Internal Report 94-23 — September 1994

Integrating Method Objects into the Object-Oriented Life Cycle Approach

Th. van Elzakker

Master's Thesis

Leiden University, the Netherlands
Faculty of Mathematics and Natural Sciences
Department of Computer Science
Software Engineering group

Braunschweig Technical University, Germany
Institute for Programming Languages
and Information Systems
Database department

Dortmund University, Germany
Department of Computer Science
Software Technology Research group, ls.10

Abstract

As each stage of a software life cycle is executed, one or more methods are utilised. Such methods, for instance specification languages or graphic modelling tools, have usually been developed to cover only a part of the development process. Multiple methods are therefore needed to cover the whole software life cycle. But, normally, independently developed methods cannot be integrated perfectly. Inconsistencies and redundancies between individual methods exist that hinder the seamless transition between the stages of the life cycle. By re-engineering the methods used or by engineering new methods, these problems may be solved.

Such method integration requires understanding of the structure of a method. [LR93] has shown that each development method may be seen as a structured collection of basic development steps. To further enhance clarity, an object-oriented life cycle model may now be built where we introduce such a structure as a method object and model its behaviour and its interaction with other objects.

Preface

When I chose to write my master's thesis in Germany, I hardly knew what to expect. In retrospect I think I made a good choice. And although the conclusion of this project is not yet the conclusion of my studies in computer science and, as things are now, not in the least the end of my academic career, my time in Germany has been a personal milestone. I enjoyed the experience abroad and I have learned more than 'just' about method objects.

I would like to express my gratitude to Prof. H.-D. Ehrich, PhD. from the Technical University in Braunschweig and Prof. W. Schäfer, PhD. from the University of Dortmund for giving me the opportunity to work and study at their departments. Also to Prof. G. Engels, PhD. from Leiden University for supporting me in going to Germany.

In Germany I was advised by Perdita Löhr, PhD. I really enjoyed working with her. By always being cheerful and enthusiastic she created an atmosphere in which I wasn't afraid to make mistakes or ask (stupid) questions. I want to thank her for that.

Finally, I want to thank Göran, Henrik, Jan-Willem, and Henk for their support and shown interest, and last but not least, a special thanks to Hans de Jong for breaking my arm and giving himself a hard time for it afterwards.

Thomas van Elzakker
Leiden, September 13, 1994

Contents

List of Figures

1 Introduction

Before an idea becomes an operational software product, it goes through several development stages. Discerning, understanding, and controlling these individual stages is vital for a well built product. In the last decade this control of development has become even more important because of the growing size of development projects. It has become practically impossible to develop a software system without some structured approach. Apart from the necessity to be able to control the complete development process, the cost aspect has become a pressing issue. As a result, software is often developed with the idea of reuse in the back of the developers mind. Reuse of software saves development time and money.

A modular approach to software development is nothing new. Breaking the development up into individually developed parts has indeed improved the managing of projects and the reusability of software. An object-oriented approach to software development takes the modularity aspect even further. The developers and the managers of the project are forced to approach every aspect of the development with an object-oriented view. Instead of just splitting up large problems into smaller ones it means identifying all the items, or rather objects, that play a role in a certain problem. By studying the behaviour of these objects and the way they interact, an object-oriented view is built. This approach has shown to improve the reusability of development products. The encapsulated objects and their well defined interface to the outside world are particularly useful for that.

When taking an object oriented view to the development process itself, we have to distinguish the objects and relationships in this process. The understanding of the development process that is thus built will enable us to manage software development more efficiently and improve the quality of software products on the whole.

In this thesis the object-oriented view to the software development process, defined at the Department of Computer Science, Technical University of Braunschweig (Germany) [LRR93] is further expanded. It is combined with the concept of generic methods, as developed by [LR93]. The result is an object-oriented model of a software development process.

- In **chapter 2**, I will introduce the basics of the software development process. Furthermore I will lift the veil off some of the problems concerned with method integration.
- To build an understandable model, I will use an object-oriented modelling technique. The structure and notation of this technique is described in **chapter 3**.
- **Chapter 4** contains a discussion of the theory of generic methods. In the second part the theory of the existing object-oriented approach to the software development process is discussed.
- In **chapter 5**, I present an object-oriented model of a software development process. This model clearly identifies the objects and their behaviour in the process. Also the relationships between the distinguished objects are described in detail.
- I conclude this thesis with **chapter 6**. A chapter on how the use of the model in chapter 5 may affect software developing.

2 The Software Development Process

In this chapter the basics of the software development process are introduced. An understanding of the structure of a software life cycle is built and the role of methods in the software development process is described. The chapter ends with a brief discussion on method integration.

2.1 The Software Life Cycle

During the development of a software system we can discern several stages in which the system is gradually developed from idea to final product. These stages, in a certain execution order, are called the *life cycle of the software system*. The traditional life cycle model for a software system is the *waterfall model*. It has well-defined starting and ending points and each stage "flows" into the next. Normally, when the system has lived through one stage, it does not return to it.

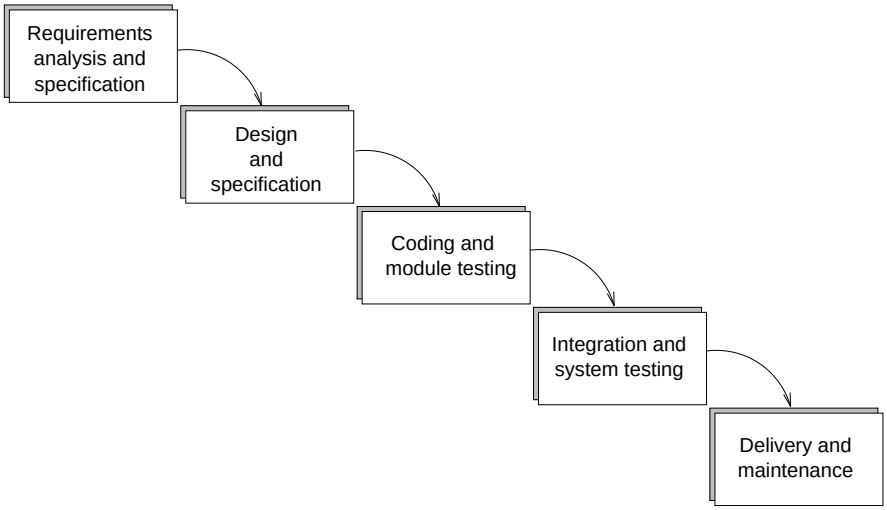

Figure 2.1: The waterfall model

In reality, the development of a software system can not be modeled this simple. The rigid structure of the waterfall model requires major development decisions to be made in the early stages. The particular problems of the later stages might not be completely understood though. Furthermore, it is very idealistic to assume that, for instance, all analysis is done before starting with the design, or that coding is only done when everything has been designed. Less rigid (and therefore more realistic) life cycle models contain most of the stages which are identified in the waterfall model. The main differences are found in the order in which these stages are entered, and the defined transition criteria.

In the early days of computing, there was no structure in the development of software. We can think of the development approach of these days as the *code-and-fix* approach. Basically, this appellation denotes a development process that is neither precisely formulated nor carefully controlled; rather, software production consists of the iteration of two steps:

(1) write code and (2) fix it to eliminate errors, enhance existing functionality, or add new features [GJM91]. Capturing the development of a software system in a life cycle model gives some important advantages over the code-and-fix approach. Firstly, developers are indeed stimulated to think ahead and to anticipate problems that could develop in the lower development stages. Secondly, breaking up the development in distinct stages, each one well described, makes the development process better to plan and to manage.

This is the basic idea behind building a software life cycle model: the software development process should be subject to discipline, planning and management.

Because of its rigidity, the waterfall model is not a very good model of a software development process. In a survey on software life cycles [LRR93] identified a collection of aspects of development processes which are insufficiently reflected in life cycle models. These are:

- Completeness of stages requires the problem to be understood in its entirety. In real development life, this is the exceptional case. Thus, the less completeness is strictly requested for the result of the stage, the better.
- Flexible iterative proceeding is indispensable for life cycle models because problem and specification evolve over (development) time.
- Separations between and within stages increase the probability of errors. Thus, life cycle models should emphasise integration aspects and the seamless transition between development stages.
- Always handling the complete specification during the entire development process contradicts human working principles which base on small work units. Thus, life cycle models should support modular and incremental working styles.
- Besides modelling the problem domain and the system design, organisational tasks have to be modelled, too. A life cycle model has to determine how the modelling process is to be organised. These aspects are left out by most life cycle models.
- Measures, by which the quality of a software product or the progress of the development process can be evaluated, are missing. Life cycle models should incorporate some kind of criteria for measuring specifications.

2.2 Methods and Method Integration

Developers who use a life cycle model usually adopt specific *methods* to obtain the required results of a stage in the model. The set of methods used in a certain software production process is called its *methodology* [LRS94]. For each distinct stage of a model, numerous methods exists. Some methods cover two or more stages, other methods cover just part of a stage. A method may be a number of things, depending on the stage of the development process. A method does more than define and order the steps that must be taken in the process. A method also defines the language in which the results are presented and it models the objects that are represented, manipulated and analysed in such a way that irrelevant details are filtered out. Furthermore, a method should have some degree of guidance for applying the method in the form of literature or manuals. Many software

developers have their own methods for the stages of the software development process. Examples of well known-methods are OMT [RBP+91], OOA/OOD [CY91a, CY91b], TROLL [JSHS91], SADT [Ros77, MM88] and LOTOS [ISO88, DE93].

In a software development process often several independently developed methods are used. This approach is bound to give rise to problems such as inconsistency, redundancy and possible loss of information. One alternative is to use one standard method for the whole development process. It is unrealistic to assume that such a method covers all stages and aspects of the development process perfectly. Therefore choosing this option leads to a lower quality of the product or a longer development time. By using the best method for each part of the process the developers are forced to handle the incompatibilities between the chosen methods. They will have to sew the methods together. This kind of "method building" typically involves [Kro93]:

- Choosing a set of existing methods which support different parts and aspects of the overall process
- Modifying the methods to fit the intended purpose
- Developing novel methods to fill the holes and to bridge the gaps between the chosen methods
- Combining the modified and developed methods to form essentially a new and more comprehensive method

Combining two or more existing methods to produce a single method for a given purpose is called *method integration*. By integrating methods the developer can combine the strengths of the methods for each part of the development process and, by doing so, reduce the weaknesses of the individual methods. The goal of integrating methods is to produce one single method which is then "the method" for a given process.

▷ **Method Integration** is the activity of resolving existing incompatibilities between methods so that they can be safely and effectively used together [Kro93].

To classify method integration, we can look at the relation of a method to the stage that it is used in. In this context we want to consider an iterative and concurrent life cycle model (figure 2.2). In such a model, the information stream is not just from early stages to later stages in the development process. There is also feedback from later stages to earlier stages. The stages themselves are considered to be parallel processes with information flowing between the processes. In this regard we can distinguish at least two cases of method integration [Kro93]:

- *Intra-process* method integration. This refers to the integration of methods within a specific process. Here the methods are used at the same time, concurrently.
- *Inter-process* method integration. This refers to the integration of methods across two or more processes. Here the methods are used one after another, sequentially.

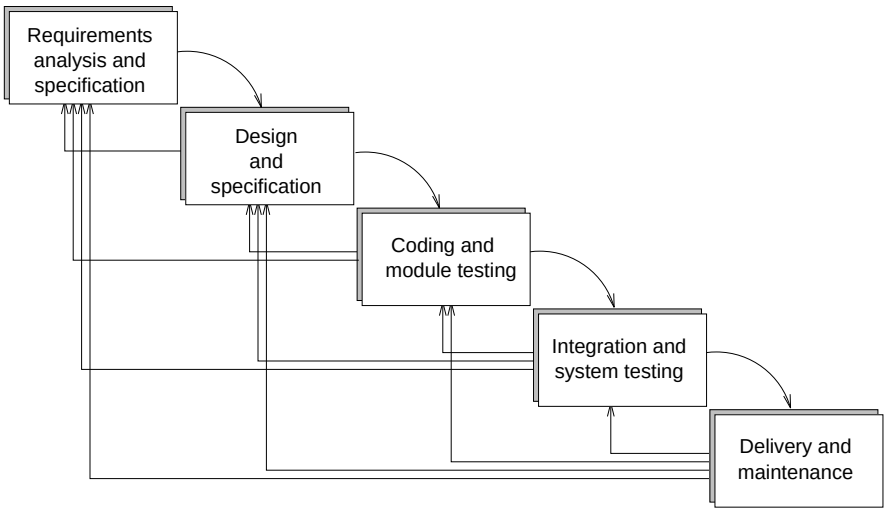

Figure 2.2: Example of an iterative life cycle model

Intra-process method integration provides a more complete support for a given process. During design, for example, the developer may want to design the communication between two systems with LOTOS and the interface for the communication with OMT. The goal of integrating methods this way is to guarantee the consistency of the overall system.

Inter-process method integration provides a seamless transition between two processes. For instance, it makes the development process on the whole more efficient by filling the gaps between methods and eliminating redundant parts of the methods.

3 Object-Oriented Modelling

This chapter introduces the object-oriented way of thinking. In the first part of this chapter the theory and terminology associated with object-orientation is explained. Following that, I will describe the object-oriented modelling and design method that I will use throughout this thesis.

3.1 Object-Orientation

Object-orientation is a modelling philosophy. Central to object-orientation is the *object*. An object is a model of a real world concept that captures its data structure as well as its behaviour in a single entity. In software development, object-oriented modelling is used to break up a problem in several parts, each described as an object. The characteristics required by an object-oriented approach generally include four aspects: *identity*, *classification*, *polymorphism*, and *inheritance* [RBP+91].

- **Identity** means that every single object has a unique identity, even if all its attributes are identical. The bicycles in figure 3.1 are exactly the same size, colour and they have the same behaviour. Still, they are two different bicycles.

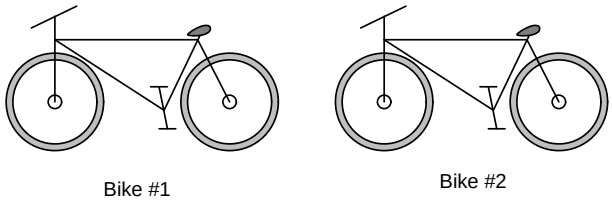

Figure 3.1: Identity.

- **Classification** means that objects are grouped in *classes*. Object classes describe objects with the same data structure (attributes) and the same behaviour (opera-

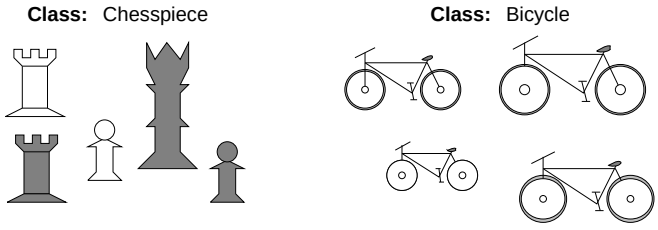

Figure 3.2: Classification.

tions). A single object is called an instance of its class. A class therefore describes a possible infinite set of individual objects. In figure 3.2 we see several specific objects grouped in two classes.

- **Polymorphism** means that the same operation may behave differently on different classes. An *operation*[1] is an action or transformation that an object performs or is subject to. The operations of an object capture its behaviour. By implementing a specific operation differently in different classes, polymorphism is realised. Figure 3.3 illustrates two classes with each a different implementation of the operation *move*.

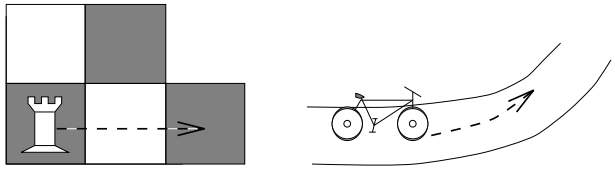

Figure 3.3: Polymorphism.

- **Inheritance**. Through inheritance, classes share attributes and operations, based on a hierarchical relationship. An attribute or (polymorphic) operation defined in a class can also be found in its subclasses. In addition, a subclass can have extra attributes and operations. This is called *specialisation*. In figure 3.4 the class Geometric Figure has two subclasses. These subclasses have all the attributes and operations from their *superclass* and some specialised ones. Since an instance of a class is at the same time an instance of all superclasses of this class, all class features must apply to the subclass instances. A subclass may alter an implementation of an operation but may not change the operation's signature. When new features are added to a specific operation this is called *extension*. Constraining attributes of a subclass is called *restriction*.

