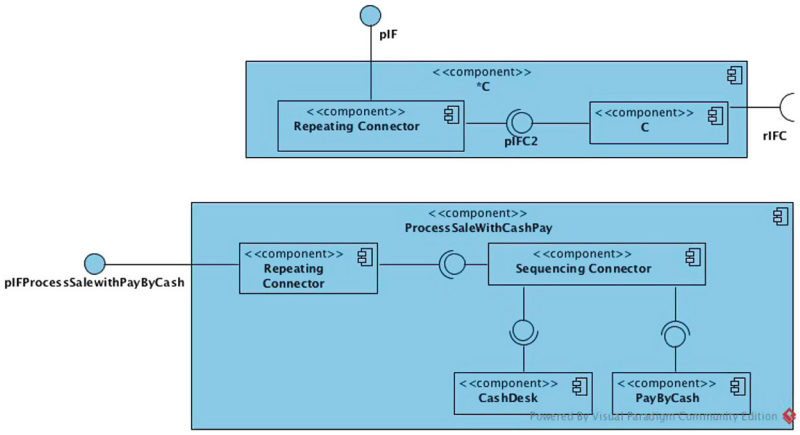


Fig. 38. Repeating connector

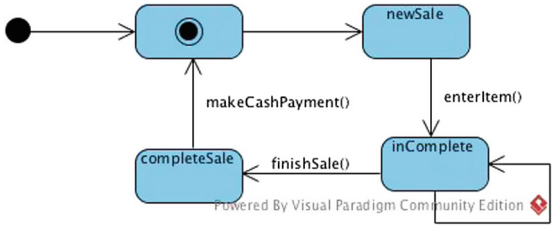


Fig. 39. State machine diagram of component ProcessSaleWithCashPayment

Choice. Given two components C_1 and C_2, the provided and required interfaces of $C_1 \oplus C_2$ are the unions of the provided interfaces and the required interfaces, respectively. The behaviour of component $C_1 \oplus C_2$ behaviour depends on the first move,

– if the first state transition is a transition of the state machine diagram of C_1, then the behaviour of C_1 is selected;
– otherwise the behaviour of C_2 is selected.

If C_1 and C_2 share operations in their provided interfaces which can take place from their initial states or one of these two components has autonomous transitions from the initial state, $C_1 \oplus C_2$ exhibits *non-deterministic* behaviour. Dealing with non-deterministic choice is both tricky in practice and theory. Without proper theoretical understanding of non-determinism, its practical handling is also hard. We should try to avoid non-deterministic choice in the first place when building component models.

Note that $\oplus$ is commutative and associative, thus $\oplus$ may combine an arbitrary number of components. We use the **choice connector** component to realise this general abstract choice operator. We consider the composition of $\mathsf{MakePayment} = \oplus(\mathsf{MakecashPayment}, \mathsf{MakeCreditPayment}, \mathsf{MakeCheckPayment})$ as an example to show the choice connector in Fig. 40. The state machine diagram of MakePayment is given in Fig. 41, in which the three final states can be combined into one state.

Now the use case Process Sale can be modelled as $*(\mathsf{CaskDesk}; \mathsf{MakePayment})$. The component diagram is then the same as the bottom diagram in Fig. 38 but with component PayByCash being replaced with the choice composite component MapkePayment, and its state machine diagram is given in Fig. 27 of Part II. Its component sequence diagram is given in Fig. 42, as an example of a general component sequence diagram.

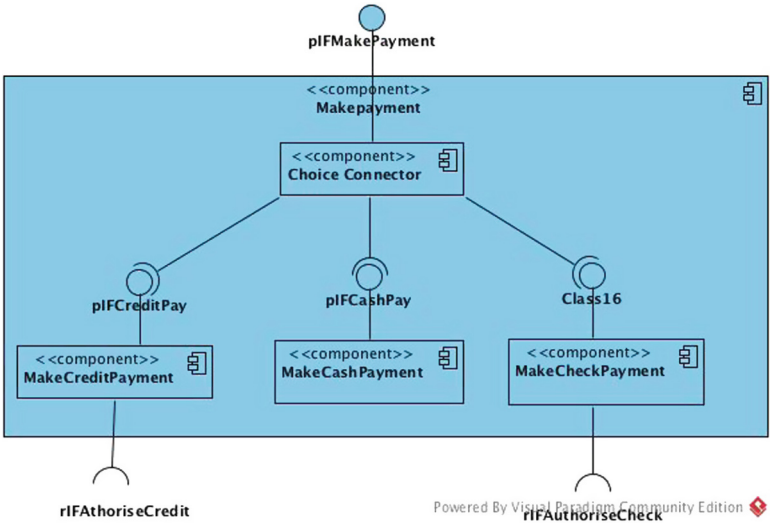


Fig. 40. Choice connector

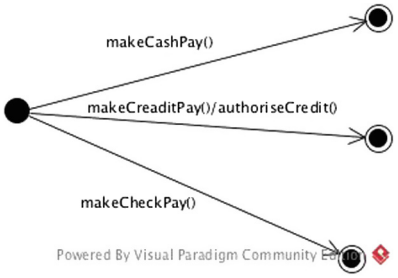


Fig. 41. State diagram of choice connector

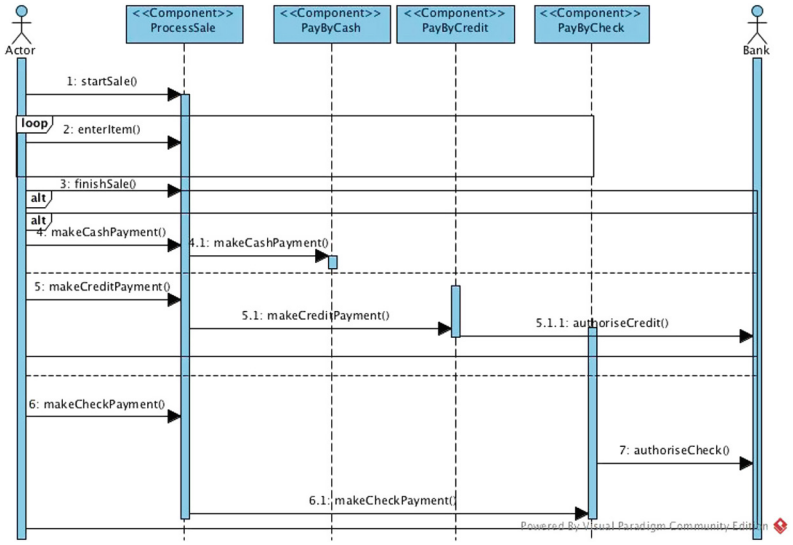


Fig. 42. General component sequence diagram

Parallel Composition. The parallel composition operator $C_1 \| C_2$ for general components C_1 and C_2 is complicated when the required interface of one component, say C_1, shares common operations with the provided interface of another. In this case, C_1 can use operations provided by component C_2 and the behaviour of the parallel composed component may suffer deadlocks and divergences. To handle this case in general, advanced semantic theory is needed. We restrict ourselves to the case when the provided interfaces of C_1 and C_2 do not contain operations in their required interfaces so that feedback method invocations are avoided. The provided interface and required interface of $C_1 \| C_2$ are the union of the provided interfaces and the union of the required interfaces of C_1 and C_2, respectively. The state machine diagram of $C_1 \| C_2$ is defined from the state machine diagrams of C_1 and C_2 as follows

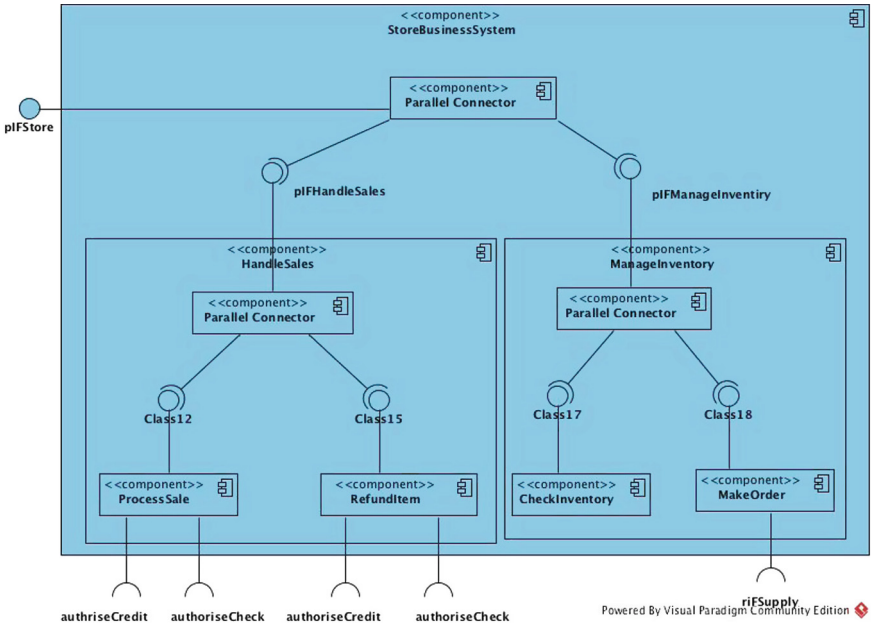


Fig. 43. Parallel connector and an architecture model of a store business

- a pair (s_1, s_2) of states of C_1 and C_2 is a state of $C_1 \| C_2$;
- if a transition s_1 to s_1' by an action a_1 is a transition in the state diagrams of C_1, the transition from (s_1, s_2) to (s_1', s_2) by action a_1 is transition in the state machine diagram of $C_1 \| C_2$;
- if a transition s_2 to s_2' by an action a_2 is a transition in the state diagrams of C_2, the transition from (s_1, s_2) to (s_1, s_2') by action a_2 is transition in the state machine diagram of $C_1 \| C_2$;
- if a transition s_1 to s_1' by an action a is a transition in the state diagrams of C_1 and a transition s_2 to s_2' by an action a is a transition in the state diagrams of C_2 the transition from (s_1, s_2) to (s_1', s_2') by action a is transition in the state machine diagram of $C_1 \| C_2$.