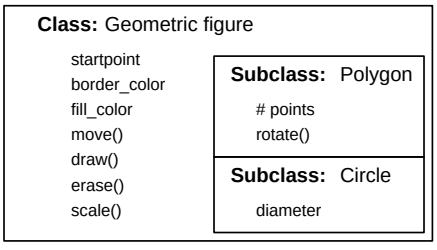

Figure 3.4: Inheritance.

To accurately describe and model objects and associations in this paper I will use the "Object Modeling Technique" (OMT). A method described by Rumbaugh et al [RBP+91] which I will present in the following sections.

[1] The specific implementation of an operation by a certain class is called a *method*. To avoid confusion with other meanings of the word "method" in this thesis, I will not use it in this context.

3.2 The "Object Modeling Technique" (OMT)

OMT uses three types of models to describe a system: the *object model*, describing the objects in the system and their relationships, the *dynamic model*, describing the interactions among objects in the system, and the *functional model*, describing the data transformations of the system. In the next three sections, these three models are each discussed in more detail.

3.2.1 The Object Model

The structure of the objects in a system are described by the object model. In an object model a framework is built in which the other two models, the functional and dynamic model, may be placed. The object model captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterise each class of objects.

A formal graphic notation for object models is called an *object diagram*. There are two types of object diagrams in which we model the structure of classes and objects: *class diagrams* and *instance diagrams*.

- A **Class Diagram** is a schema, pattern or template for describing many possible instances of data. A class diagram describes *object classes*.
- An **Instance Diagram** describes how a particular set of objects relate to each other. An instance diagram describes *object instances*.

Classes and Instances

The most essential part of the object model is the object class. Classes are the result of grouping object instances with similar properties (attributes), common behaviour, common relationships to other classes, and common semantics. The graphic notation of a

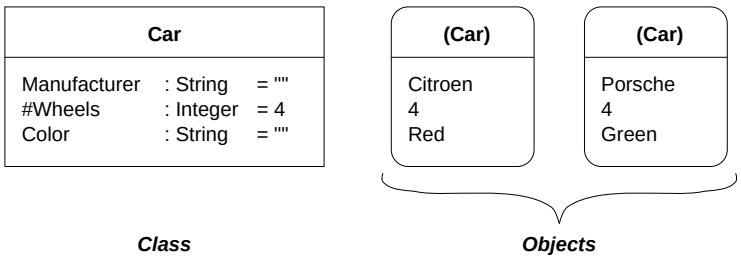

Figure 3.5: The graphic notation of a class and objects in OMT

class is a box. The name of the class is put centrally in bold characters at the top of the box. The attributes are written underneath, separated by a horizontal line. Another line separates the attributes from the list of operations. A single object instance is drawn as a box with rounded corners. The class of the object, its type, is put in parentheses at the top of the rounded box. Instead of the name and the type of the attributes, their value is shown (cf. figure 3.5).

Associations

Ofcourse, modelling an object class itself is not enough. We must also model the *associations* of the classes with other classes. Modelling associations between classes is done by drawing a line between the classes that are associated (cf. figure 3.7). Next to this line the name of the association is written. An association describes a group of *links* with common structure and common semantics. A link is a physical or conceptual connection between objects. An association describes the relationship between classes, a link describes the relationship between instances. I will now describe the concepts of associations and links in more detail and show how they are modelled, using OMT.

- Associations differ in **multiplicity**. Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity is only modelled in a class diagram. Figure 3.6 illustrates the modelling of multiplicity of associations. The multiplicity indicators define the minimum and maximum number of objects that participate in one end of an association. For instance, a "Many" multiplicity indicates that many instances of the class where the indicator ball is drawn may join in the association. It doesn't say anything about how many instances of the class on the other end of the association participate in the association.

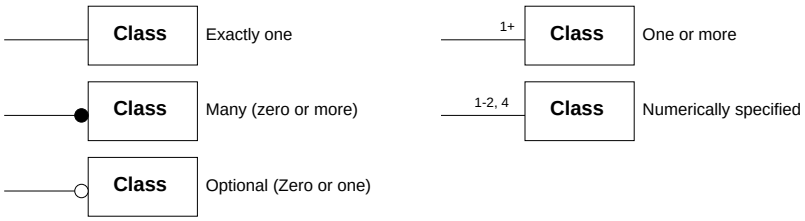

Figure 3.6: Notation of multiplicity of associations in OMT

- Links and associations may model more than "just" a connection between classes or instances. A link may have an attribute value. Such a **link attribute** couples a value to every instance of a certain association. For each link the value may be

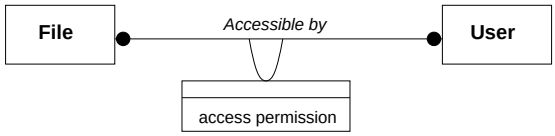

Figure 3.7: A link attribute for a many-to-many association

different, but the type of the attribute is the same for all links of the association. The way a link attribute is modelled is shown in figure 3.7. A box is connected to the association with a loop. The attributes are written in the second part of the box.

The box that is attached to the association in figure 3.7 is in fact a class. A link attribute is a special case of modelling an association as a class. Every time a link

is instantiated, an object of this class is also instantiated. In fact, they are exactly the same thing. When an association is modelled as a class, it becomes possible to associate it with other classes.

- Classes that take part in an association have a certain **role** in that association. Sometimes the role may not be exactly clear and we can then explicitly name it in the model. In this case the *role name* is written next to the association line near the class that plays that role and uniquely identifies one end of an association. It is necessary to use role names when modelling associations between two objects of the same class. Role names are also useful when modelling more than one association between two classes. A role name can be viewed upon as a derived attribute of the class. To avoid confusion with the other attributes and to keep the roles apart, all role names on the far end of associations attached to a class must be unique and they must not have the same name as an attribute of the class itself. Figure 3.8 is an illustration in which both uses of a role name are shown. A person may have a boss, who is a person himself (an association between two objects of the same class). Persons may work for one or more companies. Persons may have been working for other companies (more than one association between two classes).

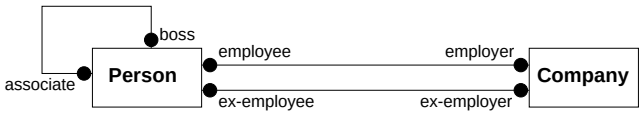

Figure 3.8: Role names for an association

- Another characteristic of associations is **ordering**. In most cases, when a set of objects takes part in a many-association, the order within the set makes no difference. In some cases however, it does. We can think of a set of customers at a post-office. These customers have lined up in a certain order. Modelling such an ordered set of objects on the many-side of an association is done by writing "{ordered}" next to the multiplicity dot of the association (cf. figure 3.9).

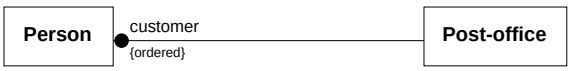

Figure 3.9: Ordered sets in an association

- To improve understanding of a many-to-one association, we may want to remodel it as a **qualified** association. A qualifier is an attribute from a class that takes part in a many-to-many or many-to-one association on the "Many"-side. If the value of the particular attribute is unique in every instance of that class, then it uniquely identifies the link between the instances. By modelling this attribute as a qualifier on the association we effectively reduce the multiplicity of the association. The qualifier is drawn in a small rectangle on the association line, touching the class it qualifies. Figure 3.10 shows the use and notation of a qualifier.

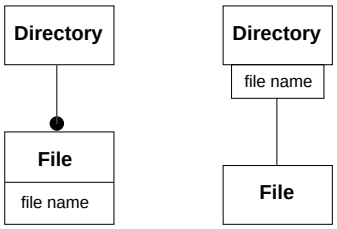

Figure 3.10: Unqualified and qualified association

Qualification improves semantic accuracy and increases the visibility of navigation paths. It is much more informative to be told that a directory and a file name combine to identify a file, rather than to be told that a directory has many files. The qualification syntax indicates that each file name is unique within its directory.

If the attribute, taken from the class on the many-side, does not have a unique value for every object instance, qualification will not reduce the multiplicity. What will happen is that the set of related objects is partitioned in disjoint subsets. Because the attribute value is not unique for every object instance, the subsets may contain more than one object. When we look at figure 3.10, we can see that if we had taken an attribute such as filesize as the qualifier, we would still have had a many-to-one association because more than one file with the same file length may belong to a specific directory.

- **Aggregation**. Aggregation is a tight association between classes that is the *part-of* relationship. Objects that are the components of another class of objects are associated through aggregation. Each component of an assembly has its own ag-

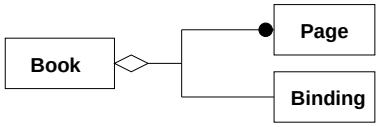

Figure 3.11: Aggregation

gregation relationship with multiplicity and possibly link attributes. An aggregation is drawn with a small diamond on the assembly end of the relationship (cf. figure 3.11). Aggregation induces a property that is called *propagation*, or *triggering of*

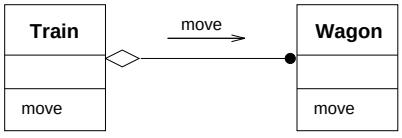

Figure 3.12: Propagation of operations

operations. Some operations, when applied to the assembly, are automatically prop-

agated to (some of) the components. For example move is often such an operation. Propagation of operations is modelled by drawing an arrow in the direction of the propagation, parallel to the association line. Next to the arrow, the name of the operation is written (cf. figure 3.12).

- Apart from binary associations, we may want to model ternary or higher order associations. As a rule, associations of an order higher than three should be avoided. They are complicated to draw, to implement, and to think about. In figure 3.13 is shown how a ternary association is modelled. The multiplicity notation used for binary associations is not unambiguous when used for n-ary ($n > 2$) associations. In the latter case, we must specify a *candidate key*. This means specifying which combinations of instances of the classes participating in the association, uniquely identify a link. In figure 3.13 is shown how to model a candidate key in a ternary association. All possible candidate keys for a single relation must be written underneath the association.

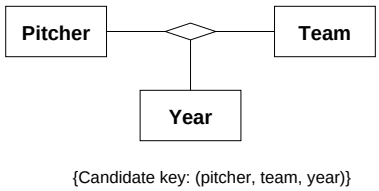

Figure 3.13: A ternary association with its candidate key

Generalisation

At the beginning of this chapter I mentioned the four aspects of object-orientation. One of these aspects is inheritance. The following part of this section will be about modelling inheritance and **generalisation**. Generalisation is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is a *subclass*. Attributes and operations are inherited by the subclasses from the superclass. The notation of generalisation is a triangle connecting a superclass to its subclasses. The superclass is connected by a line to the apex of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of the triangle. Figure 3.14 shows how it is done. In this figure we see generalization across two levels. A bike and a car are descended from the class Landbased. The class Rowing boat and the class Motorboat are descended from the class Waterbased. Both classes Landbased and Waterbased are descendants of the class Vehicle. *Descendent* and *ancestor* are terms that are used across all levels of generalisation. For instance, in figure 3.14, the ancestors of the class Car are the classes Landbased and Vehicle.

The triple dot in the figure indicates that there are additional subclasses that are not shown on the diagram. When the generalisation is based on the value of an attribute, this attribute may be stated next to the triangle. Such an attribute is called the *discriminator*.

The generalization shown here is *disjoint*. Non disjoint generalisation is modeled by using a filled triangle instead of an unfilled one.

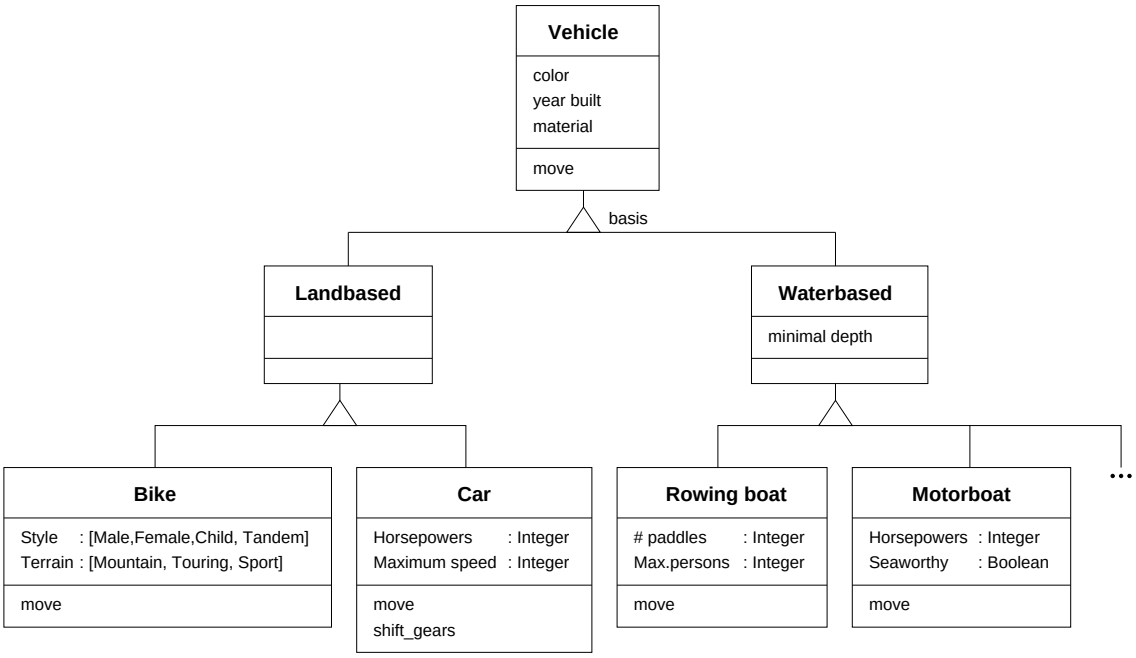

Figure 3.14: Example of generalisation and inheritance of features

Until now, only single inheritance is mentioned but *multiple inheritance* is also possible. Multiple inheritance occurs when a subclass inherits features from more than one superclass. Such a class is called a *join class*. Ambiguities that are created by multiple inheritance must be resolved in the implementation of the join class. Actually, since inheritance is a language feature and generalisation is the conceptual relationship, *multiple generalisation* would be a more correct term. But since the term multiple inheritance is so widely spread as it is, not using it would be confusing. Modelling multiple inheritance is shown in figure 3.15, where the class Amphibious Vehicle is the join class.

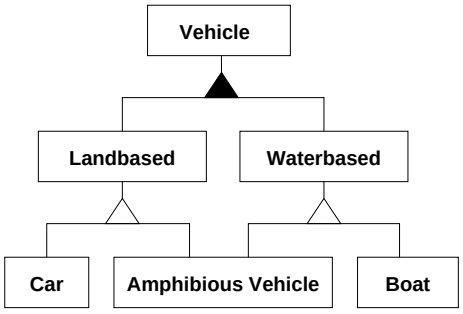

Figure 3.15: Multiple inheritance from non disjoint classes

Metadata

Metadata is data that describes other data. For example, the description of a building in a blueprint is metadata. The description of a class is also metadata. Metadata may be used in object-oriented modelling when we want to model a class that describes another class. Consider a class CarModel. A manufacturer of cars may have a number of different car models. Every specific car model may be modelled as an instantiation of the class CarModel. Such an instantiation may be considered to be a class itself. The class CarModel then describes another class and is therefore metadata. Such a class describing another class is called a *pattern* or a *metaclass* and an instance of a metaclass (i.e. the class being described) is called an *individual*. In the case of modelling metadata we may want to put a pattern and an individual in the same model. This is done by modelling *instantiation*. The notation of instantiation is a dotted arrow drawn from the instance to the class. In figure 3.16 the example of the CarModel and Car classes is drawn. Showing the instantiation of a metaclass this way can also be done with regular classes instantiating specific object instances when necessary.

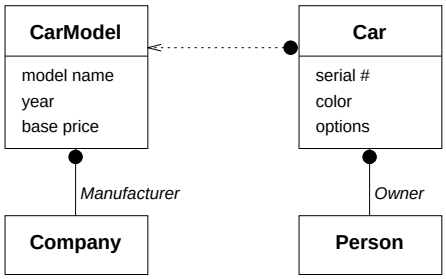

Figure 3.16: Patterns and individuals

When classes are considered as instances from metaclasses, such instances have *class attributes* and *class operations*. A class attribute is a value common to the entire class. It is a value for an attribute from the metaclass. A class operation is an operation on the class itself. For example, operations to create an instance are class operations. A class attribute or a class operation is modelled by leading the name with a dollar sign. Figure 3.17 shows a class Window with its class features.

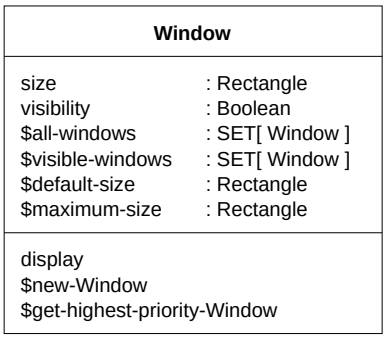

Figure 3.17: Class with class features

Constraints

Constraints are functional relationships between classes, instances, associations and links. By stating a constraint we can restrict the values these entities can have. Normally, constraints should be modelled in the functional model. However, simple constraints may be modelled in the object model. In fact, multiplicity and ordering are constraints on association links. Some constraints (like multiplicity) have a special notation, but constraints are otherwise delimited by braces and written near the constrained entity. A dotted line connects multiple constrained entities. An arrow may be used to connect a constrained entity to the entity it depends on. In figure 3.18 we see examples of a constraint on an object and a constraint between associations.

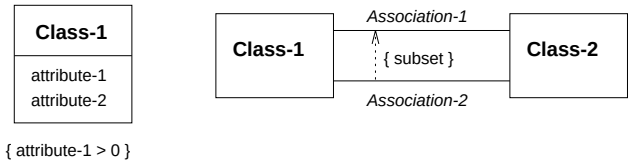

Figure 3.18: Constraints

We may model some entities as *derived entities*. Derived entities are always redundant but may ease comprehension. In the model they are indicated by a slash (/) (cf. figure 3.19). A derived entity always has a derivation rule, explaining how the entity was derived. Such a derivation rule is a constraint on the derived entity and is modelled as such.

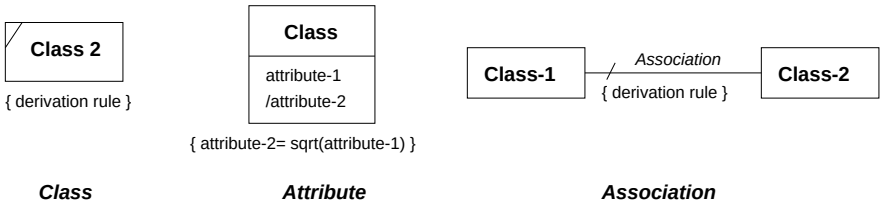

Figure 3.19: Derived entities

This concludes the section about the object model. Figure 3.20 is an example of a complete model in which most of the structures out of this section are used. It also illustrates a term that has not been mentioned before, namely a *sheet*. A sheet is a grouping construct to split up large models that do not fit onto one page. A sheet is always exactly one page in size. As a rule not more than one view on a situation is put on one sheet. Classes may appear on multiple sheets, but associations only appear on one single sheet. In the example in figure 3.20 the sheet displays the module of a certain model that is about a book. It is labelled as such and numbered. The labelling and/or the numbering may be used to cross-reference between sheets. In the example, the circled numbers "2" and "3" are references to other sheets.

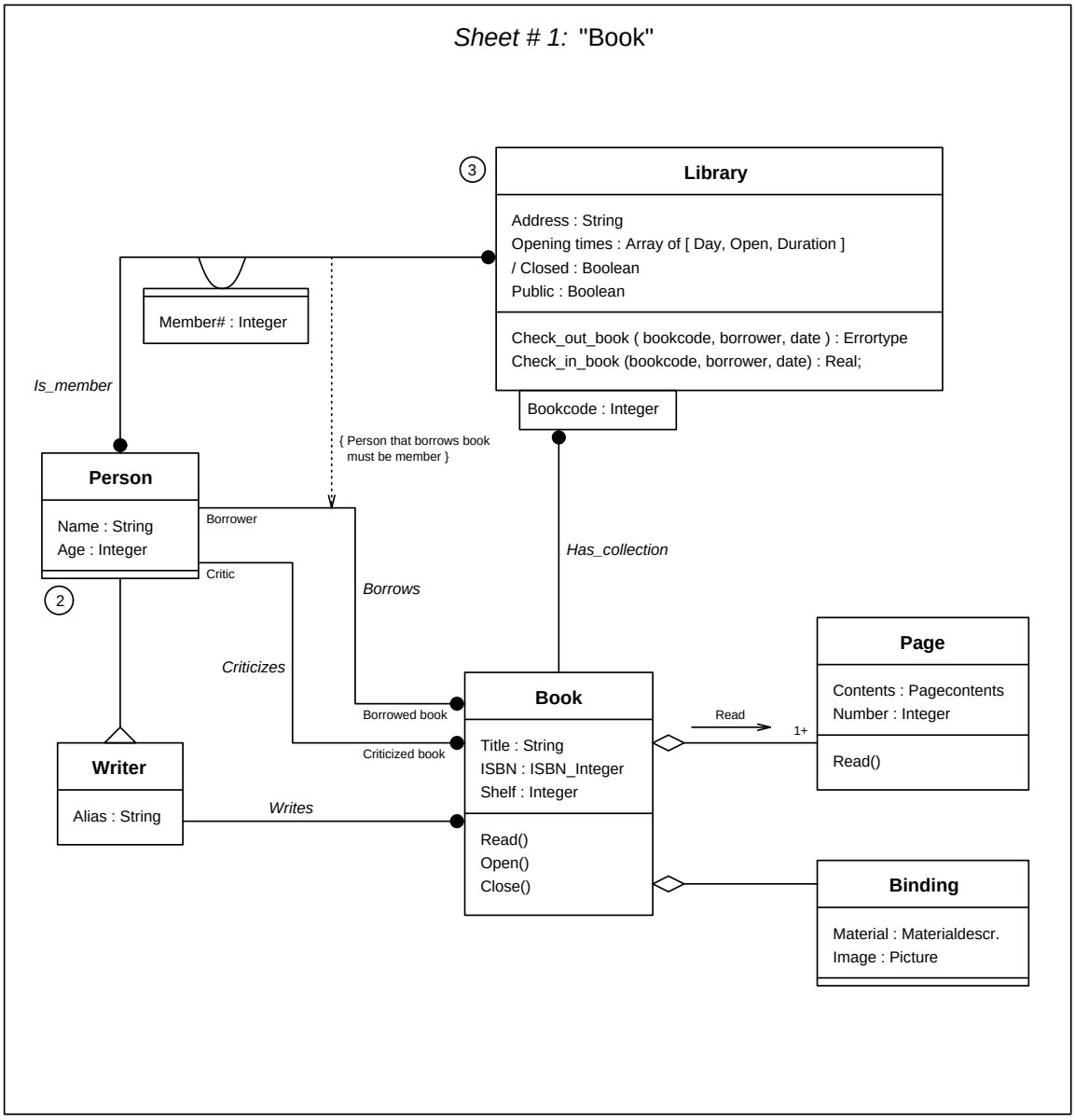

Figure 3.20: Example of a sheet out of an object model

3.2.2 The Dynamic Model

In an object model, the static structure of objects and the relationship between them are modelled. The temporal relationships, the changes to the objects and their relationships over time, are modelled in the *dynamic model*. In the following section I will discuss the aspects of dynamic modelling as presented by [RBP+91]. Dynamic models are modelled using *state diagrams*. The notation used for structured state diagrams is mainly based on the notation of David Harel [Har87].

States and Events

The dynamic model specifies allowable sequences of changes to objects. A specific state diagram describes all or part of the behaviour of one object of a given class. Central aspects to dynamic modelling are *states* and *events*.

- A **state** is an abstraction of the attribute values and links of an object.
- An **event** is something that happens at a point in time.

Over time the objects stimulate each other, resulting in a change of states. An event is a single stimulus from one object to another. In a *state diagram*, states and events are related. States are connected by a *transition* which is triggered by a specific event. A state is drawn as a rounded box containing an (optional) name. A transition is drawn as an arrow from the receiving state to the target state. The transitions are labelled with the name of the event causing it. An occurence of a transition as the result of an event is called the *firing* of the transition. The object then enters the state at the arrow-end of the transition. In figure 3.21 we see a model of a Match changing states because it is lighted (event). When a transition is not labelled, it fires automatically when the state comes to an end. For instance when the Match is in the state Burning, it may automatically enter the state Burned out when the flame goes out. This type of transitions are called *automatic transitions*.

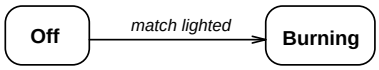

Figure 3.21: A transition from one state into another

A *condition* is a Boolean function on object values. A state can be defined in terms of conditions. When an event fires a transition, only when a certain condition is true, this condition is called a *guard*. Guards are written between brackets, following the event that they guard (cf. figure 3.22).

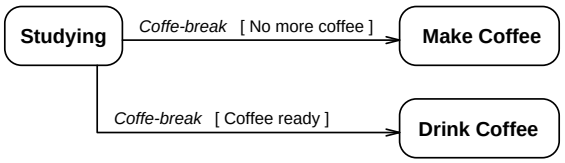

Figure 3.22: The use of conditions as guards

When an object is in a certain state, that particular state is active. The state may be active with some continuous operation, or it may be active with sequential processes. These time-consuming *activities* may be modelled by writing them in the rounded box of the state. Writing "*do:*" before the name of the activity indicates that the activity starts when the state is entered. In figure 3.23 is shown how activities are noted down in the state diagram.

A state takes a certain amount of time. The duration of an event, however, is insignificant compared to the resolution of the state diagram. An *action* is an instantaneous operation,

Figure 3.23: Modelling activities

related to an event. An action is modelled by writing the name of the action beside the name of the event and its conditions, separated by a slash. For clarity, the action is sometimes written in a slightly smaller font size. To model events that do not change the state of the object (for instance, signalling another object), the name of the event may be inside the rounded box. The events of entering and exiting a state are thus modelled by, respectively, *entry* and *exit*. Events may also be sent to other objects. This can be done by modelling an action "*send E(attributes values)*". Another object that receives such an event would then act upon it. Another notation of sending an event is drawing a dotted arrow from a transition to an object class that receives the event (cf. figure 3.24).

In figure 3.24 the complete notation of states, events and transitions is shown, including actions.

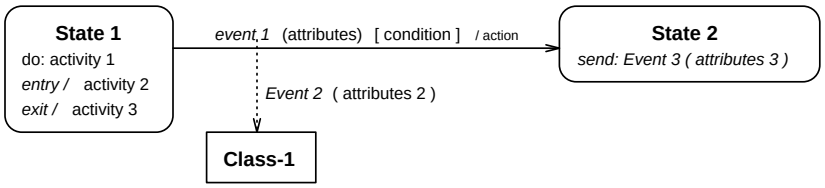

Figure 3.24: The summary of the notation of states, events, and transitions

Structuring

For large problems, an ordinary state diagram would be impractical because of its dimensions. Structuring State diagram into separate state machines solves many of these mapping problems.

We may nest a state diagram into an activity in a higher-level diagram. Such an activity would then be modelled as an ordinary state with input and output transitions. Events can also be expanded into state diagrams. The underlying events are drawn separately as a state diagram. A filled circle indicates the start of a nested state diagram and the bulls eye indicates the ending point. When expanding an event, the dot is labelled with the name of the event that is generated on the higher level state diagram. In figure 3.25 three state diagrams are drawn. The first state diagram is the state diagram that is hierarchically on top. The second state diagram is nested in the state Reading. The third is nested in the event "select(book)". This type of nesting is called hierarchical refinement. An object is in a certain state in every diagram of the hierarchy. It is the "*and*-relationship".

Nested states are actually a form of generalisation on states. When applying generalisation to states, the substates inherit the transitions of their superstate in a manner much like subclasses inherit the features of their superclass(es). The substate may override a transition of its superstate. Generalisation on states is indicated by drawing a *contour*, a rounded box, around all the various substates. In figure 3.26 the notation for generalisa-

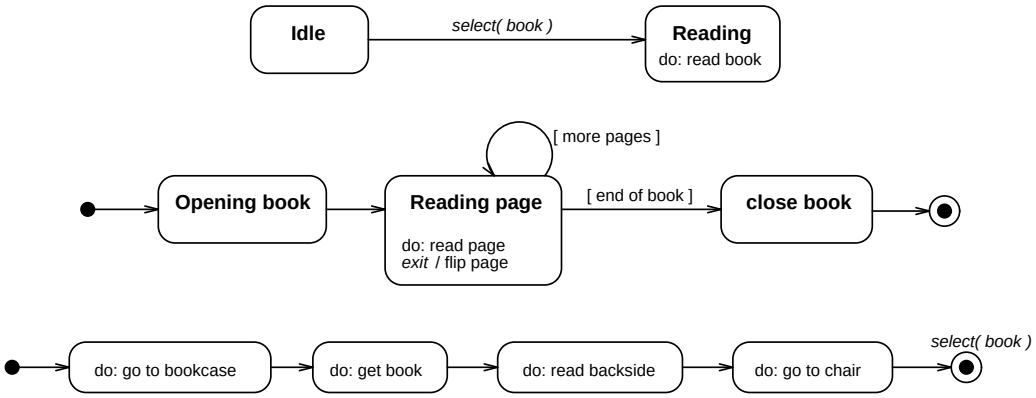

Figure 3.25: Example of nested state-diagrams

tion is demonstrated. With generalisation the object is, unlike with ordinary nesting of states, in just one state.

Besides state generalisation, events may also be generalised. This is modelled by a class diagram of the event. Such a hierarchy provides for different levels of abstraction of an event to be used in different places in a model.

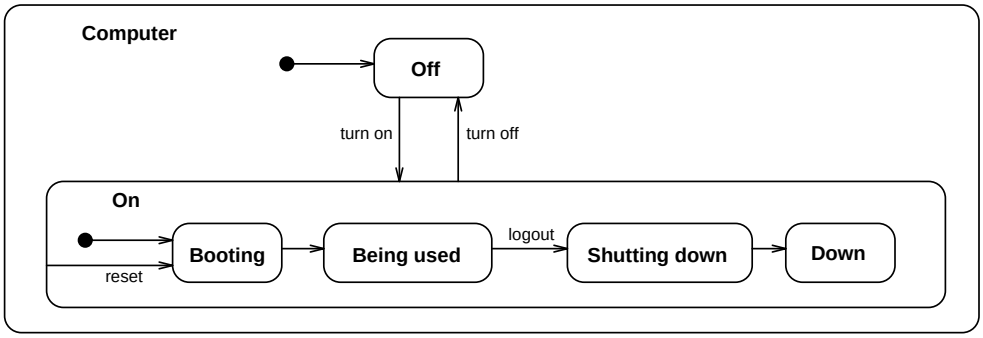

Figure 3.26: Notation for generalisation of states

Concurrency

Since most systems consist of different objects interacting, a dynamic model for one object can never describe the whole system. Modelling concurrency in a dynamic model is therefore needed. When an object is an aggregate of objects, the separate parts may be modelled as separate state diagrams. Guards may be used to communicate between the parts. A certain transition may, for instance, only take place when another part is in a certain state.

When an object can be partitioned into subsets of attributes and links with each having its own state diagram, we speak of concurrency within the object. The object as a whole is then modelled by drawing a rounded box around all substates, separating these substates with dotted lines (cf. figure 3.27).

Modelling concurrency within an object enables the splitting and synchronisation of control of the object. When an object enters two concurrent states from one state, this is

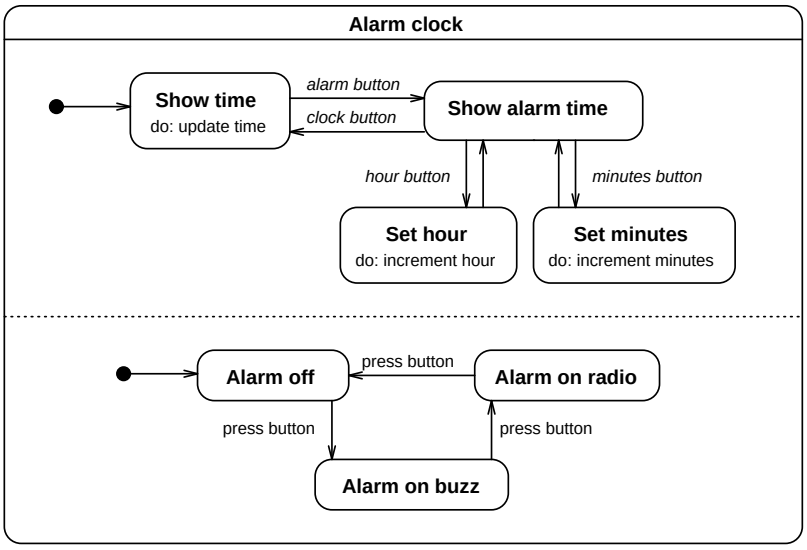

Figure 3.27: The modelling of concurrency within an object

called splitting of control. It is indicated by drawing an arrow that forks from one state to both concurrent states. Synchronisation is modelled by drawing an arrow with a forked tail from both concurrent states to the state in which control is synchronised.