This definition is illustrated in Fig. 44.

Note that $\|$ is commutative and associative, thus it can be used to compose an arbitrary number of components, and $C_1 \| C_2 \| C_3$ can be written $\| (C_1, C_2, C_2)$. Consider use cases Process Sale, Refund Item, Make Order, and Check Inventory. The first two use case are about handling sales and the last two are about inventory management. A component HandleSales = ProcessSale$\|$RefundItem and a component InventoryManagement = CheckInventory$\|$MakeOrder. The (restricted version) StoreBusinessSsystem = HandleSales$\|$InventoryManagement can be defined. We introduce a component called **parallel connector** to represent the component diagram of a parallel composite component. The component diagram of StoreBusinessSsystem is shown in Fig. 43.

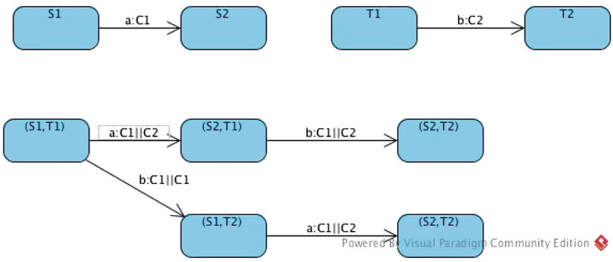


Fig. 44. Parallel composition of state diagrams

Pipeline Connector. Another often used composition of components is "plugging" an operation $m()$ provided by one component, say C_1, to a required interface of a component, C_2, which requires the operation"$m()$". For example, if Process Sale also causes changes to the stock of products, it can call an provided operation of the Managing Inventory use case instead of designing an operation in the component for Process Sale. Notice that Managing Inventory is mostly likely to operate on the Store database system. The **pipeline operator** is defined such that

- $C_1 >> C_2$ is defined if the provided interface of C_2 is disjoint with the required interface of C_1;
- when $C_1 >> C_2$ is defined, its provided interface is the union of the provided interfaces of C_1 and C_2 minus the required interface of C_2 (because any provided operations of C_1 that is also required operations of C_2 are plugged together) and its required interface is the union of the required interfaces of C_1 and C_2 minus the provided interface of C_1 (because any required operations of C_2 that are also provided operations of C_1 are connected);
- the behaviour $C_1 >> C_2$ is similar to the parallel composition $C_1 \| C_2$, but for a transition from s_1 to s_1' by a provided operation a in C_1 and a transition from s_2 to s_2' by $b/a{\cdot}tr$ that requires operation a in a required operation of C_2, there is a transition from (s_1, s_2) to (s_1', s_2') by the action b/tr that does not require any operation in $C_1 >> C_2$. Here tr is a sequence of required operations of C_1. This definition is illustrated in Fig. 45.

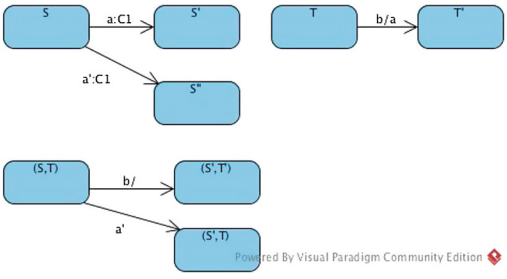


Fig. 45. Plugging state machine diagrams

The connectors used to realise abstract composition operators are open components and the pipeline operator is used to plug the components being connected to the connector. Also, both the parallel composition and pipeline operators can be defined by a general parallel composition, but the semantics of the general parallel composition requires strong theoretical background to comprehend.

Renaming and Restriction Operators. To support reuse of (models of) components, operations of interfaces often need to be renamed. For this, we define that two signatures $m(x; y)$ and $n(u; v)$ (syntactically) **matchable** if x and u are of the same type and y and v are of the same type. In this case the two signatures also called of the **same type**. Given a component C and an operation $n()$ that is of the same type of an operation $m()$, $C[m/n]$ is component obtained from C by replacing the interface operation $m()$ of C with $n()$. We call is a renaming function $[m/n]$.

In fact, renaming $C[m/n]$ does not required m to be an interface operation of C and in this case the renaming function has no effect and $C[m/n]$ is the same as C. The use of renaming includes avoiding conflicts of names when composing components, and in particular renaming a provided operation of C_1 with the name of a required operation of C_2 when plugging C_1 to C_2. In component diagrams, renaming connectors are shown in Fig. 46.

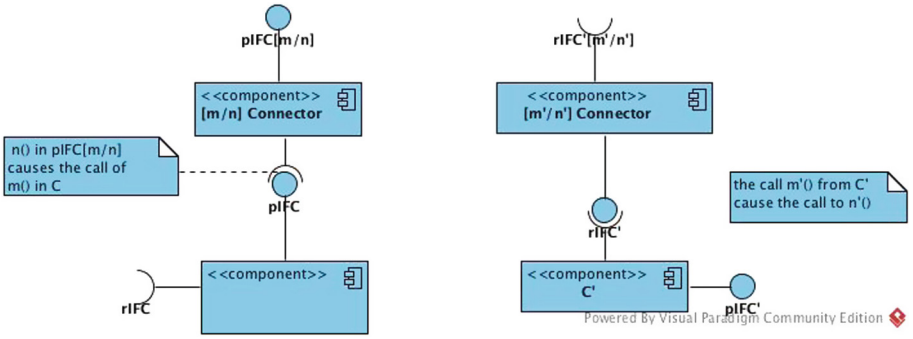


Fig. 46. Renaming provided and required operations

Another useful operator on components is to restrict the access to some of the provided operations. Given a component C and M is a set of operation signatures, $C\backslash M$ is the component obtained from C after **restricting** or **hiding** the operations in M from being used for interaction with the actors, that is

- The provided interface of $C\backslash M$ is the set operations in the provided interface of C minus the operations in M;
- The required interface of $C\backslash M$ is the required interface of C;

– The state machine diagram of $C\backslash M$ is obtained from that of C by removing all transitions made by operations in M.

Note that we do not restrict operations in required interfaces of components. We use a component called the **hiding connector** to realise the restriction operator, as shown in Fig. 47.

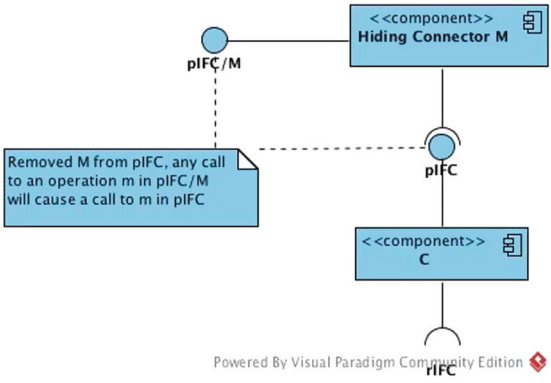


Fig. 47. Hiding connector

Relation to the rCOS Formal Method. Theoretical frameworks, such as CSP [31], CCS [66], I/O automata [64] and Statecharts [24] show the importance of the composition operators in concurrent and distributed computer systems, sound and complete algebraic theories are also developed, but they do not deal well with the complex issues in object-oriented, component-based and interface contract driven development approach. We have spent quite some research effort in the development of the rCOS semantic theory [11,15,21,37,58,91], but large gaps still remain.

11 Object-Oriented Design of Components

The models discussed in Part II on requirements analysis focus on abstract, global and external functionality and behaviour. They are seen from the application domain point of view with a users' perspective. In Sect. 10.1, we transformed the use cases into *components* interrelated through their *interfaces*. The contract of an interface operation specifies what the operation does in terms of preconditions and postconditions, but it does not transformed how software objects are going work collectively to fulfil the contract. We need to transform the contracts into algorithms of object interactions and operations on object states. From the algorithms, program code can be easily developed or even automatically generated.

The essential technique is to decompose the functionality of interface operations into *responsibilities* of related objects, and design the object interactions by *assigning* (or delegating) the responsibilities to appropriate objects. This section studies the general principles for responsibility assignment which are called *design patterns for object responsibility assignment* (GRASP) [42]. Before we introduce the concept of object responsibility, we first discuss the UML diagrams for representing object interactions.

11.1 Object Sequence Diagrams

The UML defines two kinds of interaction diagrams, either of which can be used to express behaviour of object interactions: *object sequence diagrams* and *collaboration diagrams*. There are CASE tools to automatically translate interaction diagrams of one kind to interaction diagrams of another. We use object sequence diagrams, which are similar to component sequence diagrams (and use case sequence diagrams).

If we treat each object as a component and other objects as external actors, an object sequence diagram is a component sequence diagram. However, as show in [45], objects and components are different in nature and not all objects can be treated as components. We are not going to introduce the precise syntax of the UML object sequence diagrams using the UML meta model definition or the abstract syntactic definition in formal languages. Instead, we will show how different object-oriented decompositions mechanisms can be represented in object sequence diagrams.

Decomposition by Sequence of Object Method Invocations. We often need to decompose the functionality (i.e., the grand responsibility) of an operation specified by its contract into a sequence of method invocations to methods number of objects. These methods represent sub-functionalities (or partial responsibilities). We represent this form of decomposition by the object sequence diagram in Fig. 48. It shows that method $m(in; out)$ is decomposed into a sequence of operations.

```
Class B:: m(in;out) {c.m1(in1;out1); c.m2(in2;out2); c.m3(in3;out3)
                    }
```

In an object sequencing diagram, as commented in Fig. 48,

- **instances** of classes are written in boxes,
- a **directed link** between two objects represents an instance of an association between their classes and visibility,
- a **message** represents an invocation of a method of the target object (server) from the source object (client), and
- the messages are numbed to represent the order in which they are executed.