3.2.3 The Functional Model

The third part of OMT is the Functional Model. Here is specified *what* happens, whereas in the dynamic model is specified *when* it happens and in de object model *to what* it happens. In the functional model the implementation of the behaviour of objects (the operations in the object model and the activities and events in the dynamic model) are specified. Traditional data flow diagrams are used to show the flow of values from external inputs, through operations and internal data stores, to external outputs.

A data flow diagram consists of a few basic components: *processes*, *data flows*, *actors*, and *data stores*. Processes describe computations done on input values. The flows out of a process are the results of that operation. A process is drawn as an ellipse containing a description of the transformation of values that the process performs. The input and output values of a process are indicated with data flow arrows. Each process has a fixed number of these arrows. The arrows may be labelled to show their role in the transformation. In figure 3.28 can be seen how a process (in this case a mathematical computation) is modelled as a process with two input and two output data flows.

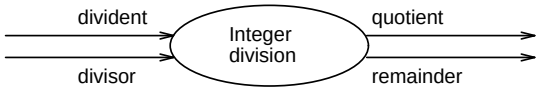

Figure 3.28: Modelling a process

A data flow connects the output of an actor, data store, or process to the input of an actor,

data store, or process. The arrow has a label that describes the data (cf. figure 3.29). Splitting up a data value into subsets, or the sending of one data value to multiple inputs, is drawn as a multi-forked arrow. Combining different parts of a data value into one data

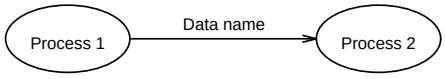

Figure 3.29: Notation of data flows

flow is done by drawing an arrow with a multi-forked tail. Nesting a flow diagram into another flow diagram is done by "hiding" the elements of one flow diagram into one single process which is then as such modelled in another diagram. The first diagram is then "nested" in the second. When working with nested data flow diagrams, it may happen that a certain data flow comes from, or is directed at, a process on a higher level. These arrows have one end open. It comes from, or points at, nothing. They are drawn again, with the same label, in the diagram where the process or object to which they were to connect are drawn.

Actors are active objects that produce or consume the input and output flow of a data flow graph. Therefore they are sometimes called terminators. Their structure should be described in the object model and their functionality in the dynamic model. In figure 3.30 "text" is produced by an actor Person and flows into the process "format" to leave this process as "formatted text".

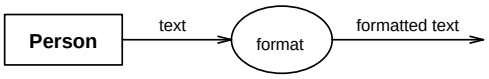

Figure 3.30: Notation of actors in a flow diagram

Unlike actors, *data stores* are passive objects that may hold data for later access. The structure of a data store and the description of the update and access operations permitted must be defined in the object model. Data stores may provide access of data in a different order than stored.

The usage of data stores is illustrated in figure 3.31. Figure 3.31a shows a data store in which a data value is being stored. The value is the result of the process "get coordinates" where the arrow originates from. The value is labelled to indicate which value is stored. In figure 3.31b an update action is illustrated. Updating a values means first retrieving the value, modifying it and storing the result as the new value. The arrow for retrieving and the arrow for storing the arrow are combined into a two-headed arrow. This arrow may be labelled to indicate what data is updated. The retrieving of a value from a data store is shown in figure 3.31c. The arrow is drawn from the data store to the process that uses the value. Again, the arrow is labelled. Unlabelled arrows may also be used. This indicates that the complete data store is used in the process. For example, a lookup table.

Control information, such as the time at which processes are executed, or choices between alternate data paths are not modelled in a flow diagram. This information belongs in the dynamic model, although OMT allows for *control flows* to be inserted in the diagram to improve clarity. These control flows contain a Boolean value that is the result of some

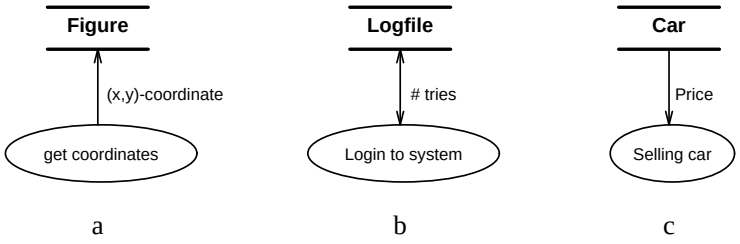

Figure 3.31: Illustration of the usage of data stores

process. This value is used as a guard for a certain receiving process. The notation is an arrow with a dotted line instead of a solid one, as shown in figure 3.32 where the accessing of a file is guarded by the privileges of a user. A control flow is redundant information since all guards should also be modelled in the dynamic model.

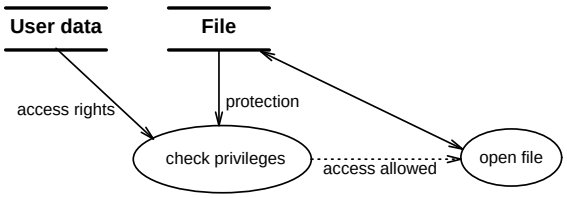

Figure 3.32: Notation of a control flow

In OMT the creation of a new object instance is modelled in the functional model as a data flow that becomes an object. This is drawn as an arrow with an open arrowhead. This notation is not found in the traditional data flow diagrams. In figure 3.33 an example is given.

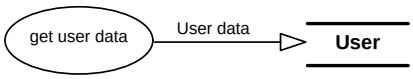

Figure 3.33: Modelling the instantiation of an object in the functional model

A data flow diagram is particularly useful for showing the high-level functionality of a system and its breakdown into smaller functional units. Each process may be expanded into a new data flow diagram and so enhance clarity of a complex system.

4 Methods and Documents

In this chapter the topic of methods is covered more deeply and the concept of generic methods is explained. In the last part of the chapter, document methods and their place in the object-oriented life cycle are discussed.

4.1 Methods

In section 2.2 methods were briefly introduced and some characteristics of methods were given. That particular view upon methods is not universal. [LR93] discusses how several views upon methods exist:

- [Bal90] looks upon a method as having a one-to-one relationship with a development phase and being the basic part of the development process. He calls it a *Basic Technique*.
- [OHM+91] look at the method as being the whole development process and take one development step as the basic part of the process. Important in their view is that a method produces a *document*.
- [Ste91] shows a more explicit definition of a method:

  Method $\hat{=}$ Name + Notation + Rules

  In this way he requires the method to have a name, a notation (language) to describe the specification, and explicit rules on how to apply the method. These rules are a specific set of procedures that guarantee a uniform structure of the produced documents. Furthermore he rquires evaluation criteria to compare alternative specifications.
- [Wie91] describes the structure of a method as:

  Method $\hat{=}$ Specification language + Practical rules + Models

  Here the practical rules consist of three parts. First, the tasks that have to be performed to develop a model. Second, the order of these tasks in particular cases. And third, heuristics for these tasks. He says that especially the development of specifications which meet the design criteria better, require the statement of some structured semantics for the method.

From these views, [LR93] has developed the definition of a method structure as is illustrated in figure 4.1. The main difference between this definition and the definitions that are usually found in literature is that it defines a method recursively. A method may be a part of another method or a method may consist of multiple methods as parts. The recursion stops when a method is nothing else then a single basic development step. These are the basic elements of methods. The collection of these elements holds basic development steps like "Evolute", "Integration of models", etc. With the method definition of [LR93] both these basic development steps as well as complex combinations of methods can be used like methods. With this view upon methods the complete development process of a software system may be considered to be a method.

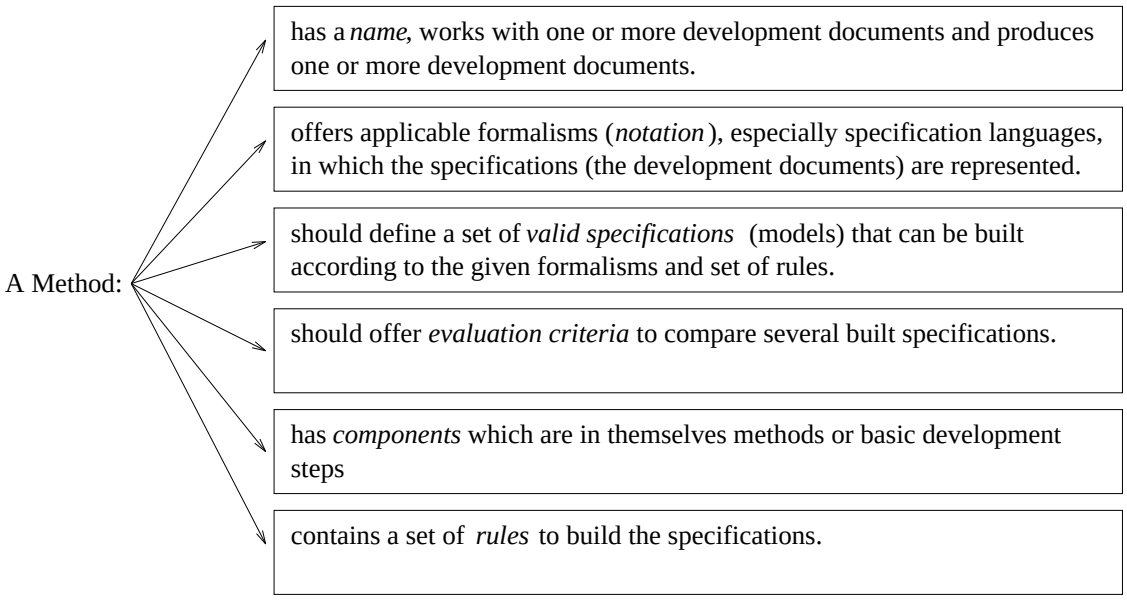

Figure 4.1: The definition of a method structure

To illustrate how a specific method fits into the above definition, the following example regarding OMT is given:

Name: "Object Modeling Technique"

Notation: Object model, dynamic model, and functional model notations.

Rules: (for instance) "Underline the nouns in a text to find the object classes".

Valid specifications: Defined by OMT-Sematics

Evaluation criteria: (for instance) "Ternary relationships are unwanted".

Components: Techniques to develop the object model, the dynamic model, and the functional model.

4.2 Generic Methods

In the above section, a definition of methods was built. In this section the next step is taken: the step from concrete methods towards generic methods (cf. [LR93]).

To do this we take a good look at development methods. We see that the foundation of a method is a collection of fundamental *development steps*. These steps are executed in a certain order defined by the method. We call the collection of development steps found in a method the *method core*. There are not many of these development steps. All development methods are built from the same small set of fundamental development steps, where some steps may be found in many methods and some may be found in few methods. The complete set of these development steps is called the method core for development methods, and [LR93] distinguishes the following steps to form such a method core:

- Evolution of specifications.
- Split up of specifications.
- Abstraction of specifciations.
- Integration of specifications.
- Representation of specifications.
- Syntactical check of specifications.
- Analysis of specifications.
- Validation of specifications.
- Generation of test data.

From this method core for development methods, *generic* method cores may now be developed. In this context, "genericity" means that the method cores are formalised with generic parameters. By assigning values (formal parameters) to these generic parameters, the generic method core produces a specific, concrete method core. Thus, the generic parameters take away anything that is related to a specific method. Generic method cores are used to construct generic methods.

To illustrate how genericity is used a closer look at one of these steps may be helpful:

With "evolution of specifications" is meant the construction of a specification by adding, deleting, and modifying parts of it, starting with an empty specification and developing it further. The information needed to perform an "evolution" step are formalised into parameters. The parameters needed are the specification that is to be changed, the affected element, the new element and how it is integrated into the specification, and the used notation. The sort of evolution step that is taken is defined by the instantiation of the parameters. Adding a part to the specification is indicated by the instantiation of a new element, and not an affected element. The reverse is true for deleting a part of the specification, where there is no new element, but just an affected element, instantiated. Modifying a part is indicated by instantiating both the affected element and the new element. The result of the evolution step is a changed specification. Formalised with generic parameters it would look like:

Evolution (Specification, Affected Element, New Element, Integration, Notation) : Specification

This is the generic definition of the evolution step. It is not part of any method in particular but rather part of all methods. By instantiating the parameters with method-specific values, we define a evolution step for a specific method. By doing so for every development step from the method core, we can engineer a method. For the OMT object model the formalised evolution step with instantiated generic parameters (formal parameters) would look like:

Evolution (OMT-Object Model, Object Model of Affected Element, Object Model of New Element, Object Model Integration, OMT-Specification language) : OMT-Object Model

By this the evolution step is adapted to a specific method: OMT. Assigning values to generic parameters is essential to define a concrete method. When actually developing something, these concrete method steps are executed. This means that the formal parameters of the development step are then instantiated with actual values. To take the example of OMT one step further: The instantiation of the evolution step in OMT could look like:

Evolution (Objectmodel for "Student", <EMPTY>, OMT-attribute "last_name", "Add as OMT-attribute to class Student", Grammatical and context-sensitive rules of OMT) : Objectmodel for "Student"

In this development step, the argument <EMPTY> for the "Affected Element" indicates the addition of a new element. The new element "OMT-attribute last_name" is added to the "Object Model for Student" as an "OMT-attribute to class Student" according to the "Grammatical and context-sensitive rules of OMT ". This could be a normal development step when building an object model of a student.

Similar to the evolution step, the other development steps can be written in their generic form as follows:

Split up of specifications:

This development step is used to refine or decompose specification elements. The parameters are the specification from which the element is taken. The affected element is the element that is taken out of the specification. The specification part is the new specification that is formed by the element that is taken out of the original specification. The integration rules and notation are the same as with the evolution development step. The result is an altered specification. In generic form:

Split_Up (Specification, Affected Element, Specification Part, Integration, Notation) : Specification

Abstraction of specifications:

This development step is used to build an abstract of part of a specification. Instead of representing the whole part, the part is represented as a single object with only the relevant details modelled. This makes the higher-level specification clearer and easier to "read". The parameters involved in this development step are: The specification, the affected specification part, the abstract element and its integration into the specification and the method-specific notation. As a result, the new specification is given. In generic form:

ABSTRACTION (SPECIFICATION, SPECIFICATION PART, ABSTRACT ELEMENT, INTEGRATION, NOTATION) : SPECIFICATION

Integration of specifications:

With this development step two different specifications are integrated into a single one. The parameters for the generic form are the two specifications, some indication of where they connect to each other, the integration rules of the method and the notation. The result is a single new specification. The generic form looks like:

INTEGRATION (SPECIFICATION, SPECIFICATION, { CONNECTION POINTS }, INTEGRATION, NOTATION) : SPECIFICATION

Representation of specifications:

With this development step the specification, or parts of it, are represented. The parameters needed to do this are the specification, a selection of elements that should be represented, and the notation in which to represent them.

REPRESENTATION (SPECIFICATION, { REPRESENTED ELEMENTS }, NOTATION) : SPECIFICATION

Syntactical check of specifications:

This development step takes a part out of a specification and checks the syntactical correctness of it against the given notation and rules. When finding an error it gives a message to say so. It also gives a position and an explanation. Its generic form looks like:

SYNTACTICAL_CHECK (SPECIFICATION, SPECIFICATION PART, NOTATION) : ({ NOTIFICATION, POSITION, REFERENCE })

Analysis of specifications:

This development step tells something about a specification. A qualitative analysis tells if some element is used in a specification. As a result it gives a Boolean value. If this value is TRUE, the position of the first occurrence is given. Qualitative analysis tells how often a certain element is used in a specification. As a result it gives an integer value. The generic form of these development steps is:

QUALITATIVE ANALYSIS (SPECIFICATION, SEARCH ELEMENT, NOTATION) : (BOOLEANVALUE, { POSITION })

QUANTITATIVE ANALYSIS (SPECIFICATION, SEARCH ELEMENT, NOTATION) : NUMBER

Validation of specifications:

This development step produces test results for the developer to see if the specification specifies exactly what he/she wants. The parameters needed for this step are the specification, test data to produce the test results with, the specification part that is to be validated, and the notation to be used. The results are errors (if any) and test results. The generic form is:

VALIDATION (SPECIFICATION, TEST DATA, SPECIFICATION PART, NOTATION) : ({ ERRORS }, TEST-RESULTS)

Generation of test-data:

The generation of test-data is used for several purposes. When the test-data is random, it may be used to find errors in the specification. Realistic test-data may be used for walk-throughs, or prototyping. Simple test-data may be used for validation purposes or benchmarks. The sort of test-data that must be generated is set with the "quality" parameter. Other parameters are the specification, the specification part for which test-data must be generated, and the notation used. The result is the required test-data.

GENERATE TEST-DATA (SPECIFICATION, SPECIFICATION PART, { QUALITY }, NOTATION) : (TEST-DATA)

Considering the fact that every method is built out of a method core, and considering that all development steps in the method core can be written in their generic form with input and output parameters, would it be possible to write down a method in its generic form? To do this it must be possible to order the development steps. The following constructors are used to enable that:

SEQ $p_1\ p_2\ \ldots\ p_n$

Sequence
This constructor indicates a sequential order of the steps or partial methods following it.