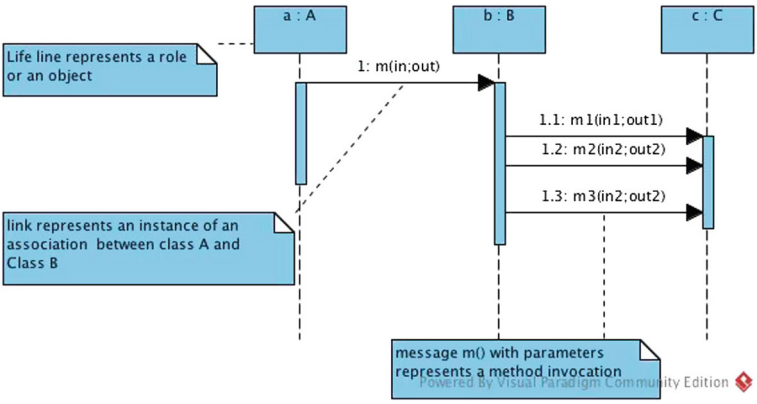


Fig. 48. Decomposition by sequence of method invocations

Note that

1. a message can be passed between two objects only when the two objects are linked in the current state, and
2. two objects can only be linked by an association of the classes of the objects - type correctness.

We also write a message *message*(*parameter:parameterType*; *return: returnType*) with a return parameter in a similar style to Java as follows:

```
return : = message(parameter: parameterType): returnType
```

Recursive Method Invocation. It is often the case that an operation of a component or a method of an object is decomposed into recursive or nested method invocations as shown in Fig. 49, which represents the method decomposition.

```
Class B:: m() {o2.m1()
               }
Class C:: m1(){o3.n1();03.n2()
               }
```

As an example, we decompose *makeCashPay*(*amount : Quantity*) in the way shown in the object sequence diagram in Fig. 50. The message *Create*() represents the call for the constructor of the target class. In an object-oriented programming language, such as Java, *makeCashPay*(*amount* : *Quantity*) is coded as

```
Class CashDsk::makeCashPay(amount:Quantity){s.makeCaskPay(amount)
                                            }
Class Sale::makeCashPay(amount:Quantity){p: =new CashPayment(amount)
                                        }
```

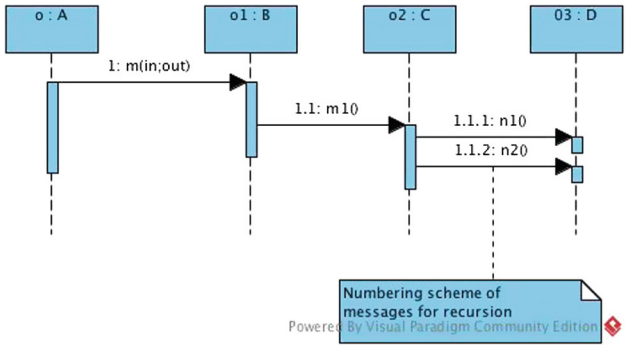


Fig. 49. Decomposition by recursive method invocations

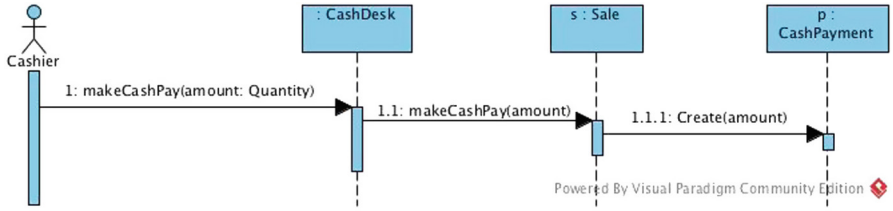


Fig. 50. A design of *makeCashPay* operation

As in component sequence diagrams, loops and choices are provided in object sequence diagrams, but only conditional choices are considered (to model case statements and **if-then-else** statements). However, here we need to discuss messages to **container objects**, which are also called **multiobjects**. The role of an association is modelled as a multiobject if its multiplicity in the association is more than one. For example, Figs. 18 and 21 in Sect. 7 of Part II show that a Sale instance aggregates a set of SaleLineItem instances. In the design, a Sale instance is linked to a multiobject of SaleLineItem by the aggregation associations. We use a stereotype $\langle\langle Set \rangle\rangle$ to represent a multiobject.

Messages to Multiobjects. There are two kinds of messages to a multiobject. A message of the first kind is sent to the multiobject as a single object instead of a broadcast to each of the member objects contained in the multiobject. For example, if we want to check the size of a Sale instance, i.e.. the number of SaleLineItem instances in the Sale instance, we need to send the multiobject $\langle\langle Set \rangle\rangle$ Sale a method call, say *size*(). This is shown in Fig. 51. In Java, a multiobject is often implemented by a variable of **vector type**.

When a method needs to be broadcast to each element in an multiobject, an iteration of a method call to the multiobject to extract links to each individual object, following by a message sent to each individual object using a temporary link (or reference/pointer in object-oriented programs). For example, when we

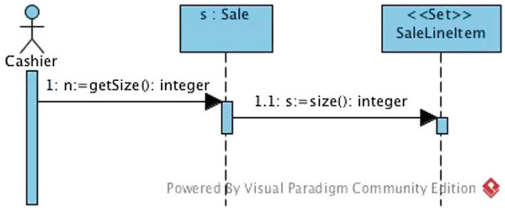


Fig. 51. Single message to a multiobject

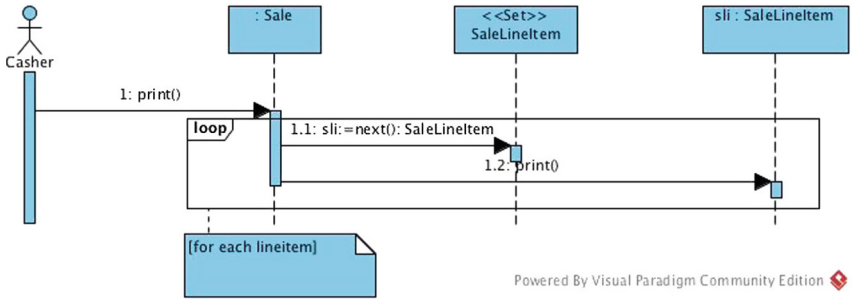


Fig. 52. Message to each element in a multiobject

print out a Sale instance, we need to print each SaleLineItem of the Sale. This is shown in Fig. 52.

Once the object sequence diagrams are created for all the interface operations of the architectural components, we actually have obtained a design model from which program code can be generated. Thus, the question is how we systematically constructed the object sequence diagrams from the model of the component-based architecture design. To this end, we introduce the design patterns for general principles in assigning responsibility (GRASP).

11.2 GRASP: Patterns of Assigning Responsibilities to Objects

Experienced object-oriented developers build up both general principles and idiomatic solutions called patterns that guide them in the creation of software. Design patterns are most popularly promoted by the Gang of Four in their book [23]. A **pattern** is a named problem/solution pair that can be applied to new context, with advice on how to apply it. We introduce five design patterns which are well-known as GRASP - General Responsibility Assignment Software Patterns (or Principles), consisting of guidelines for assigning responsibility to classes and objects in object-oriented design.

Responsibilities of Classes and Objects. Classes in the conceptual class models presented in Sect. 7 in Part II do not have methods. They have not been

assigned responsibilities related to the use cases. In other words, we have not decided what they need to do to contribute to the realisation of the use cases. Object-oriented design of a component analyses what a class in the class model of the component is able to do and decide what it should do to contribute to the realisation of the contracts of the interface operations.

A **responsibility** of an object is a contract or obligation of the object. Responsibilities are related to the obligations of objects in term of their behaviour. There are in general two types of responsibilities:

1. **Doing responsibilities:** these are about the actions that an object can perform, including
   - doing something itself (such as changing its state),
   - initiating an action or operation in other objects (such as calling methods of other objects), and
   - controlling and coordinating activities in other objects (receiving method invocation and passing data to other objects).
2. **Knowing responsibilities:** these are about the knowledge an object maintains including
   - knowing about its encapsulated data. E.g., a product specification knows the prices of the product,
   - knowing about related objects, e.g., the Catalog knows all the product specifications and a student knows the modules he or she takes, and
   - knowing about things it can derive or calculate, e.g., if a Student knows his or her date of birth he or she knows his or her current age.

It is important to note that the knowing responsibilities of a class are clearly represented in the conceptual class model in terms of attributes of classes and associations between objects; and the doing responsibilities of a class are determined by its knowing responsibility, i.e., what an object can do depends on what it knows. Deciding the doing responsibilities of a class requires analysis and deduction, and thus is more challenging.

Object orientation supports the principle of information hiding, i.e., *data encapsulation*. All information in an object-oriented system is stored in its objects and can only be manipulated when the objects are asked to perform some actions. In order to use an object, we only need to know the interface that consists of the public methods and public attributes of the object.

The general steps to design the object sequence diagrams of a component is described as follows.

1. Start with the responsibilities which are identified from the use cases and conceptual models (knowing responsibilities), and the contracts of the interface operations of the component.
2. Assign these responsibilities to objects, then decide what the objects needs to do to fulfil these responsibilities in order to identify further responsibilities which are again assigned to objects.
3. Repeat these steps until the identified responsibilities are fulfilled and a object sequence diagram is completed.

For example, we may assign the responsibility of knowing the date of a Sale instance to the instance itself (a knowing responsibility), and the responsibility for printing a Sale instance to the instance itself (a doing responsibility).

Responsibilities of a class are implemented by programmed methods of the class which either acts alone or collaborates with other methods and objects. For example, the class Sale might define a method *print*() that prints an instance. To fulfil this responsibility, object Sale has to collaborate with other objects by sending a message to each of SaleLineItem objects contained in the Sale asking them to print themselves.

Using the UML, responsibilities are assigned to objects when creating an object sequence diagram, and the object sequence diagram represents both of the assignment of responsibilities to objects and the collaboration between objects for their fulfilment. For example, Fig. 52 in the previous subsection indicates that

1. Sale objects have been given the responsibility to print themselves, which is invoked with a message *print*() to Sale.
2. To fulfil this responsibility, Sale needs to collaborate with the SaleLineItem objects it contains, asking them to print themselves, thus each saleLine having a method *print*().

The Five GRASP Patterns: Each of these principles or solutions describes a problem to be solved and a solution to the problem. We follow the style of presentations of patterns see in Larman's textbook [42].

Pattern Name :	The name given to the patterns for easy reference.
Solution :	Description of the solution of the problem.
Problem :	Description of the problem that the pattern solves.

We now introduce the five patterns **Expert Pattern**, **Creator**, **Low Coupling**, **High Cohesion** and **Controller**. We will use some responsibilities of Process Sale use case as illustrating examples. Attention should be paid to how the conceptual class model and contracts component interfaces (or use case operations) are used in identification of responsibilities of classes.

Expert Pattern. We start with **Expert Pattern**

Pattern Name:	Expert.
Solution:	Assign a responsibility to the information expert - the class which has information necessary to fulfil the responsibility.
Problem:	What is the most basic principle by which responsibilities are assigned in object-oriented design.

The key of Expert pattern is to identify the *information expert* from the conceptual class model for a given responsibility. Consider ProcessSaleWithCashPay component for example, it has the responsibility to calculate the total of the Sale. This responsibility needs to be assigned to a class. When we assign responsibilities, we had better to state the responsibility clearly:

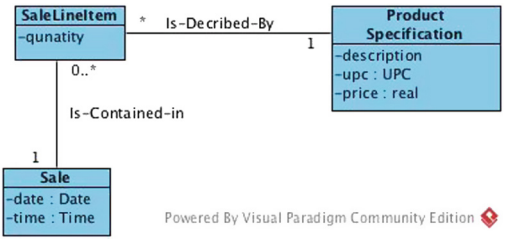


Fig. 53. Information expert for the total of Sale

Which object should be responsible for knowing the grand total of the Sale instance?

By pattern Expert, we should look for the class of objects which has the following information that is needed for calculating the grand total of the Sale:

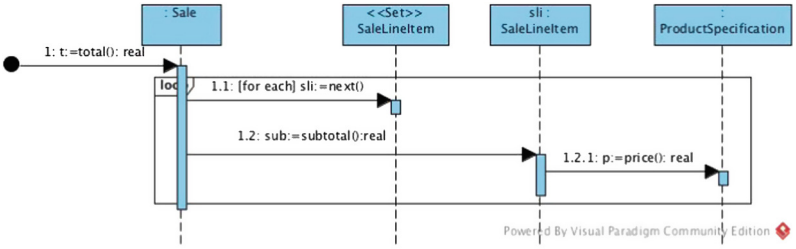
- all the SaleLineItem instances of the Sale instance, and
- the sum of their subtotal of SaleLineItem instances.

Looking at the conceptual class diagram in Fig. 18 in Sect. 7 of Part II, we extract the partial class diagram in Fig. 53. This class diagram shows that only Sale knows the above two pieces of information for calculating the grand total of the Sale. Thus by Expert, Sale is the correct class for this responsibility.

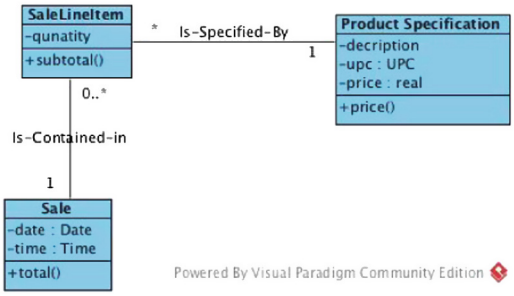
We assign the responsibility of returning the total, represented by the method *total*(), to class Sale. This is shown in Fig. 54a as a message $1 : total()$ to object :Sale. Next the functionality is decomposed into responsibilities of *knowing the subtotal* of each SaleLineItem of Sale. And each SaleLineItem is the information expert of its subtotal. This is represented by the responsibility assignment of $1.2 : sub := subtotal()$ in Fig. 54a. Further, the subtotal functionality is decomposed into getting the price of the item and the quantity of item. The information expert of knowing the price is ProductionSpecification and the information expert of quantity is the SaleLineItem itself. This leads to a further responsibility assignment $1.2.1 : p := price()$ to ProductSpecification. The construction of the object sequence diagram shows that while looking for an information expert, further decomposition of responsibilities and looking for sub-information experts are often needed.

Notice the message $1 : t = total()$ to :Sale is not from the actor Cashier. This is because it is not an actor to trigger the operation *total*(). Instead the information of *total*() is directly picked up by the GUI object for display. We do not discuss GUI design and how GUI objects are linked to the application objects.

Along the creation of object sequence diagrams, methods representing responsibilities are assigned to the classes. This is represented in the partial *design class diagram* in Fig. 54b. A **design class diagram** is a class diagram which contains classes and operations of classes, as well as attributes of classes



(a) Object sequence diagram for design of *total*()



(b) Design class diagram

Fig. 54. Design of *total*()

and associations among classes. The design of object sequence diagrams for operations thus also transforms the conceptual class diagram to a corresponding design class diagram.

To be precise, a class diagrams also contains information about directions of *navigability* (or *visibility*) of associations and *dependencies* among classes. Dependencies represent the directions of messages and flows of parameters of methods. It seems that Visual Paradigm does not support visual presentation of navigability, but the object sequence diagrams show the directions of messages anyway.

The pattern Expert is the most fundamental principle in object-oriented design. We will see it is used gain and again in the following subsection about the design of the operations of Process Sale with Cash Payment. Expert "expresses the common intuition that objects do things related to the information they have - fundamentally, objects do things related to information they know" [42]. A good analogy is that if a corporation is to produce the annual financial report of its business, it is obvious that this responsibility should be given to the chief financial officer, and then the chief financial officer would give the responsibilities for producing the different parts to different divisions of his or her departments which has the relevant information and data.

The use of Expert also maintains encapsulation, since objects use their own information to fulfil responsibilities. This also implies low coupling, which means

no extra links needed to be formed apart from those which have to be there. Low coupling implies high independency of objects that leads to more robust and maintainable systems.

Creator. The creation of objects is one of the most common activities in an object system. Consequently, it is important to have a general principle for the assignment of creation responsibilities.

Pattern Name: Creator.
Solution: Assign class B the responsibility to create an instance of a class A (B is a creator of A objects) if one of the following is true:

- B aggregates A objects.
- B contains A objects.
- B records instances of A objects.
- B closely uses A objects.
- B has the initialising data that will be passed to A when it is created (thus B is an expert with respect to creating A objects).

Problem: What should be responsible for creating a new instance of some class?

Consider the postconditions of *enterItem*(*upc:UPC*, *qty:Quantity*) of component ProcessSaleWithCashPay. We identify the responsibility "creating a SaleLineItem instance". Look into partial class diagrams in Fig. 53 as a part in the conceptual model in Fig. 18 in Sect. 7 of Part II. A Sale instance aggregates many SaleLineItem objects, Creator suggests Sale is a good candidate to be responsible for creating the SaleLineItem instance. The responsibility is then delegated to SaleLineItem to create itself. Notice that in all object-oriented programming languages a class always has "constructor" for creating objects of the class. However, the creator of an object *o* is the object which calls the constructor method of the class of *o*. This leads to a design of object sequence diagram in Fig. 55. The assignment of creating a SaleLineItem instance to Sale also identifies a method *makeLineItem*(*upc:UPC*, *qty: Quantity*) of class Sale, which should be added to the design class diagram in Fig. 54b.

Consider *makeCashPayment*(). Its postconditions indicates the responsibility of "creating a CashPayment instance". Which object should be the creator of the CashPayment? Creator suggests both Sale, which is closely related to CashPayment, and CashDesk, which represents the component of the sale handler handling payments too, can be candidates of the creator. Recall that contract of *makePayment*() also has a postcondition for the CashPayment created to be linked to the Sale. We have two alternative designs in Fig. 56a, b respectively.

Compare the two designs, the one in Fig. 56a is much simpler, and the link between :Sale and :CashPayment is naturally established through creation. Furthermore, look into the conceptual class diagram in Fig. 18 in Sect. 7 of Part II,

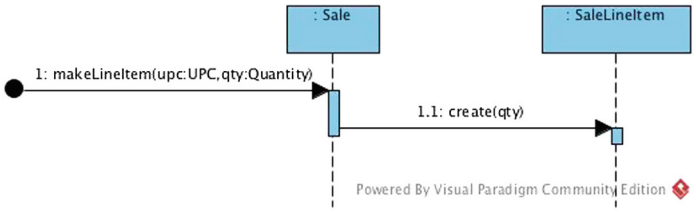
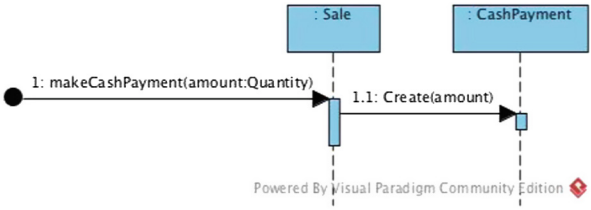


Fig. 55. Example of Creator pattern

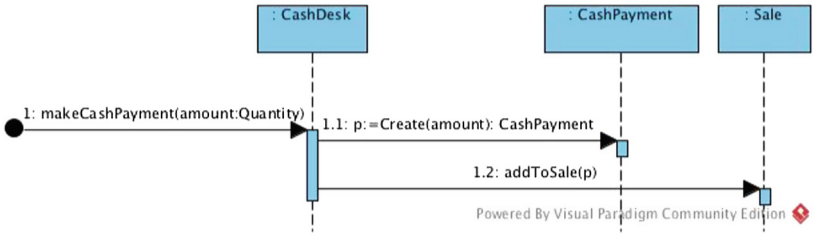
we find that there is no direct association between CashDesk and CashPayment. The message 1.1 : *p=Create*() from :CashDesk to :CashPayment cannot be sent unless we add an association. This also motivates the **Low Coupling** pattern.