PAR $p_1\ p_2\ \ldots\ p_n$

Parallelity
This constructore indicates the parallel execution of the steps or partial methods following it. Synchronisation takes place before the first step that falls out of the scope of the PAR-constructor is executed.

IF $Cond_1\ p_{11}\ p_{12}\ \ldots\ p_{1n}$
$\ldots$
$Cond_m\ p_{m1}\ p_{m2}\ \ldots\ p_{mn}$

Conditional selection
This constructor indicates which sequence of steps or partial methods are executed depending on a certain condition. Only if this condition is found to be true, the seqence is executed. The conditions are each checked sequentially.

CASE *Epression*
IndexValue$_1$ p_{11} p_{12} ... p_{1n}
...
IndexValue$_m$ p_{m1} p_{m2} ... p_{mn}

Selection dependent on value
This constructor selects the next (sequence of) steps depending on the value of an expression.

WHILE *Condition* p_1 p_2 ... p_n

Conditional iteration
The WHILE constructor is used to indicate that when the method comes to this point, all steps and partial methods in the scope of the WHILE will be iterated for as long as the condition remains true. When the condition is false when the method first reaches it, the sequence of steps in its scope is not executed at all.

FOR *Index IndexValue*$_1$ *IndexValue*$_2$ p_1 p_2 ... p_n

Iteration
The FOR-constructor iterates the sequence of steps in its scope as often as the difference between the two IndexValues.

With these constructors the steps in a method may be ordered. Writing down generic methods is now possible except for the part that is found in every method: the creative part. Every method has parts where the person that uses the method decides to add an item or decides to validate the model. Therefore a "Creative" development step is formalised:

CREATIVE (MODEL, {RULES}, {VALIDATION CRITERIA}, NOTATION) : SIGNAL

The idea behind a creative part in a method is that the designer takes the model, and the knowledge of the notation, together with his own validation criteria and rules (which are impossible to formalise), and makes a decision. This decision is inserted in the method as a signal that can be either "*continue*", "*break*", or "*stop*". As long as the signal is "continue" the user is happy with how the development is going and wishes to stay in the part of the method that is dependent on the "continue" signal. When the developer gives a "stop" signal, he indicates that he wishes to leave the part of the method that is dependent on the signal and wants to enter the next part of the method. A "break" signal indicates that the developer decides to stop using the part of the method that is dependent on the signal, perhaps because of dissatisfaction with the results or other (unknown) reasons and wants to go back to the last point where he was completely satisfied with how the development went. This means going back to the last time that a "continue" signal was given and continue the method at this point.

It is now possible to write down the generic form of a method as illustrated in the following example. The header of the example is a formalised notation of what was defined in figure 4.1.

METHOD (COLLECT INFORMATION, { NOTATION }, { RULES }, { EVALUATION CRITERIA }, < REPRESENTATION, CREATIVE, EVOLUTION, ABSTRACT, SPLIT-UP >, { SPECIFICATION }) : { SPECIFICATION }

SEQ
REPRESENTATION (SPECIFICATION, { REPRESENTED ELEMENTS }, NOTATION) : SPECIFICATION
WHILE CREATIVE
EVOLUTION (SPECIFICATION, AFFECTED ELEMENT, NEW ELEMENT, INTEGRATION, NOTATION) : SPECIFICATION
IF
Specification unclear
SEQ
ABSTRACT (SPECIFICATION, SPECIFICATION PART, ABSTRACT ELEMENT, INTEGRATION, NOTATION) : SPECIFICATION
REPRESENTATION (SPECIFICATION, { REPRESENTED ELEMENTS }, NOTATION) : SPECIFICATION
CREATIVE
SEQ
SPLIT UP (SPECIFICATION, AFFECTED ELEMENT, SPECIFICATION PART, INTEGRATION, NOTATION) : SPECIFICATION
REPRESENTATION (SPECIFICATION, { REPRESENTED ELEMENTS }, NOTATION) : SPECIFICATION
Specification has multiple abstraction levels AND CREATIVE
REPRESENTATION (SPECIFICATION, { REPRESENTED ELEMENTS }, NOTATION) : SPECIFICATION

In this example, the method describes how a software developer collects information and puts it into a specification for as long as he/she wants to. Every time a new part of the specification is added, removed or modified, the specification is checked on clearness. As long as it is unclear, parts of the specification are abstracted to enhance clearness. When the specification becomes to complex, the developer may decide to split the specification up into two new parts. When the specification has multiple abstraction levels, the software developer may decide to have the specification represented with an extra abstract level. When satisfied with the representation he/she may continue with adding, removing or modifying parts and go through the whole loop again.

4.3 Documents

The result of a method is a document that is being developed. In the following section I will first present the results of a study done by [LRR93]. They have developed an object-oriented view on a life cycle with the document as central object. The concept of methods is not involved in this view. In the second part of this section I describe why the described document objects can not be connected with method objects. At least, not in their presented form and not with the concept of a method as described in the first section of this chapter.

4.3.1 Document Objects - A study

In this section an object oriented view is given on documents (cf. [LRR93]). The concept of a method object and its role in the development process is not mentioned here.

The choice of the document centered view in the development process is not a surprising one. A well developed document is the main goal of the development process. Documents go through a lot of changes and well designed documents have a uniform, incremental, and modular structure which make them eligible for reuse.

To incorporate documents into the object-oriented life cycle model, we look at the collection of documents produced during the development of an application. Each individual document is in a particular development state at a given time. Some documents may be operational while others are still in the requirements analysis state. In figure 4.2 this concept is illustrated by an application context bubble in which the complete set of documents from the development process are captured. Each circle and square is represents a document. The particular development state is indicated by the color.

Besides giving examples of the individual document states, figure 4.2 also shows that certain associations between documents exist. [LRR93] have identified two different types of associations between documents. A *part_of* association and a *depends_on* association.

The *part_of* association is an association that exists when one document is composed of other documents. It is indicated in figure 4.2 by a solid arrow pointing from the composite document to its part. For instance, documents *u3.2*, *p4.2*, and *p4.3* in the figure are each *part_of* document *p4.1*. The composite structure of a document may come into existence when a certain document becomes to complex or to unclear. When this happens it is split up and form two or more new documents. The original specification is now a composition of the new documents. Existing documents, for instance from libraries, that are used in a specification may be embedded into a document using a *part_of* relationship.

The *depends_on* association exists when a certain document is influenced in its development by another document. The documents concerned do not have to be involved in an aggregate relationship with one another. In figure 4.2 this type of association is indicated with an arrow with a dotted line, pointing from the document that influences towards the document that is dependent. The *depends_on* association between documents is instantiated by either timing or management decisions in the development process.

The combination of documents (the application context bubble) indicates the state of the

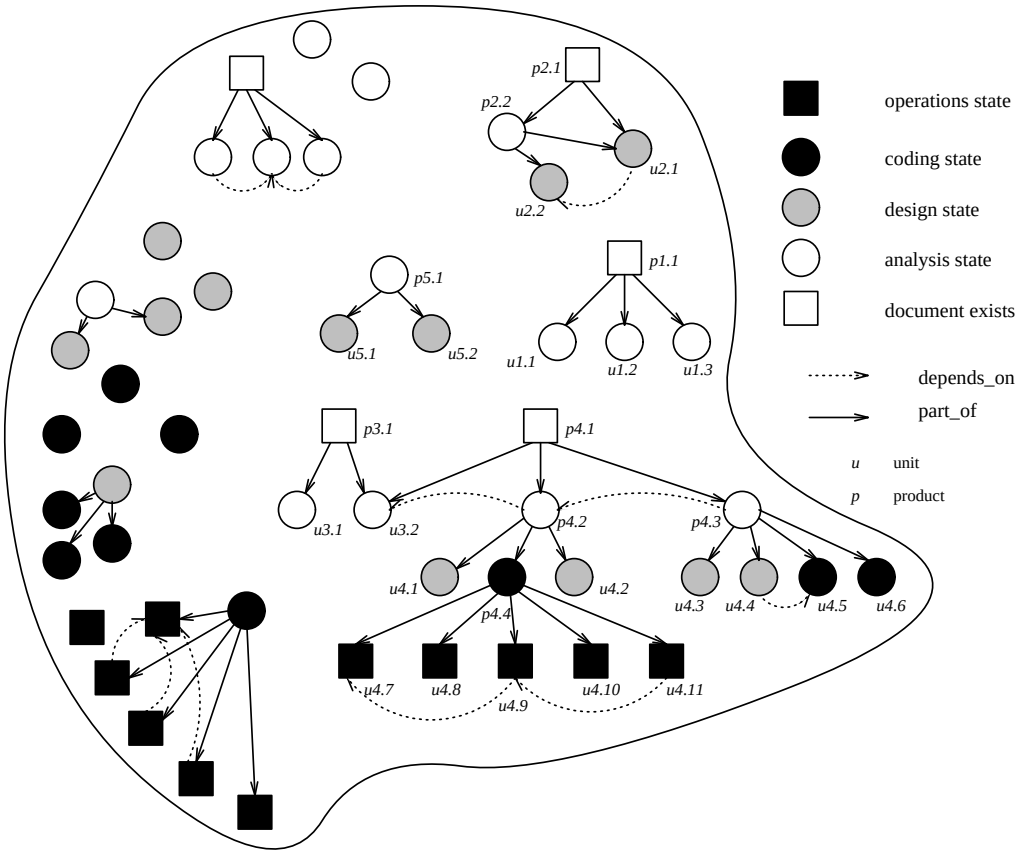

Figure 4.2: An application context bubble for software development

individual documents used and produced in the development process. Each document has its own development life cycle that it goes through. The state of the whole development process is reflected by the combination of the states of the documents. In this document-centered view a software life-cycle no longer has a global execution order.

The composite nature of a document induces two main behavioral patterns. When a document consists of other documents its behavior is mainly concerned with the development and integration of its components. When a document does not have any components, its behavior is concerned with its own development. Thus, a document may be modeled by two types of objects: The **product** and the **unit**. Where a Unit may be involved in a aggregate relationship with a Product or not, but a Product is always composite structure with Units. In figure 4.2 this is indicated with the characters '*p*' or '*u*' written next to the document. Here '*p*' means that the document is a product, and '*u*' means that it is a unit. [LRR93] make this split up very explicitely. When a unit becomes to complex, it is split up into new units and a product is instantiated that is composed of these units. In the document centered view that is presented in [LRR93] these two objects differ in structure and behaviour. In figure 4.3 the Unit class and the Product class are represented.

As we can see in this figure, the operations of each class are very different. The operations of the Unit are mainly for development of the Unit, the operations of the Product are mainly for structuring the document.

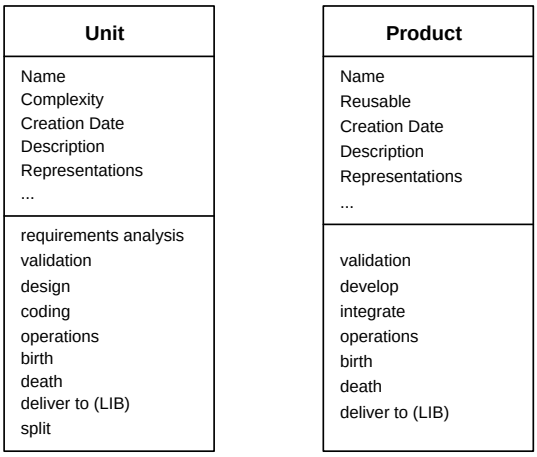

Figure 4.3: The Unit class and the Product class

In figure 4.4 the dynamical model of a Unit is given. We can see even better that its entire behaviour is concerned with development.

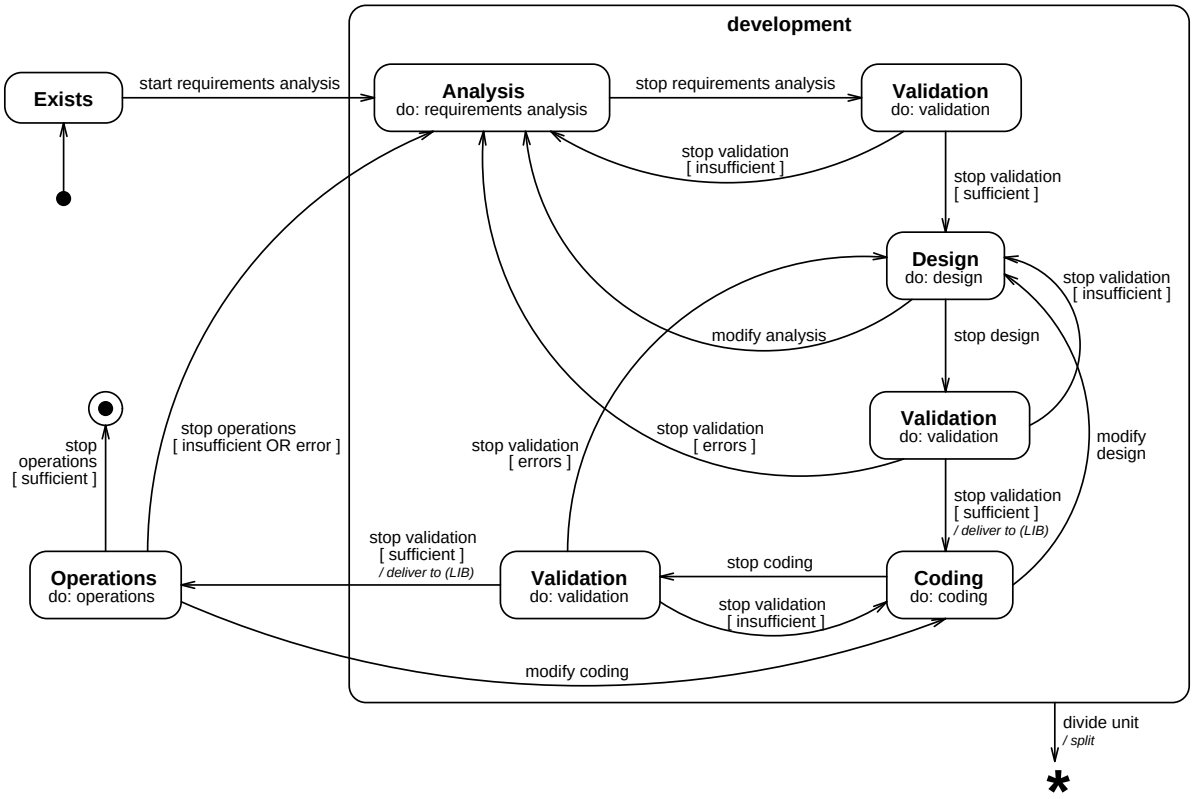

Figure 4.4: The dynamical model of a Unit

After a Unit comes to birth, it goes through a complete development life cycle. The life cycle that the state chart describes is very much like that of the iterative waterfall model. (cf. figure 2.2). An important aspect of this variant is the introduction of deliverances for

reuse. These deliverances are added after the validation state following the design state and after the validation state following the coding state. When the unit becomes to complex it goes into the state indicated with the asterisk (*). In this state the restructuring of the document takes place. The exact implementation is not given, but we may imagine the unit to give birth to two new units and a product, setting up the associations between them and killing itself after copying parts of its contents into each of the new units.

The dynamical model of a Product object are illustrated in figure 4.5.