Low Coupling. The design in Fig. 56a is preferable over the one in Fig. 56b because it does not need an extra link formed between CashDesk and CashPayment. Thus, the former keeps the "coupling" low. In object-oriented programming languages such as C++, Java, and Smalltalk, common forms of coupling from a class A to a class B to include:

- A has an attribute which refers to a instance of B, being represented by an association in a class diagram;
- A has a method which



(a) Creator of *CashPayment*



(b) An alternative Creator of *CashPayment*

Fig. 56. Two alternative Creators of *CashPayment*

  - has a reference (or pointer) to an instance of B,
  - calls a method of B itself by any means including a parameter,
  - local variable of type B, or
  - the object returned from a message being an instance of B;
- A is a direct or an indirect subclass of B, shown as specialisation relation in a class diagram.

Classes whose objects have too many links to other objects are not easy to be reused, as to reuse such a class the related class should be used. Components with classes of high coupling are difficult to maintain too, as changing to such a class would cause changes to the related classes. For better reusability and maintainability of components, the pattern of Low Coupling suggests that coupling should be kept low.

Pattern Name :	Low Coupling.
Solution :	Assign a responsibility so that coupling remains low.
Problem :	How to support low dependency an increased reuse?

The design in Fig. 56a conforms to Lower Coupling.

High Cohesion. Cohesion is a measure of how strongly related and focused the responsibilities of a class are. A class with high cohesion has highly related functional responsibilities, and it is not overloaded with a large amount of responsibility. A class with low cohesion is undesirable as it suffers from the following problems: *hard to comprehend, hard to reuse*, and *hard to maintain*, and thus it is prone to errors.

Pattern Name :	High Cohesion.
Solution :	Assign a responsibility so that cohesion remains high.
Problem :	How to keep complexity manageable?

According to pattern High Cohesion, the design in Fig. 56a is preferable over the one in Fig. 56b also. This is because CashDesk is the interface object of ProcessSale. Thus, it is naturally and primarily responsible for handling the provided operations of the component (see pattern Controller below), and the responsibility for creating a payment is not logical related to the interface operations. The benefits from the use of the High Cohesion Pattern include:

- clarity and ease of comprehension of the design is increased,
- maintenance and enhancements are simplified,
- low coupling is often supported,
- supports reuse and easy maintenance.

Controller. To create a design of a component, we need to assign the responsibility of handling the operations of component's provided interfaces. The question is which object should be responsible for receiving the provided interface

operations from its actors, and then delegates the responsibilities specified by the contracts of the operations to further objects.

Pattern Name: Controller.
Solution: Assign the responsibility for handling a system (input) event to a class representing one of the following choices:

- Represents the "overall system" (facade controller)[18].
- Represents the overall business or organisation (facade controller).
- Represents something in the real-world that is active (for example, the role of a person) that might be involved in the task (role controller).
- Represents an artificial handler of all system (input) events of a use case (or 'operations of provided interfaces of a component'), usually named "'⟨*UseCaseName*⟩*Handler*"' (use-case controller or 'component interface handler').

Problem: Who should be responsible for handling a system input event (or 'the operations of the provided interface of the component')?

Notice that we added some alternative interpretations in the single quotes, such as 'component interface handler' and 'operations of provided interfaces of a component', in relation to our component-based design. This pattern also suggests to use the same controller class for all system input events in the same use case. It also implies that a controller is a non-user interface object responsible for handling provided operations of the component.

With our component based model of architecture, it is become more obvious when deciding the controller classes as they are just the classes to implement the provided interfaces, and we have one controller class for each interface. Consider the use case Process Sale with Cash Payment (also component ProcessSaleByCash), we have given the specification of the contracts of the operations *startSale*(), *enterItem*(), *finishSale*(), and *makeCashPayment*() of the use case. We need to assign these operations to a controller class. According to the pattern of Controller, we have the following candidates

1. CashDesk: represents a component interface handler[19].
2. Store: represents the overall business or organisation.
3. Cashier: represents something in the real-world (such as the role of a person) that is active and might be involved in the task.
4. ProcessSaleHandler: represents an artificial handler of all the operations of the use case[20].

[19] This does not explicitly indicated by Controller pattern, but it is an object that Cashier actor uses to handle the operations. It also represents the Cash Desk PC.
[20] The choice of CashDesk can also be seen as an artificial handler.

The decision on which of these four is the most appropriate controller is influenced by other factors, such as cohesion and coupling. In the next subsection, we will use CashDesk as the controller to show the design of the provided operations of the component ProcessSaleByCash. The reason is that it is already in the conceptual class diagram with associations naturally identified, and this object does not have much other functionality to carry out.

The relation between the controller class and the component provided interface is that former implements the latter. Thus, we have class CashDesk implements pIFProcessSaleCash and in Java this corresponding the class definition below

```
Interface pIFProcessSaleCash{
   startSale();
   enterItem();
   finishSale();
   makecashPayment()
 }

Class CashDesk implements  pIFProcessSaleCash {
    startSale(){as designed in the next subsection};
    enterItem(){as designed in the next subsection};
    finishSale() {as designed in the next subsection};
    makeCashPayment(){as designed in the next subsection}
```

11.3 Design Component ProcessSaleWithCashPay

We first discuss the design of Start Up operation that creates the initial object structure of the system so that all business use processes reply on. Recall the post conditions of *StartUp* operation:

1. A Sore, CashDesk, Catalog and ProductSpecification were created (instance creation).
2. ProductSpecification was associated with Catalog.
3. Catalog was associated with Store.
4. CashDesk was associated with Store.
5. Catalog was associated with CashDesk.

This contract specifies the initialisation of the execution when an application is launched. The principle is to create an initial domain object first, and then through which the other objects that need to be created in the StartUp operation are created. The problem of how to create the initial domain object is dependent upon the object-oriented programming language and operating system. The following is, for example, how *StartUp*() is done using Java applet.

```
public class ProcessSaleApplet extends Applet {
    public void init() {
```

```
    cashDesk := store.getCashDesk();
    }
// Store is the initial domain object.
// The Store constructor creates other domain objects
    private Store store := new Store();
    private CashDesk cashDesk;
    private Sale sale;
  }
```

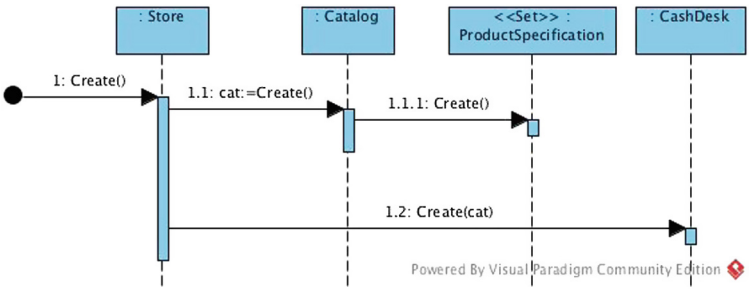


Fig. 57. Design of *StartUp*()

The object sequence diagram for the design of *StartUp*() operation does not show its controller so as to abstract the platform details away. Instead, the object sequence diagram will start from the creation of the initial domain object, and in this case it is **Store**. The design is shown in Fig. 57. After the *StartUp*() operation is executed. Some management use cases can be carried out to add product specifications to the Catalog instance cat. Notice that the creation of the Catalog instance cat also creates the (empty) set of ProductSpecification instances; and the creation of the CashDesk uses cat as a parameter so that the CashDesk is linked to the Catalog. Therefore the postconditions of *StartUp*() are all met by the design. Now we design the operations of Process Sale with Cash Payment.

Design of startSale(). The contract of *startSale*() assumes the preconditions of the existence of the objects Store, CashDesk and Catalog that were created by the *StartUp*() operation. The postcondition is simply to create a new Sale instance. By Controller, CashDesk it is also the Creator of the new Sale instance. The design is given in Fig. 58.

Design of enterItem(). This operation carries most of the complexity, compared to the other operations of the use case. Its contract assumes the existence of the **CashDesk** instance, the Catalog instance and its contained ProductSpecifications. The operation has input parameters upc:UPC and qty:Quantity of the item to

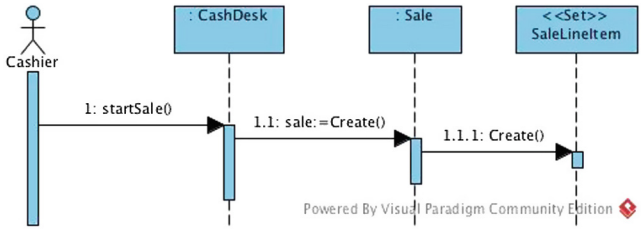


Fig. 58. Design of *startSale*()

create. The contract also assumes the precondition that product identity upc is valid, i.e., can be found in the Catalog. The responsibilities are specified in the postconditions of the contract, i.e., to create the new SaleLineItem instance and associate it to the Sale. Here are the ideas of the design

1. by Controller, *enterItem*() is assigned to Cashdesk;
2. by Creator, Sale is the creator of SaleLineItem;
3. to create the SaleLineItem instance, however, the product specification of the item and price need to be obtained and passed to the constructor of SaleLineItem; and
4. by Expert, Catalog is the information expert to find the specification for the given upc.

This analysis leads to the design given in Fig. 59.