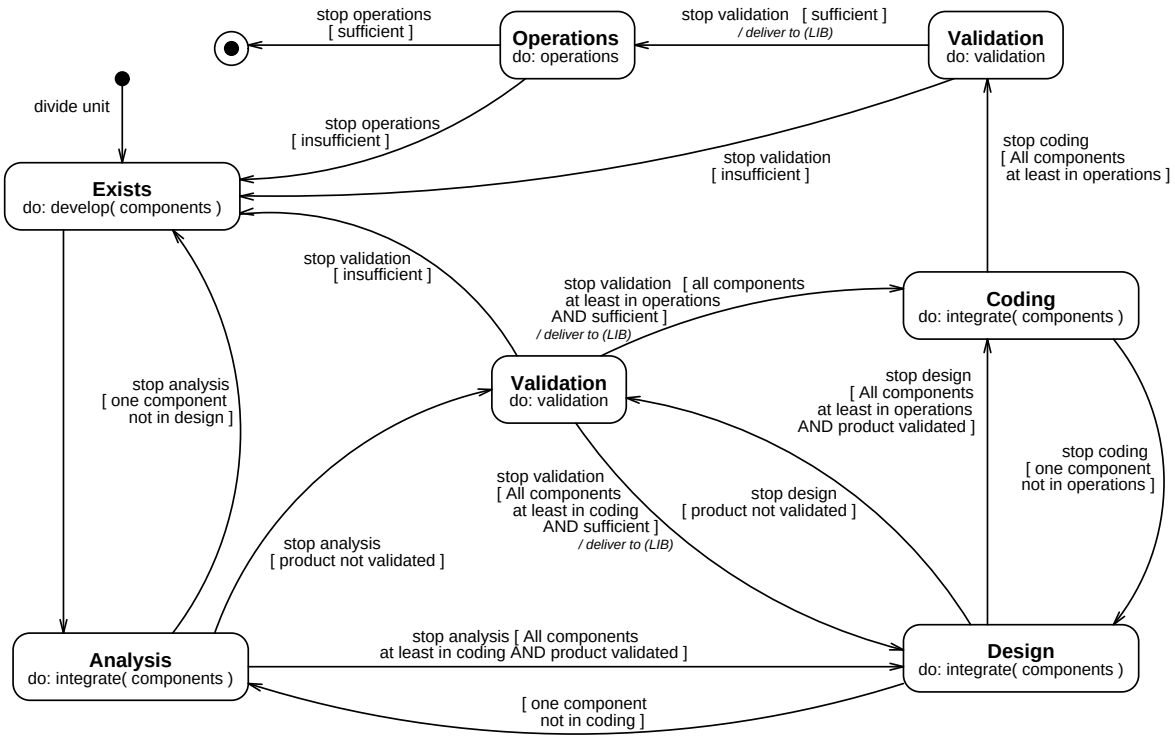

Figure 4.5: A dynamical model of a Product object

The states depicted in figure 4.5 reflect the development states that the Product may be in during its life cycle. As we can see, the states of the Product are mainly concerned with developing its components and integrate them into a final product. Also here the reusability priciple is supported by deliverances of the document to a library whenever a document has successfully left a validation state.

4.3.2 Overlap between Document Objects and Methods

The described document structure in the previous section, is quite complex. The document is actually two types of objects which form an aggregate relationship with each other. It possesses states that reflect the documents place in the development process. The transitions between these states are triggered by advancement in this development process.

A method is designed for a certain development process. While using the method, the document that will eventually be the product is developed using the defined steps in the

method. These steps are specific for stages in the development process: A method may contain an "analysis" stage or a "coding" stage. The state of a document is thus defined by the method. Namely, the part of the method in which the document is used defines its state.

What [LRR93] have defined as a document is a combination of a method and a document structure. Thereby giving extra attention to the software engineering principles modularity, incrementality, evolvability and reusability. The latter by defining an aggregate structure for the document object and by embedding the concept of building a document library in the model. In figures 4.4 and 4.5, a variant of the iterative waterfall model is used to show the life cycle of such a document. In the earlier section of this chapter, ordering of development steps was defined within a method. Thus, the life cycle of a document is captured in the method that is used to develop that document.

To model the document object in a method-centered view on object-oriented life cycles, we take away from the model of a document the part that overlaps with the model of a method. We are then left with the two aspects of a document object that are not modelled in the model of a method:

- the aggregate structure of the document, and
- the building of a document library.

These aspects will function as building blocks for the document object that is added to the model of the method object, developed in the next chapter.

5 Method Objects

In Chapter 4, methods were split up into a method core and a method-specific part. We saw that a method core consists of several basic development steps. We also saw how we could write the generic form of a method. In the second part we saw the introduction of an object-oriented view on documents and their place in the object-oriented life-cycle model. In this chapter an object-oriented model of a method is given. The resulting method objects are associated with users and engineers and the structure of the documents is extended to fit into the model.

5.1 Object Model

In chapter 4 the generic structure of a method is developed (cf. [LR93]). We saw that the structure of each method consists of other methods and basic development steps which are combined using constructors. A particular method is then engineered by building a generic form of that method of which the generic parameters are then instantiated. The method that is thus instantiated, may be used by issuing values for its formal parameters.

The Method, and Generic_Method classes

When we look at a method as an object with its own structure, we see the generic method as a second object that describes it. The generic method is the pattern of a method. In figure 5.1 this is modelled accordingly.

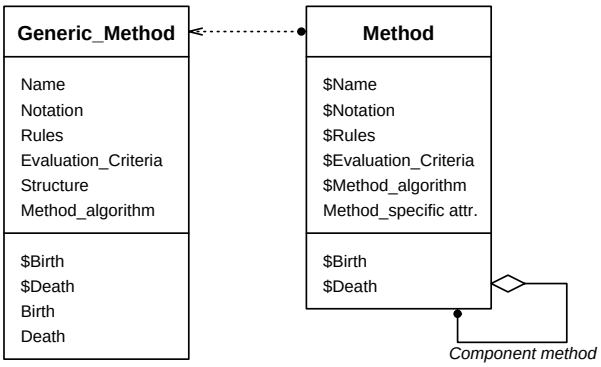

Figure 5.1: Modelling a generic method as a metaclass

As we can see in this model, some of the attributes of the Generic method are class attributes of the method. Although the attributes in this model are just a suggestion for a set of attributes, it can be seen that they are derived from the definition of a development method (cf. figure 4.1). The components of the method are related to the method by the aggregate relationship "Component method". The number of components in a method is not fixed to a maximum, and may be zero. Therefore, the multiplicity on the components end of this relationship is modelled as "Many". The multiplicity of the instantiation of a generic method is also "Many" because the number of methods is also not fixed to a maximum.

Figure 5.2 is an instantiation of the model in figure 5.1. In it, the structure of the Method class and its describing class Generic_Method is illustrated for the OMT-method.

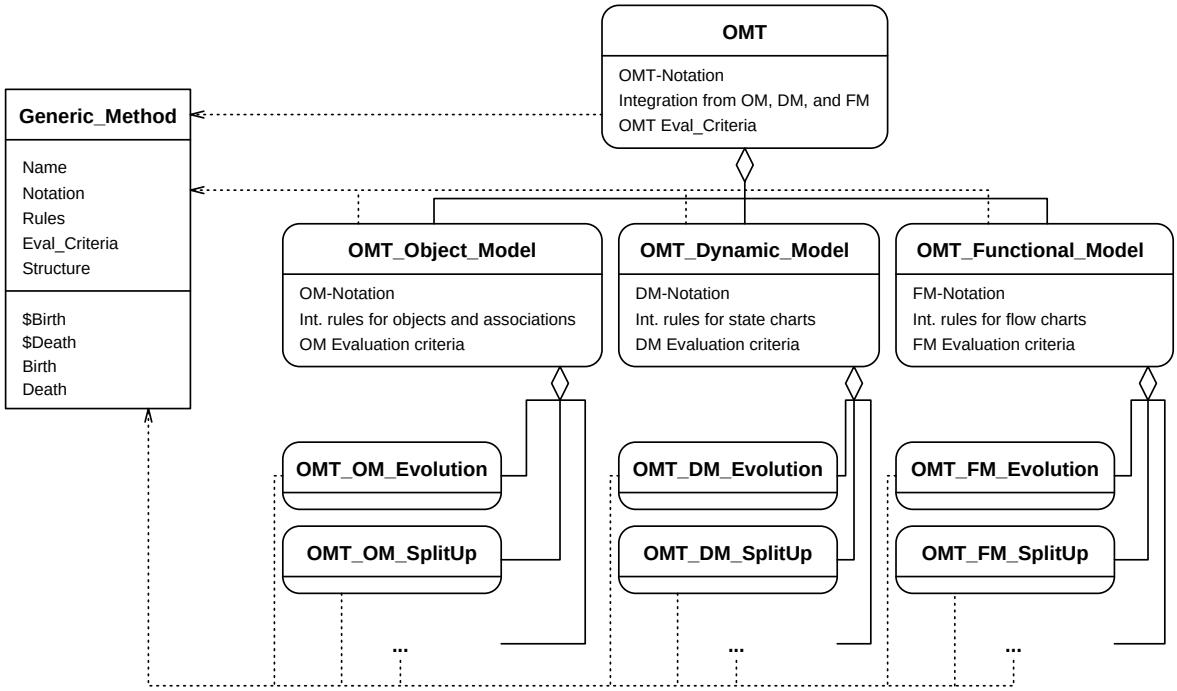

Figure 5.2: An instance diagram for the OMT-method

As you can see, all methods are instantiations of the Generic_Method class. They are associated in an aggregate relationship with each other. We can imagine the engineering of OMT as a bottom-up process where first the component methods are designed and instantiated. Next, the method that consists of these components is designed and instantiated. And so on.

The Person class

It is obvious that a person plays a role in using and designing methods. Therefore, we include a class Person in our model. The question now is what association(s) exist between a person and a method.

A person in the role of *software developer* uses one or more methods. We model this as an association "Uses" between the Person and Method class. A particular person does not have to be using any method at all. On the other hand, the maximum number of methods that any person is associated with is limited only by the developer himself. The multiplicity on the Method side of the association is therefore "Many". The multiplicity on the other side of the association, the Person side, is also "Many". This is because it is open how many developers may be associated with a particular method, and a method may not be used by any developer at all.

What other associations exist between a method and a person? A method has to be developed. Therefore a person can take the role of *method engineer*. A method engineer designs the method using existing methods, basic development steps and constructors. We

model this as an association "Designs", between the classes Person and Generic_Method. Technically, the method engineer builds a generic method and instantiates the generic parameters of it so that a method with only formal parameters to instantiate (i.e. the class Method) remains. Any number of engineers may be working on a method but at least one must be. This explains the multiplicity "One or more" on the Person side of the association. The multiplicity on the Generic_Method side of the association is "Many" because an person may design any number of methods but doesn't necessarily has to.

A third association in which the class Person is involved is when a person doesn't use a particular method, doesn't design that method, but is just familiar with it. This association "Is_familiar_with" exists between the class Person and the class Generic_Method. The explanation for this is that familiarity with a certain designed method means that an instance of the class Person is familiar with the complete class of that particular method, i.e. an instance of the Generic_Method class. The multiplicity on the Person side is "One or more" because there is always one person familiar with a certain method (the engineer) and there is no limit on how many persons are familiar with a specific method. On the Generic_Method side the multiplicity is "Many", for any person may know several methods but does not necessarily have to know any.

After adding the class Person to our model in figure 5.1 we get figure 5.3.

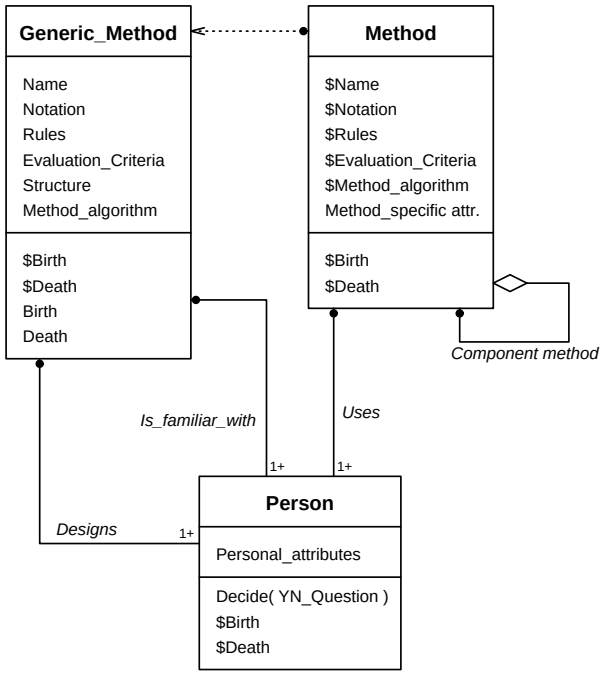

Figure 5.3: The class Person added to the class diagram

The Document class

To build a good model for a document, we have to ask ourselves the question: What exactly is a document? In my view a document is an object that is created and developed by using a method. A method always acts on one or more documents. A document may be a structure of document parts or may be a single part itself. The way a document fits

into the model so far is illustrated in figure 5.4.

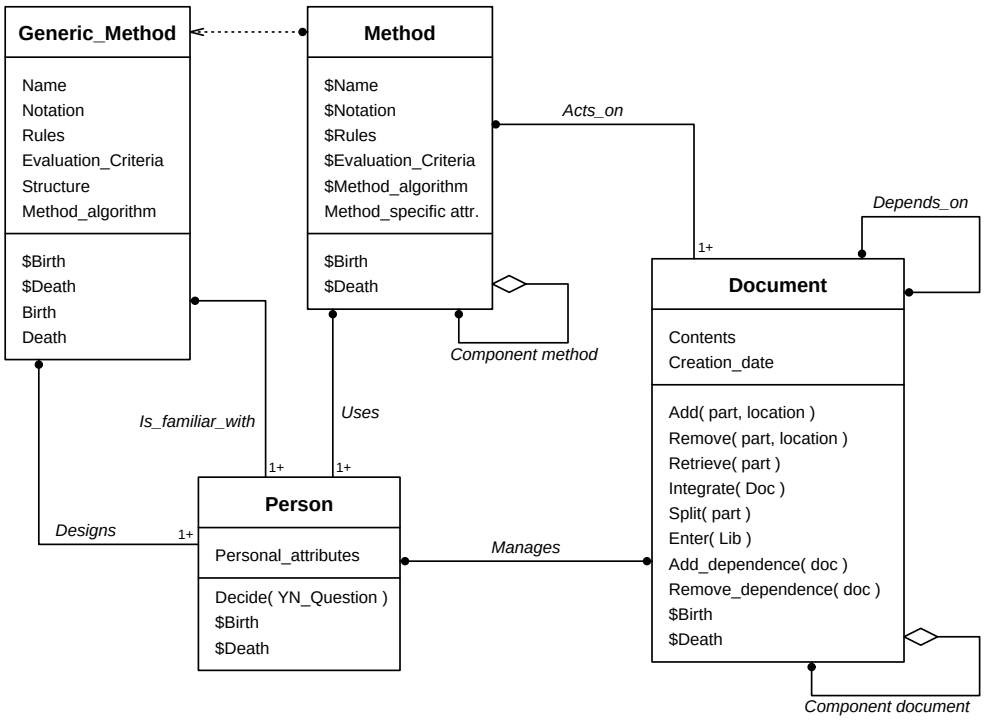

Figure 5.4: The class Document added to the class diagram

The multiplicity of the "Acts_On" association is "Many" on the Method side. Multiple methods may act on the same document and a method may also exist without a link to a specific method. The "One or more" multiplicity on the Document side is explained by the fact that a method may act on multiple documents, but always at least on one.

The associations that a document has with itself are discussed in section 4.3.1. The *Component_document* association has a multiplicity of "Many" because a document may be a composite of any number of other documents, but doesn't necessarily have to have any components at all. The multiplicity of the *Depends_on* association is "Many" on both sides. Any document may be dependent on any number of documents but doesn't necessarily have to be dependent on any. In the other direction, the document may influence any number of other documents but doesn't necessarily have to.

There is also an association between the Document and the Person class. This association, *Manages* exists so that the software developer, or the document manager may influence the status of a document. He may create documents, send documents to the library, or may build *Depends_on* links. As said in section 4.3.1, there are two ways a *Depends_on* links is instantiated: time induced or management decisions. In this model, the time induced links are a directly defined in the method and its structure. They do therefore not exist as instantiations of the *Depends_on* association. Management decisions however are made by a person and do indeed instantiate *Depends_on* links.

5.2 Dynamical Model

First I will set up dynamic models of the individual classes, starting with the Method object itself. Where and how a method communicates with other classes is not yet embedded in this model. Before we can do that we have to build a better understanding of each of the classes individually.

The Method class

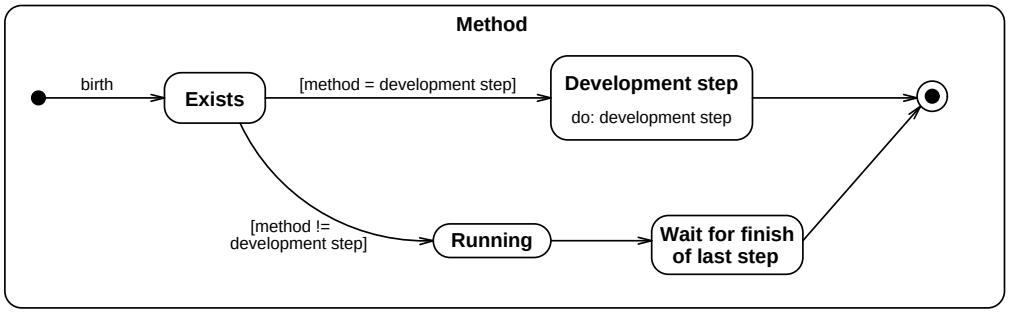

Figure 5.5: Top level state chart of the dynamic model of the Method class

Figure 5.5 is the top-level state chart of the dynamic model of the Method class. When a method is instantiated it comes into the Exists state. From there on there are two possibilities. The first is that the method is a basic development step. In that case the step is performed. The development steps use documents and send events to these documents like *Add(part)*, *Remove(part)*, *Split(part)*, *Retrieve(part)*, and *Integrate(document)*. When the development step is completed, the method comes to an end.