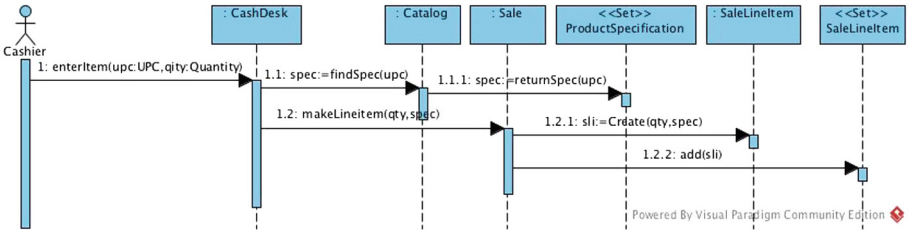


Fig. 59. Design of *enterItem*()

Creating the object sequence diagrams also improves the understanding of the preconditions of an operation. For example, the precondition that the CashDesk instance is linked with Catalog instance, establish by *StartUp*(); and that the existence of the Sale and the existence of the ProductSpecification, established by *startSale*() and some management use cases. On the other hand, an object sequence diagram can also be checked against the contract of the operation, that is all preconditions are checked and postconditions are established.

Design of finishSale(). This is a simple operation and its execution assumes the preconditions established by *startSale*() and *enterItem*(). The postcondition is simply to set the attribute isComplete attribute of Sale. By Controller CashDesk is the controller of *finishSale*(), and Sale is the expert for setting its attribute isComplete to true. The design is shown in Fig. 60.

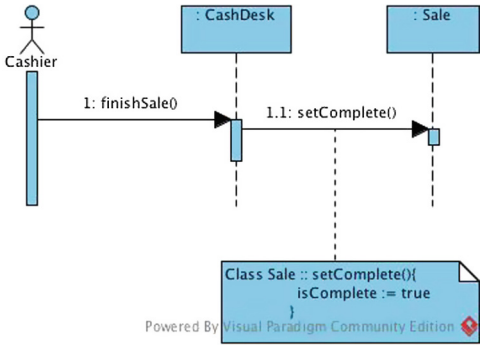


Fig. 60. Design of *finishSale*()

Design of makeCashPayment(). The precondition of this operation is that the isComplete attribute of the Sale is true. This has been established by *finishSale*(). The postconditions include creating of CashPayment, associating it to the Sale and associating the sale with the Store to log the complete sale. The Creator of cashPayment is designed to Sale in Fig. 56a. The expert for logging the complete Sale is Store, as it aggregates (complete) Sale instances. This analysis leads to the design in Fig. 61.

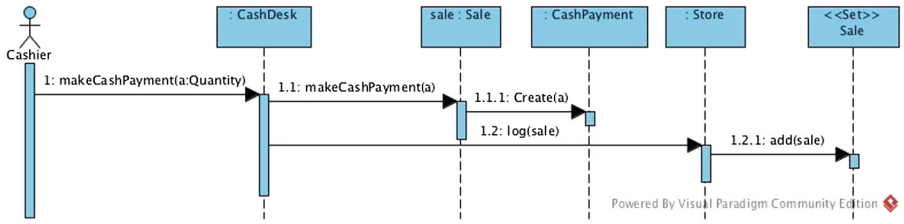


Fig. 61. Design of *makeCashPayment*()

The design of the object sequence diagrams of the operations also contributes significantly to the understanding and identification of the preconditions of the operations that are difficult to identify during the analysis of the use case sequence diagram and the contracts of their operations. The dependency of the

preconditions of an operation on the postconditions established by other operations can be checked using the state machine diagram of the component. For example, the state machine diagram in Fig. 39 for component ProcessSaleWithCashPay is used to guide the above discussion of the preconditions in the design of the operations.

11.4 Design Patterns and Structural Refinement

We now extend the notion of structural refinement from conceptual class models to design class models. The difference is that classes in the latter contain methods. The methods of the classes are invoked by the operations in the provided interfaces of the components according to the object sequence diagrams of the interface operations.

A class model (either conceptual or design) $\mathcal{C}$ is **adequate** for an interface operation if an object sequence for this operation can be constructed using the methods of in the classes. A class model $\mathcal{C}$ is a **structural refinement** of a class model $\mathcal{C}_1$, denoted by $\mathcal{C}_1 \sqsubseteq \mathcal{C}$, if $\mathcal{C}_1$ is adequate for an interface operation, then $\mathcal{C}$ is also adequate for this operation.

The expert pattern is then statement as a refinement rule shown in Fig. 62. We explain the diagram as follows

- a responsibility $S[c.(x)]$ is assigned to class C that contains a sub-responsibility $c(x)$,
- class B is the information expert of the responsibility of $c(x)$, here attributes x can contain role names to refer to associated objects of class B, and
- the responsibility is represented as a method $m()$ and assigned to class B.

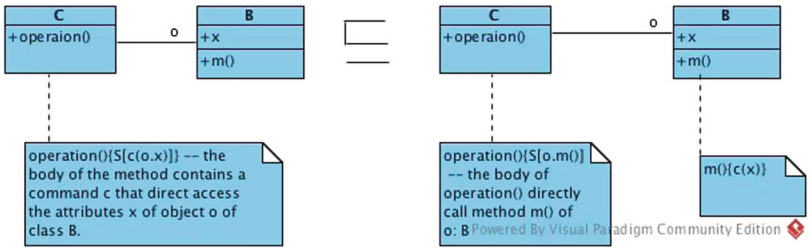
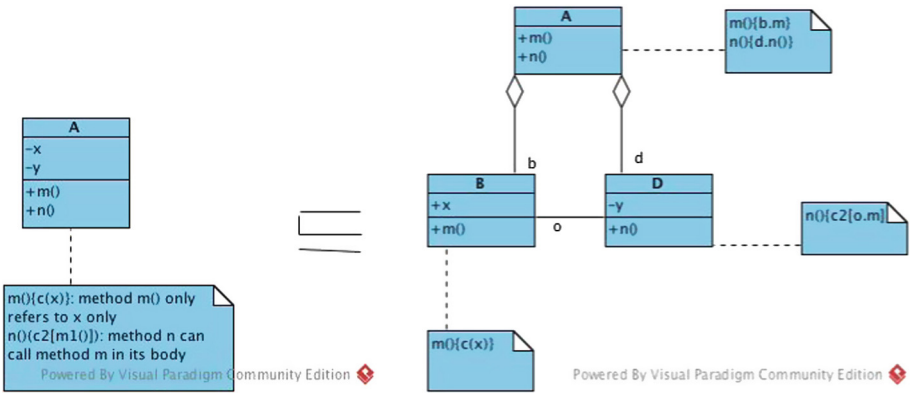


Fig. 62. Expert pattern as a structural refinement

A structural refinement rule for class decomposition is given in Fig. 63, that also reflects High Cohesion. We explained this refinement as follows

- assume class A contains two sets of attributes x and y (including role names of associations with class C);
- class C are given responsibilities represented by methods m and n, and m only refers to attributes x;

– we can decompose A into three classes A, B and D such that B maintains x only and it is assigned with the responsibility m;
– class A is responsible for coordinating the responsibilities of class B and D.



(a) A low cohesion class (b) Refined class model

Fig. 63. Class decomposition

The class model in Fig. 64 also refines the low cohesion class in Fig. 63a, and clearly this model is of lower coupling than the class model in Fig. 63b.

The refinement in Figs. 62, 63 and 64 can be used in the context of any larger class models that contain them. The proposition about refinement in Subsect. 7.8 of Part II can now be extended as: a class model $\mathcal{C}$ is a structural refinement of a class model $\mathcal{C}_1$, if $\mathcal{C}$ can be obtained by one of the following changes made to $\mathcal{C}_1$:

– adding a class,
– adding an attribute to a class,
– adding an association between two classes,
– increasing the multiplicity of a role of an association (that is equivalent to adding attributes at the level of program code),
– promoting an attribute a subclasses to its superclass,
– promoting an association of a subclass to its superclass,
– adding a method to a class, and
– promoting a method of a subclass to its superclass.

Structural refinement supports incremental modelling and design.

11.5 Summary of Design

We have discussed all the operations of Process Sale with Cash Payment, or equivalently the operations of the interface of component ProcessSaleByCash.

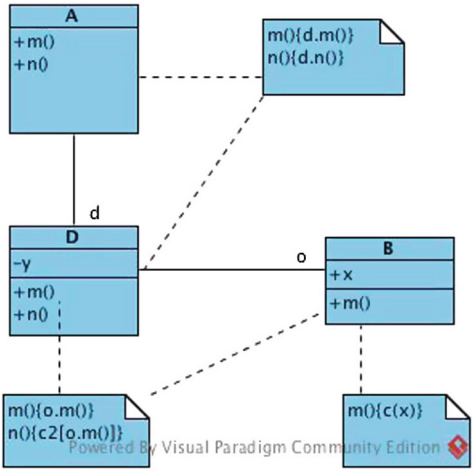


Fig. 64. A refinement of low coupling

The messages in the object sequence diagrams of these operations should be collected and recorded in the classes of the target objects of the messages. After doing these, the conceptual class diagram is transformed into the design class diagram, whose methods and directions of navigability (or visibilities) of the associations of the class are determined by the object sequence diagrams. With the design class diagram, the object sequence diagrams and the state machine diagrams of a component, an object-oriented program (skeleton) for the component can be generated. The skeleton contains the methods of the classes and object interactions as method calls (represented by the messages in the object sequence diagrams) in their bodies, only leaving significant algorithms for manipulation simple attributes of objects need to be coded. The detailed design model and the program skeleton are analysable and validatable against the requirements models.

The GRASP patterns and design are mainly from Larman's textbook [42]. However, we put the discussion in relation to the component-based architecture design given in Sect. 10.1. Furthermore, models of component-based architecture can be obtained at the requirements analysis level by decomposing the use cases. On the other hand, transformations of object sequence diagrams of a component can wrap some objects into a components and abstract away the details of some object interactions inside these wrapped components. These transformations can be supported by an interactive transformation tool [46]. This will further decompose the component-based model of architecture obtained in the component-based architecture design in Sect. 10.1.

Relation to the rCOS Formal Method. The informal object-oriented design presented in this section is the motivation of the development of the object-oriented extension [27] to Unifying Theories of Programs [32]. Based on this semantic theory, a sound and complete calculus of object-oriented structure

refinement presented in [91], which also characterises the expert pattern (Creator is treated as a special case of Expert), low coupling and high cohesion as refinement rules, together with other refactoring rules. The application to the Trading System is shown in [15]. Formalisation of more design patterns can be found in [63]. Code generation from design is also presented in [62].