In the case that the method is not a basic development step we may conclude that it is a structure of methods and development steps combined using constructors (cf. [LR93]). Thus, the method goes into a state Running in which all steps in this structure are executed. When the method leaves this state, it will not end until all the methods that were initiated from it are ended, so it enters the state Wait for finish of last step. Finally the method comes to an end.

In the state Running, the structure of the method is handled. In figure 5.6 its state chart is given. A method leaves the state Running whenever the next step in the structure is another method. This method is initiated by sending a *birth* signal just before the "parent" method leaves the Running state. When the next step is a constructor, the method enters a state that handles that particular constructor.

The constructors that are used in the structure of a method are SEQ, PAR, IF, CASE, WHILE, and FOR (see section 4.2). In figure 5.7 the states that a method enters when handling a SEQ-constructor are modelled in a state-chart. We see that the first state the method enters in is again the Running state. This state is described above and in figure 5.6. A method can be in this state in multiple levels at the same time. This particular case of nesting[2] is modelled because the constructors may be nested too. For example, a method may be in the state Running while handling a SEQ-constructor, and at the same

[2]See figures 3.25 and 3.26

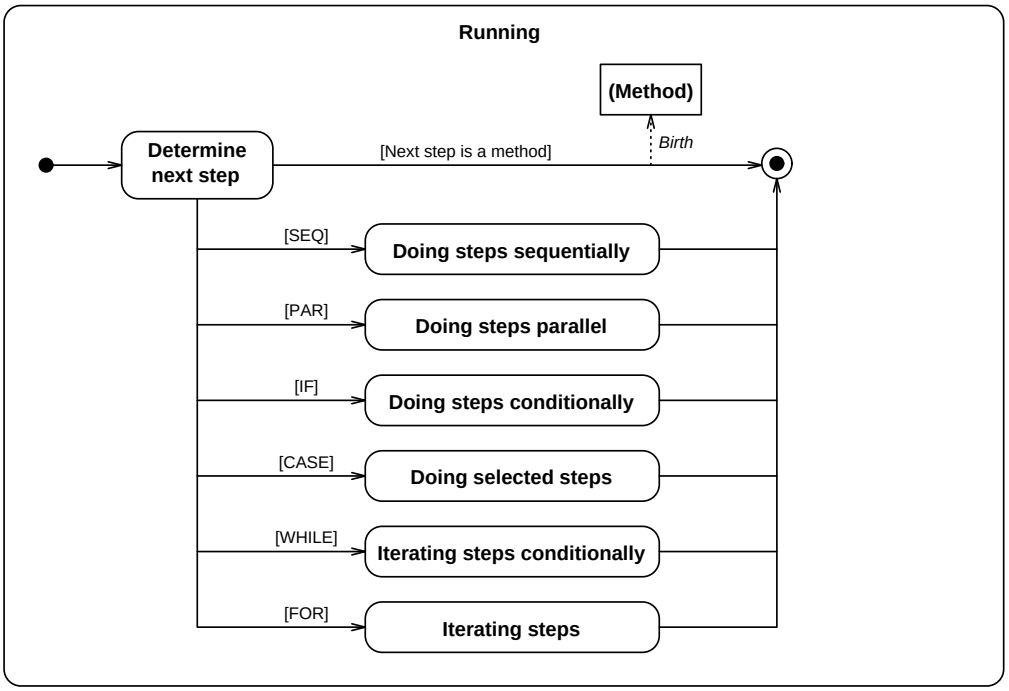

Figure 5.6: The nested state chart of the Running state

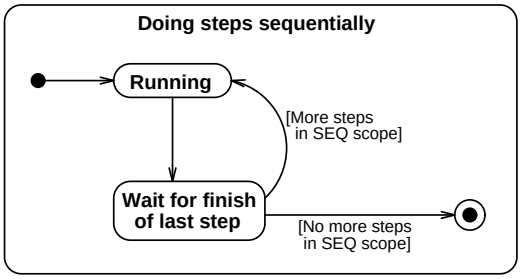

Figure 5.7: The nested state chart of the Doing steps sequentially state

time be in the state Running while handling a PAR-constructor that falls within the scope of the SEQ.

After leaving the state Running in the state chart of figure 5.7 the method waits for the last step that was handled to finish. It then either executes the next step in the scope of the SEQ or, when there are none left, leaves the state Doing steps sequentially altogether. The fact that the method waits here for each step to finish before it executes the next step is exactly what makes the SEQ-constructor different from the PAR constructor. The state chart of the state Doing steps parallel is drawn in figure 5.8.

Here the steps within the scope of the PAR are handled all immediately after each other without waiting for the one before to finish. When there are no more left, the method waits until all the steps within the scope of the PAR are finished before leaving the state Doing steps parallel.

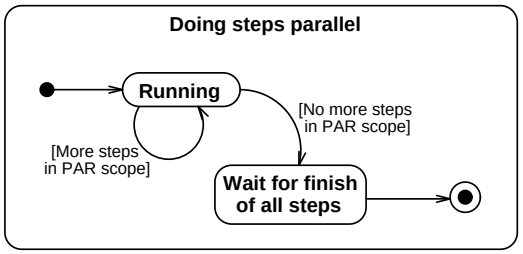

Figure 5.8: The nested state chart of the Doing steps parallel state

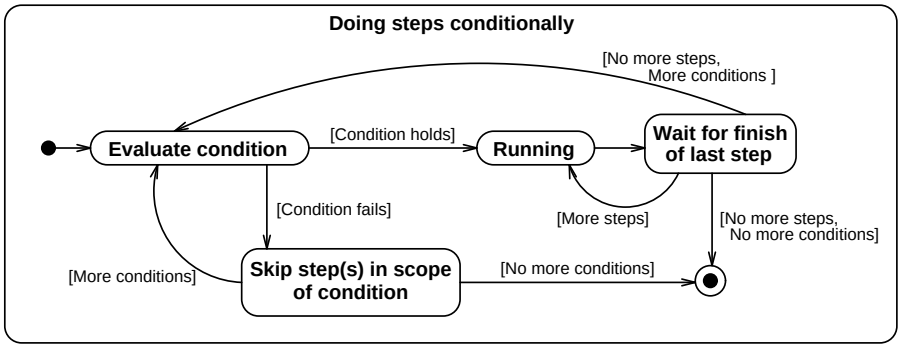

Figure 5.9: The nested state chart of the Doing steps conditionally state

With the IF constructor, a step is only handled when the condition preceeding it, holds. This condition may be a result form an earlier step or may be a signal from the user of the method. In figure 5.9 the state chart of the state Doing steps conditionally is given. Upon entering this state, the method enters the state Evaluate condition and consequently enters the Running state if the condition holds. When multiple steps follow a condition, they are treated sequentially. This may be clear when the right part of figure 5.9 is compared with figure 5.7. When the condition fails, all steps following it are skipped when the method is in the state Skip step(s) in scope of condition. The method consequently evaluates the next condition when there is one, or leaves the state Doing steps conditionally when not.

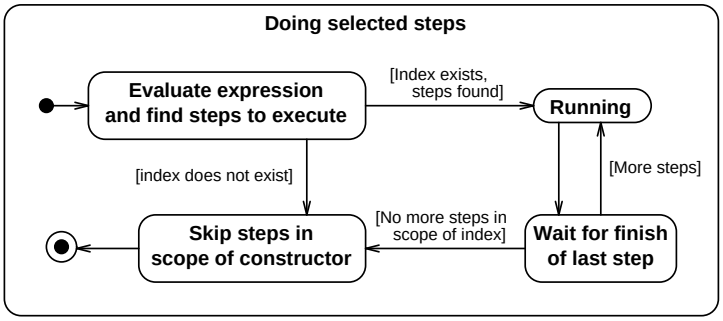

Figure 5.10: The nested state chart of the Doing selected steps state

When the method enters the state Doing selected steps it first enters the state Evaluate expression and find steps to execute. This may be seen in figure 5.10. Intuitively this means

that a choice between a set of series of steps is offered, but only one serie will be executed. The execution depends on some expression which may involve the result from an earlier step, or human intervention. When, as a result of evaluation of the expression, a series of steps is executed, they are executed sequentially.

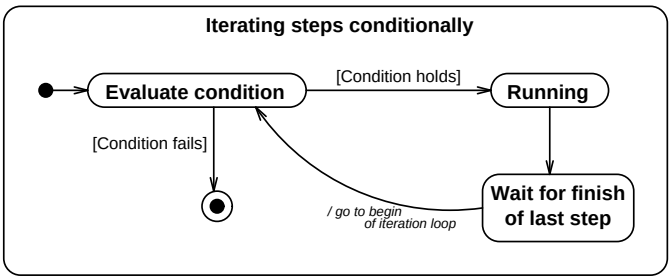

Figure 5.11: The nested state chart of the Iterating steps conditionally state

When parts of a method may be repeatedly executed, this may be done by a conditional iteration or a fixed iteration. The state chart of the former case is given in figure 5.11. The method enters the state Iterating steps conditionally. In this state, the condition is first evaluated to see if the following steps must be executed at all. When the condition holds, the steps following are executed. The method enters the state Running to do so and handles the steps sequentially. When finished with the steps, the method reenters the Evaluate condition state. This sequence continued until the condition fails and the method leaves the state Iterating steps conditionally.

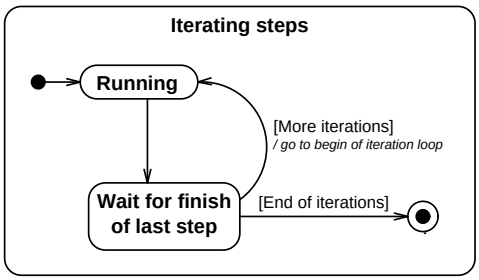

Figure 5.12: The nested state chart of the Iterating steps state

In figure 5.12 the state chart of the state Iterating steps is given. The difference with the state Iterating steps conditionally, described above, is that when in this state, the method iterates the series of steps following the FOR-constructor a fixed number of times. When the iteration is performed the set number of times, the method leaves the state. The number of times the iteration should take place may be predefined in the method itself. It may also be the result of some previous part of the method.

The Document class

In the object-oriented view on life cycles with the method as central object, the role of the document as presented in section 4.3 changes. The document becomes a more passive object, which is steered by the objects Method and Person. This change of its role is reflected in the states it may be in.

The actions that are performed by an object are limited to guarding its own structure, and library functions. Normally, a document would simply exist. It is instantiated by a signal from a developer or by a signal from a document that it has an aggregate relationship with. In its Exists state, its contents may be altered by a method.

Figure 5.13 shows the dynamical model of the Document class.

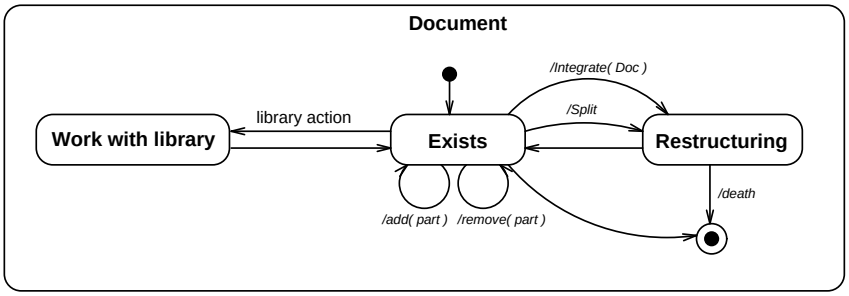

Figure 5.13: The dynamical model of the Document class

Whenever a document gets the signal to split up or to integrate two documents, it would go into a Restructuring state. When in this state, the method may give birth to new documents or may kill other documents. It may also create, duplicate or remove links. The event that causes the document to go into this state usually comes from a method when the basic development step Split_Up is executed. This method step is responsible to split up the contents of the document according to the integration rules of the method it is part of. The actual change of the structure of the document is done by the document itself.

To stimulate reuse of documents, which is a basic software engineering principle, a library of documents may be built. The exact structure and usage of such a library might be out of the scope of this thesis but some remarks should be made. Building a library of documents means that at a given point in time, decided by a person, a copy of a document is made and stored for future use. The original document is further used in the development of the final product. For instance, the specification of some communication protocol may be used in future projects. Therefore when the protocol is defined, a copy of it will be put in a library while the original document is further implemented.

5.3 Management Methods

In this section the theory of method-modelling is taken one step further. With the construction of the development method model it became clear that there is another influence on the development process which is not captured in my model: the management of the method.

When we think about development methods and the way that the software developer uses them, we can project this idea to other fields. For instance, how does a project manager manage a software development project. It is likely that it is possible to build a method core of basic management steps which may be used to formalise and model software management.

With management methods, the development process is regulated. The management method may act on development methods. In other words, the development methods are also the "documents" for the management methods. The following aspect of the development methods may be modelled by introducing management methods.

With the model out of the previous section, the structure and behaviour of a method object is captured. But there is one more state chart that needs to be added to the model. In the state charts for the states Doing steps conditionally and Iterating steps conditionally, we find a state Evaluate condition. In this state, the "Creative" development step is being taken. The result of this step is a signal for the method. The result of the "Creative" is one of three possibilities. The "continue" and "stop" signals as result of the "Creative" step are modelled in the state chart in figure 5.14.

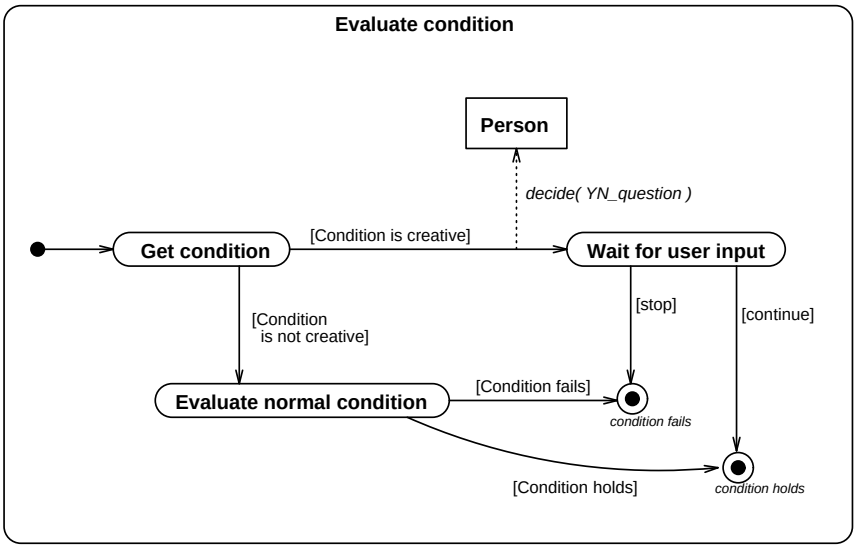

Figure 5.14: The state chart for the Evaluate condition state

However, when the developer gives a "break" signal, the method must retrace its steps and go back to the last position where the developer has given a "continue" or "stop" signal. This retracing of steps is a method-specific implementation and a general algorithm cannot be given. Also, when a method retraces its steps, the documents that the method acts on must be restored to the states they possessed at the time the "continue" or "stop" signal was given. For this purpose, the document object should store the changes to its contents and to its structure (including its *depends_on* links) in an archive each time either of these signals is given. This way the documents may always be reconstructed. The actual implementation of this archiving may be captured in the document, but how is it steered? We could see this retracing of steps and archiving of documents as *basic method management steps*.

I do not know which basic management steps form the method core for management methods, but I suspect that aspects like document library management, management of man-power, planning, budget management, and so on, would all find a place among them.

6 Summary and Conclusions

In the first part of this thesis, I have given an introduction on software life cycles and the function of methods therein. In this part I stated that using multiple methods in a (iterative) development process requires such methods to be integrated. Two types of method integration were defined: Intra-process method integration and inter-process method integration. Subsequently, the major problems that arise when integrating methods were identified.

To be able to build an object-oriented model of methods and their role in the development process I have given a short introduction on the philosophy of object-orientation. Following this I have summarised the notation of the "Object Modeling Technique" as developed by Rumbaugh et al. [RBP+91]. This method was later used to describe the actual model.