12 Summary and Future Work

We have presented, through informal but yet precise discussions, a model-driven design method for object-oriented component-based software systems. The theoretical underpinning of this method is formal model-driven method of component and object systems, knows as rCOS [15,27,37] and component interfaces [11,21,26,37,61], and formal semantics and relations of different UML models for requirements models and design [12,25,52,55].

The rCOS methods clearly reflect the model-driven development that *system design is carried out in a process through building system models* to gain confidence in requirements and designs. The process of model constructions emphases on

- the use of *abstraction* for information hiding so as to be well-focused and problem oriented;
- the use of the engineering principles of *decomposition* and *separation of concerns* for *divide and conquer* and *incremental development* and evolution; and
- the use of *formalisation* to allow the *process repeatable* and *artefacts (models) analysable*.

Alao, rCOS proposes a multiple diminutional approach to component-based architecture modelling, as shown in Fig. 65.

- First, it allows models of a component at different levels of abstract, from the top level models of *interface contracts* of components developed through use case analysis and conceptual class modelling, through models of interactions and dynamic behaviours of components, component-based architecture design and object-oriented design of individual components, to models of deployment and implementations.
- At each level of abstraction, a component has models of different viewpoints, including the *class model* (or *data model*), the specification of *static data functionality* (i.e., changes of data states), the *model of interaction protocol* with the environment (i.e., actors) of the components, and the *model of reactive behaviour*. These models of different viewpoints support the understanding of different aspects of the components and support different techniques of analysis, design and verification of different kinds of properties. Moreover, they support the separation of design and verification concerns - models of different viewpoints can be refined separately without affecting models of the other view viewpoints. For example, in the rCOS theory, we have proven that contact of the static functionality of an interface can be refined without

changing its contract of dynamic behaviour, and *vice versa*. Similarly, structure refinement of the class model preserves the specifications of contracts and interaction models.
– A model of a component is hierarchical and composed from models of 'smaller' components that interact and collaborate with each other through their interfaces. Some components can also control, monitor or coordinate other components. These compositions are realised by connector components.

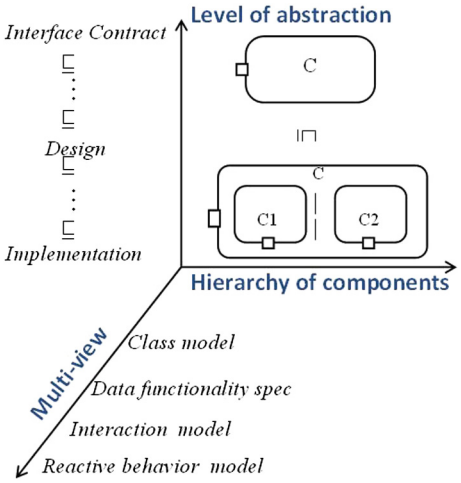


Fig. 65. rCOS modelling approach

We briefly summarise the iterative rCOS model of development process, driven by model constructions and model transformations/refinement, as followings

1. Through use case analyse and conceptual class modelling development a model of component-based architecture (Sects. 5–8).
2. Refine interface operations of components by using design patterns to generate a model object-oriented design, i.e., the collection of object-sequence diagrams and the design class diagram (Sect. 9).
3. Transform the model of object-oriented design to a model of component-based design for integration and maintenance/evolution [46] (this is not discussed in details in this chapter).
4. Realise the interfaces of components using appropriate middlewares, e.g., RMI, CORBA etc. (not covered in the these notes).
5. Code generation performed after Step 3 and Step 4.

An iteration of this process is shown in Fig. 66.

We can see that the rCOS method and development emphasis on the interface-based requirements analysis and design of components. We believe this

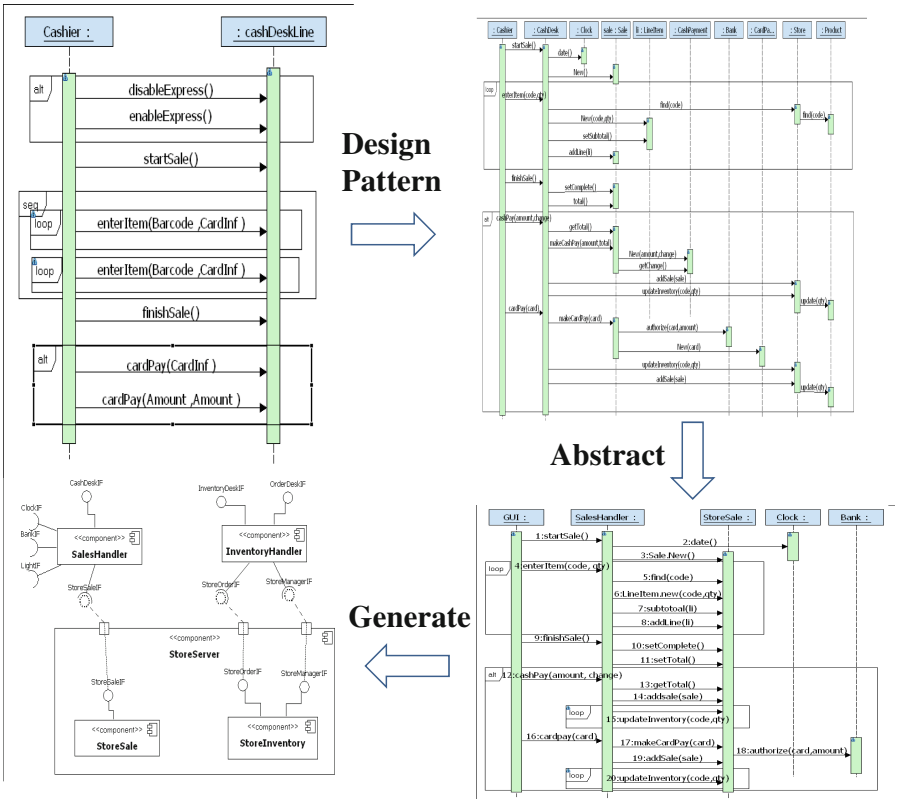


Fig. 66. rCOS development process

is becoming increasingly essential for the maintenance of and development of front end applications from modern complex evolving software-intensive systems [89]. These systems include Internet of Things (IoT) [48], Smart Cities [83] and Cyber-Physical Systems (CPS) [43]. They are becoming major networks of infrastructures for development of applications in all economic and social areas such as health and care health, environment management, transport, enterprises, manufacturing, agriculture, governance, culture, societies and home automation. These applications share a common model of architectures and involve different communication technologies and protocols among the architectural components. The research and applications thus require collaborations among experts with expertise in a variety of disciplines and various skills in software systems development.

The future development of the rCOS interface theory include its extension to models of *physical interfaces* in order to model *cyber-physical components* and their composition [54,72]. This will make the notion of interfaces very general. For example, a piece of wall or a window can be modelled interfaces between the temperatures outside and inside a room. Even the "air" between two sections

of a room can modelled as an interface that transforms the temperature of one section to that of another. However, this general notion of interfaces poses a number of challenges, for example

1. How to develop a model of contract of such interfaces, as it is often the case that there is no known physical laws or functions for defining these interfaces?
2. How to define the formal semantics and the refinement relation between interface contracts?

These are the first significant questions to ask when developing a semantic theory for these CPS components and their compositions. Further challenges including

1. how to develop design techniques and tools,
2. how to combine David Parnas's Four-Variable Model, Michael Jackson's Problem Frames Model, and the Rational Unified Process (RUP) of the use case driven approach systematically into the continuous evolutionary integration system development process?

We believe that model-driven approach is again promising, and techniques and tools of simulation with rich data and machine learning would become increasing important in building the correct models.

Acknowledgement. The development of materials of this chapter started in 1998, and it has been revised ever since through the teaching at the University of leicester during 1998–2001, United Nations University -International Institute for Software Technology (UNU-IIST, Macau) in 2001–2013, and Birmingham City University in 2015, and at many international training schools (mostly supported by the UNU-IIST). We acknowledge the materials in the textbook of Larman [42] as a major source of knowledge and ideas, that have developed in all the versions of the course notes. The extensive research at UNU-IIST on the rCOS method and the development of its tool support have significantly contributed to development of the understanding of theoretical insight of object-oriented design, component-based design, and model-driven development, and their relations. We acknowledge the contribution from Xin Chen, Zhenbang Chen, Ruzheng Dong, He Jifeng, Wei Ke, Dan Li Xiaoshan Li, Jing Liu, Charles Morisset, Anders Ravn, Volker Stolz, Shuling Wang, Jing Yang, Liang Zhao, and Naijun Zhan.

Our special thanks go to Jonathan Bowen and Anders Ravn for their careful reading and constructive comments, without that we could not have made this chapter into its current form.

References

1. Hamilton, M.: The engineer who took the apollo to the moon. https://medium.com/@verne/margaret-hamilton-the-engineer-who-took-the-apollo-to-the-moon-7d550c73d3fa
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)

4. Limet, S., Robert, S., Turki, A.: Controlling an iteration-wise coherence in dataflow. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 241–258. Springer, Heidelberg (2012)
5. Bell, M.: Service-Oriented Modeling: Service Analysis, Design, and Architecture. Wiley & Sons, New Jersey (2008)
6. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Boston (1994)
7. Bowen, J.P., Hinchey, M.G.: Formal methods. In: Gonzalez, T.F., Diaz-Herrera, J., Tucker, A.B.(eds.) Computer Science Handbook, 3rd edn. Section XI, Software Engineering, vol. 1, Computer Science and Software Engineering, Part 8, Programming Languages, Chapter 71, pp. 1–25. CRC Press (2014)
8. Brooks, F.P.: No silver bullet: essence and accidents of software engineering. IEEE Comput. **20**(4), 10–19 (1987)
9. Brooks, F.P.: The mythical man-month: after 20 years. IEEE Softw. **12**(5), 57–60 (1995)
10. Chandy, K.M., Misra, J.: Parallel Program Design: a Foundation. Addison-Wesley, Reading. (1988)
11. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007)
12. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 819–831. Springer, Heidelberg (2007)
13. Chen, Z., et al.: Modelling with relational calculus of object and component systems - rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 116–145. Springer, Heidelberg (2008)
14. Chen, Z., Li, X., Liu, Z., Stolz, V., Yang, L.: Harnessing rCOS for tool support —the CoCoME experience. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 83–114. Springer, Heidelberg (2007)
15. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Sci. Comput. Program. **74**(4), 168–196 (2009)
16. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Fitzgerald, J., Larsen, P.G., Sahara, S. (eds.) Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop, pp. 15–24. No. CS-TR-1099 in Technical report Series, University of Newcastle upon Tyne (2008)
17. Chen, Z., Morisset, C., Stolz, V.: Specification and validation of behavioural protocols in the rCOS modeler. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 387–401. Springer, Heidelberg (2010)
18. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
19. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, New York (1990)
20. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972). An ACM Turing Award lecture

21. Dong, R., Zhan, N., Zhao, L.: An interface model of software components. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 159–176. Springer, Heidelberg (2013)
22. Fischer, C.: Fault-tolerant programming by transformations. Ph.D. thesis, University of Warwick (1991)
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1994)
24. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
25. He, J., Li, X., Liu, Z.: Component-based software engineering. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 70–95. Springer, Heidelberg (2005)
26. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. **160**, 173–195 (2006)
27. He, J., Liu, Z., Li, X.: rCOS: a refinement calculus of object systems. Theor. Comput. Sci. **365**(1–2), 109–142 (2006)
28. Heineman, G.T., Councill, W.T.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Professional, Reading (2001)
29. Herold, S., et al.: CoCoME - the common component modeling example. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 16–53. Springer, Heidelberg (2008)
30. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
31. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
32. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle River (1998)
33. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
34. Holzmann, G.J.: Conquering complexity. IEEE Comput. **40**(12), 111–113 (2007)
35. Johnson, J.: My Life is Failure: 100 Things You Should Know to Be a Better Project Leader. Standish Group International, West Yarmouth (2006)
36. Jones, C.B.: Systematic Software Development using VDM. Prentice Hall, Upper Saddle River (1990)
37. Ke, W., Li, X., Liu, Z., Stolz, V.: rCOS: a formal model-driven engineering method for component-based software. Front. Comput. Sci. China **6**(1), 17–39 (2012)
38. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of OO programs. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 347–366. Springer, Heidelberg (2009)
39. Khaitan, S.: Design techniques and applications of cyber physical systems: a survey. IEEE Syst. J. **9**(2), 350–365 (2014)
40. Kroll, P., Kruchten, P.: The Rational Unified Process Made Easy: a Practitioner's Guide to the RUP. Addison-Wesley, Boston (2003)
41. Lamport, L.: Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
42. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice-Hall, Upper Saddle River (2001)
43. Lee, E.: Cyber physical systems: design challenges. Technical report No. UCB/EECS-2008-8, University of California, Berkeley (2008)

44. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. IEEE Comput. **26**(7), 18–41 (1993)
45. Li, D., Li, X., Liu, Z., Stolz, V.: Interactive transformations from object-oriented models to component-based models. Technical report 451, IIST, United Nations University, Macao (2011)
46. Li, D., Li, X., Liu, Z., Stolz, V.: Interactive transformations from object-oriented models to component-based models. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 97–114. Springer, Heidelberg (2012)
47. Li, D., Li, X., Liu, Z., Stolz, V.: Support formal component-based development with UML profile. In: 22nd Australian Conference on Software Engineering (ASWEC 2013), Melbourne, pp. 191–200. IEEE Computer Society, 4–7 June 2013
48. Li, X., Lu, R., Liang, X., Shen, X., Chen, J., Lin, X.: Smart community: an internet of things application. Commun. Mag. **49**(11), 68–75 (2011)
49. Li, X., Liu, Z., He, J.: Formal and use-case driven requirement analysis in UML. In: 25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, Chicago, pp. 215–224. IEEE Computer Society, 8–12 October 2001
50. Li, X., Liu, Z., He, J.: A formal semantics of UML sequence diagram. In: 15th Australian Software Engineering Conference (ASWEC 2004), Melbourne, pp. 168–177. IEEE Computer Society, 13–16 April 2004
51. Li, X., Liu, Z., He, J.: Consistency checking of UML requirements. In: 10th International Conference on Engineering of Complex Computer Systems, pp. 411–420. IEEE Computer Society (2005)
52. Liu, J., Liu, Z., He, J., Li, X.: Linking UML models of design and requirement. In: Proceedings of the 2004 Australian Software Engineering Conference, pp. 329–338. IEEE Computer Society (2004)
53. Liu, Z.: Software development with UML. Technical Report 259, IIST, United Nations University, P.O. Box 3058, Macao (2002)
54. Liu, Z., Chen, X.: Interface-driven design in evolving component-based architectures, workshop of the 25 Anniversary of ProCoS, Proceedings to be Published in Springer Lecture Notes in Computer Science (2015)
55. Liu, Z., He, J., Li, X., Chen, Y.F.: A relational model for formal object-oriented requirement analysis in UML. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 641–664. Springer, Heidelberg (2003)
56. Liu, Z., Joseph, M.: Transformation of programs for fault-tolerance. Form. Asp. Comput. **4**(5), 442–469 (1992)
57. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Trans. Program. Lang. Syst. **21**(1), 46–89 (1999)
58. Liu, Z., Kang, E.Y., Zhan, N.: Composition and refinement of components. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 238–257. Springer, Heidelberg (2010)
59. Liu, Z., Li, X., He, J.: Using transition systems to unify UML models. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 535–547. Springer, Heidelberg (2002)
60. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), pp. 371–382. IEEE Computer Society, full version as UNU-IIST Technical report 343 (2006)
61. Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tool for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 62–80. Springer, Heidelberg (2010)

62. Long, Q., Liu, Z., Li, X., He, J.: Consistent code generation from UML models. In: Australian Software Engineering Conference, pp. 23–30. IEEE Computer Society, UNU-IIST TR 319 (2005)
63. Long, Q., Qiu, Z., Liu, Z.: Formal use of design patterns and refactoring. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. Communications in Computer and Information Science, vol. 17, pp. 323–338. Springer, Berlin (2008)
64. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Q. **2**(3), 219–246 (1989)
65. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1992)
66. Milner, R.: Communication and Concurrency. Prentice-Hall Inc., Upper Saddle River (1989)
67. Naur, P., Randell, B. (eds.): Software engineering: report of a conference sponsored by the NATO science committee, Garmisch, 7–11 October 1968, Brussels, Scientific Affairs Division, NATO (1969)
68. Nielson, H., Nielson, F.: Semantics with Applications. A Formal Introduction. Wiley, Chichester (1993)
69. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering, ICSE 2000, pp. 35–46. ACM, New York (2000)
70. Object Managment Group: Model driven architecture - a technical perspective, document number ORMSC, 01 July 2001
71. Oettinger, A.: The hardware-software complementarity. Commun. ACM **10**(10), 604–606 (1967)
72. Palomar, E., Liu, Z., Bowen, J.P., Zhang, Y., Maharjan, S.: Component-based modelling for sustainable and scalable smart meter networks. In: Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM 2014, Sydney, pp. 1–6, 19 June 2014
73. Peter, L.: The Peter Pyramid. William Morrow, New York (1986)
74. Peterson, J.L.: Petri Nets. ACM Comput. Surv. **9**(3), 223–252 (1977)
75. Plotkin, G.D.: The origins of structural operational semantics. J. Log. Algebr. Program. **60**(61), 3–15 (2004)
76. Pressman, R.S.: Software Engineering: A Practitioner's Approach. The McGraw-Hill Companies, New York (2005)
77. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
78. Randell, B., Buxton, J. (eds.): Software engineering: report of a conference sponsored by the NATO science committee, Rome, Italy, 27–31 October 1969, Brussels, Scientific Affairs Division, NATO (1969)
79. Rausch, A., Reussner, R., Mirandola, R., Plášil, F.: Introduction. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 1–3. Springer, Heidelberg (2008)
80. Rayl, A.: NASA engineers and scientists-transforming dreams into reality (2008)
81. Robinson, K.: Ariane 5: flight 501 failure - a case study (2011)
82. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
83. Shapiro, M.: Smart cities: quality of life, productivity, and the growth effects of human capital. Rev. Econ. Stat. **88**, 324–335 (2006)
84. Sommerville, I.: Software Engineering, 6th edn. Pearson Education Ltd., England (2001)

85. Spivey, J.M.: The Z Notation: a Reference Manual, 2nd edn. Prentice Hall, Upper Saddle River (1992)
86. Stellman, A., Greene, J.: Applied Software Project Management. O'Reilly Media (2005)
87. Stoy, J.E.: Denotational Semantics: the Scott-Strachey Approach to Programming Language Semantics. MIT Press, Cambridge (1977)
88. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Longman Publishing Co. Inc., Boston (2002)
89. Wirsing, M., Banâtre, J.P., Hölzl, M.M., Rauschmayer, A. (eds.): Software-Intensive Systems and New Computing Paradigms - Challenges and Visions, vol. 5380. Springer, New York (2008)
90. Yang, L., Stolz, V.: Integrating refinement into software development tools. In: Pu, G., Stolz, V. (eds.) 1st International Workshop on Harnessing Theories for Tool Support in Software. Electronic Notes in Theoretical Computer Science, vol. 207, pp. 69–88. Elsevier, Amsterdam, UNU-IIST TR 385 (2008)
91. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Form. Asp. Comput. **21**(1–2), 103–131 (2009)
92. Zhao, L., Wang, S., Liu, Z.: Graph-based object-oriented hoare logic. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 374–393. Springer, Heidelberg (2013)