Before building the model, an in depth look was taken on the structure of a method. Using the work of P. Löhr-Richter, PhD. on generic methods [LR93] I have described how development methods may be seen as structures of a relative small set of basic development steps. A generic form of each of these development steps was given. Using constructors and the generic forms of the development steps a small example was given of a generic form of a method. Having built an understanding of the structure of a method, I took a look at an existing object-oriented life cycle model which has a document object as the central object; documents being the product of a development process. The conclusion of this analysis was that the structure of the generic methods would require a new definition of a document object and its place in the model.

Finally, in chapter 5 an object-oriented model for methods has been built. This model may be extended to an object-oriented life cycle model. The theory on generic methods was used to implement a model of a method that can be engineered. In the following section I conclude this thesis by naming some of the consequences that this model may have for software development.

- **Method integration**

  I have briefly discussed method integration in section 2.2. There it was concluded that method integration requires some kind of method engineering.

  One of the most significant aspects of the model from chapter 5 is the aggregate structure of methods. By combining existing method classes, new method classes may be built. Thus, when multiple methods are used in a development process, a new method class may be engineered that has all the methods used as parts.

  The formalised form of the methods simplifies both types of method engineering. With inter-process method engineering, the method engineer designs the structure of the composite method, thereby defining the order of the methods parts and adding basic development steps, or, for example, translation methods to fill the gaps between the methods. Methods with overlapping functionality may be (partly) redesigned. With intra-process method engineering the method engineer may redesign two existing methods into one new method, thereby losing the overlap and keeping the functionality of both methods. Once a complete and working method for the whole

software development process is built, it may be instantiated again and again without major re-engineering.

The major problem here is that, currently, there are no generic forms available for the methods used today.

- **Project management**

  The structured, object-oriented approach to the development process clarifies the relationships that the objects in the process have with each other. It should not be too hard to add new objects and determine their role in the process. This way the development process may be better understood and therefore easier to control. In section 5.3, I have already suggested that management methods might fit well into the model of chapter 5.

- **Reusability**

  I have already spoken of library building. The fact that the document object and its place in the development process are now well defined opens the way to library building. At very distinct points in a method, a document may be placed in a document library for reuse purposes.

A OMT Model

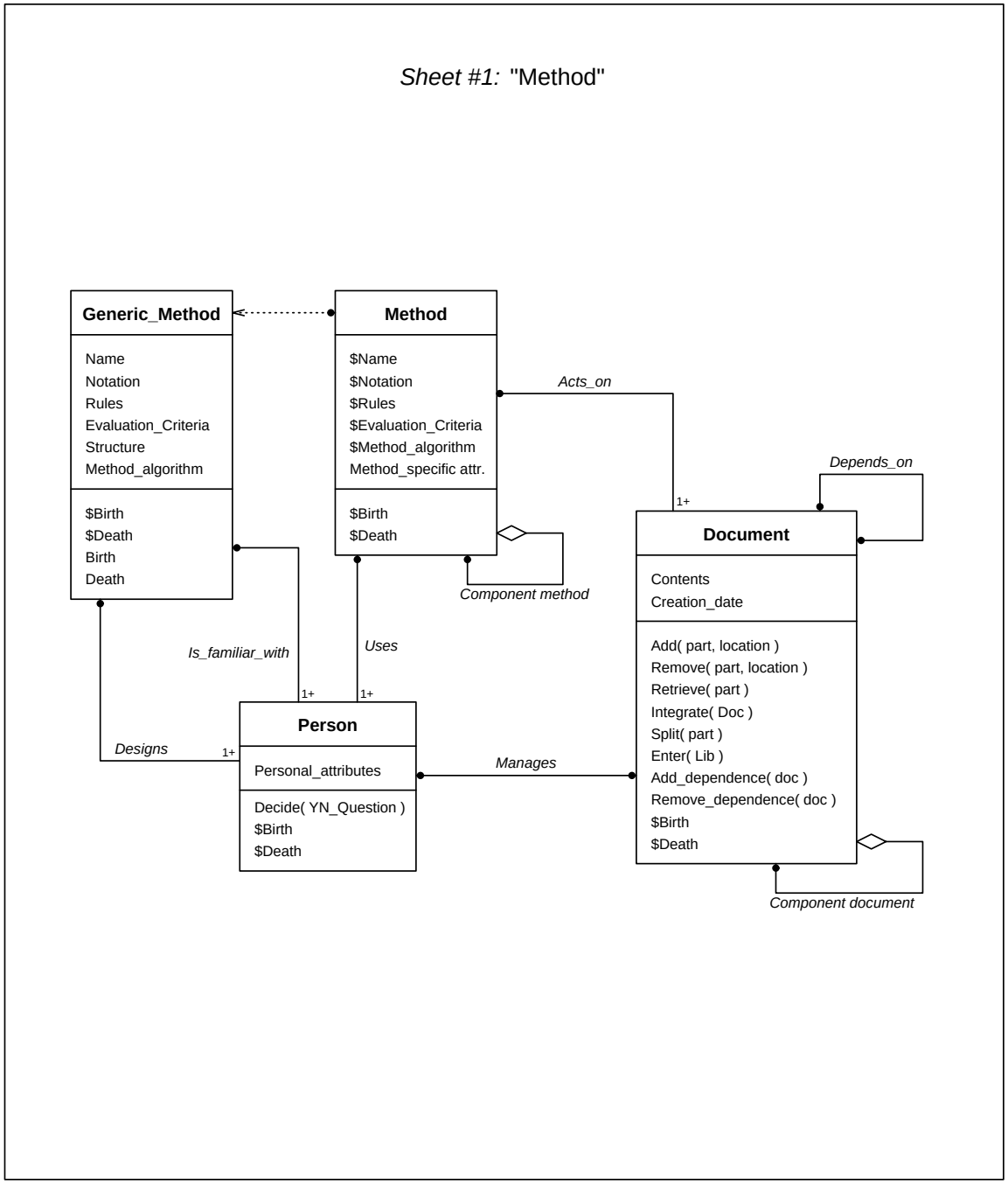

Figure A.1: The object model for the Method object

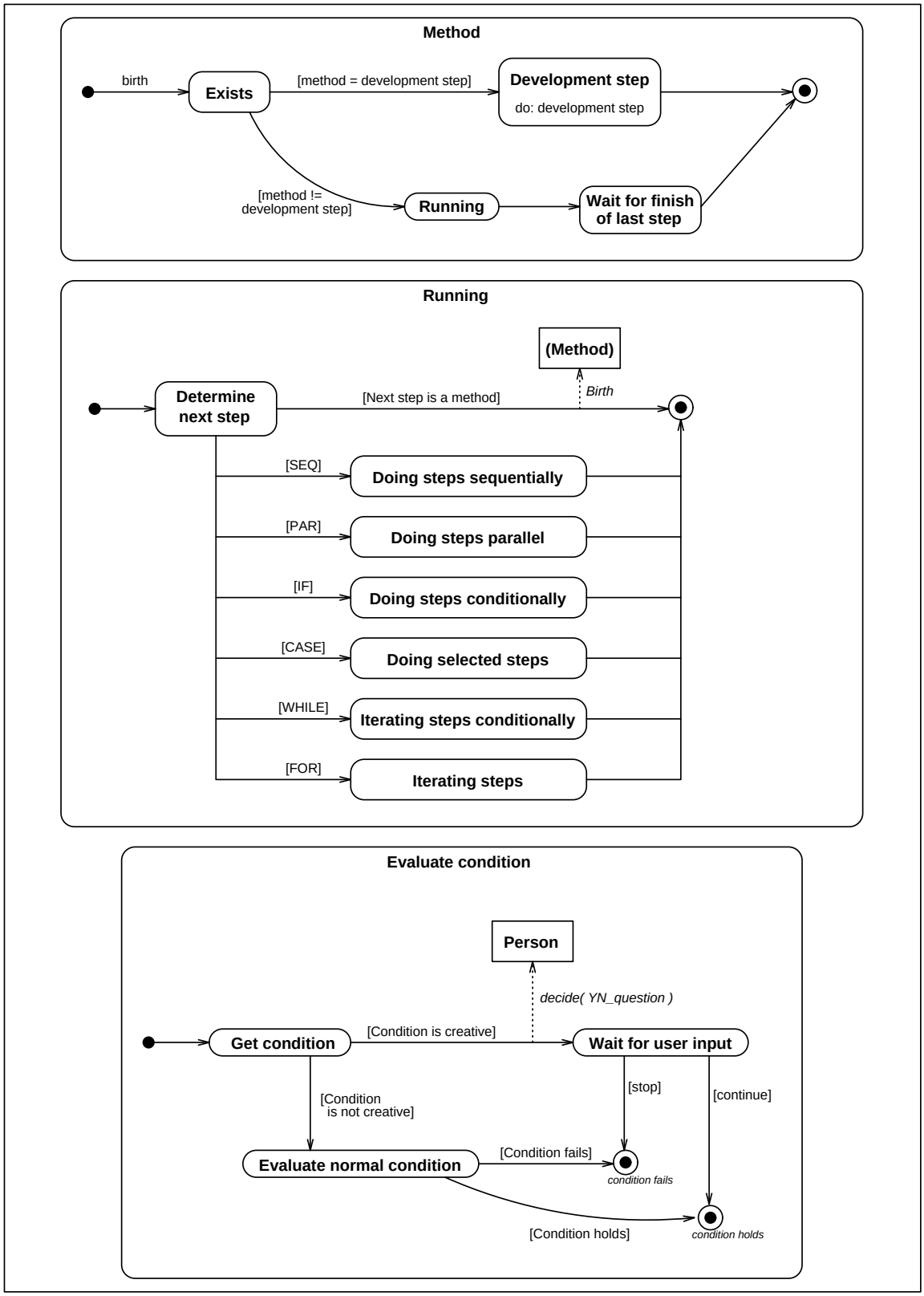

Figure A.2: The dynamical model for the Method object - part 1/2

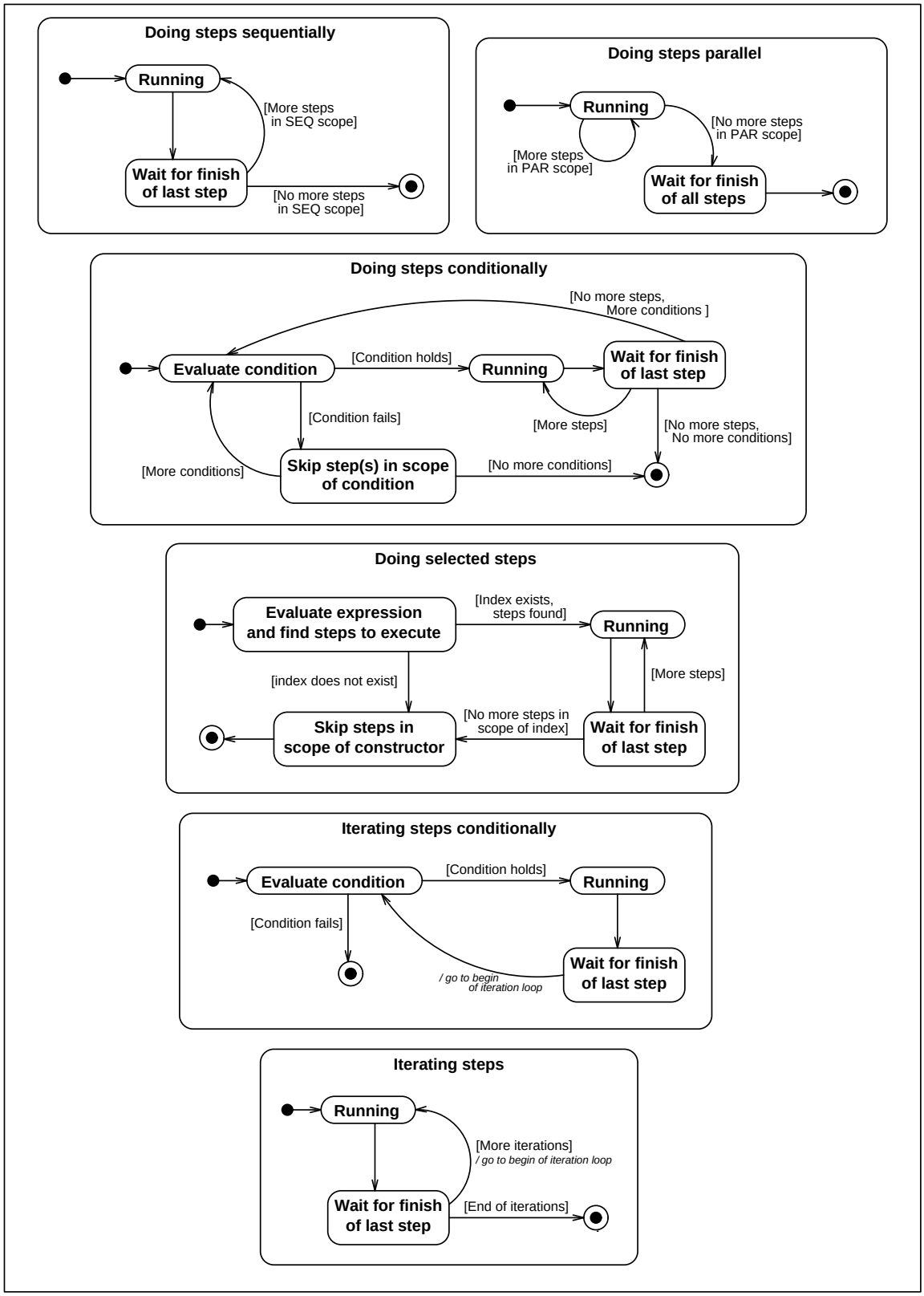

Figure A.3: The dynamical model for the Method object - part 2/2

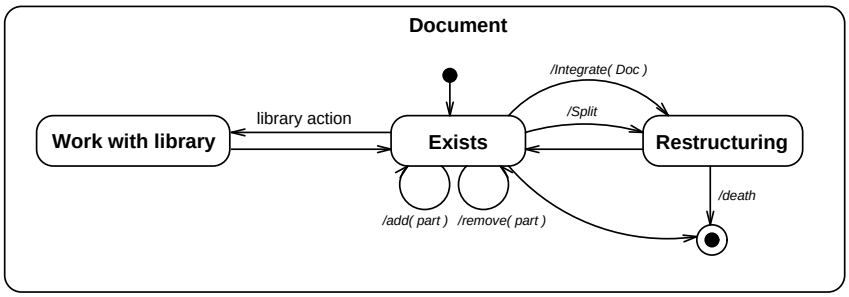

Figure A.4: The dynamical model for the Document object - part 1/1

B The Assignment

Description of the doctorate research project (draft title) "**Integrating Method Objects into the Object Oriented Life Cycle Approach**"

Life cycle models represent a very abstract class of software process models. They usually determine the basic set of development phases to be achieved by a software product and the basic execution order defined on such phases to constitute the software development process. The most well-known life cycle models include the waterfall model, the prototyping model, and the spiral model.

These traditional approaches evolved over time but still have difficulties in explicitly capturing aspects like

- iterative development styles,
- distributed and cooperative development, and
- to integrate reuse.

The object oriented life cycle approach defined at the Department of Computer Science, Technical University of Braunschweig (Germany)[LoRe93], focuses on explicitly incorporating the above aspects. For this purpose, it uses object oriented concepts, languages, and facilities for two purposes:

1. It is assumed that the underlying development process is carried out with object oriented languages. This emphasizes the need to integrate reuse.
2. To specify the life cycle model, the object oriented specification languages OMT [Rumbaugh et al. 91] and OBLOG [Sernadas et al. 90] are used. Hence, the specification serves as application of object oriented languages to software process modeling.

In contrast to the traditional life cycle models, which assume a set of global phases for the entire software product, the object oriented approach models the software development life cycle in terms of a society of interacting software documents. Together, this society yields the entire software product but this society is assumed to be developed in a distributed environment by teams of developers supported by reuse. Consequently, the current specification of the approach is centered around the specification of the software documents. They are modeled as objects independently evolving over time, i.e., behaving individually, and sending messages to each other to establish a joint development process. Documents are evolved by developers who apply development acitivities/methods to them. Presently, such activities are specified as operations associated to the document objects. Due to the abstraction level of a life cycle model, only few operations are presently attached to the document objects. To incorporate all development activities/methods available in software development, a huge collection of operations has to be defined for document objects. In order to avoid this, it suits better to introduce a separate object class for development activities/methods into the approach and to extract and define the application of methods on top of the communication between documents, methods and developers.

This task will basically constitute the subject of your doctorate research project named "Integrating Method Objects into the Object Oriented Life Cycle Approach" (draft title).

During the six months that you will work at the University of Braunschweig, I will investigate development activities, especially methods, of different development phases, classify them, and define an object class for them. Firstly, this means to identify

- what are the characteristics of methods,
- how methods can be described by an object class,
- how method objects should interact with document objects, and
- what are the problems basic to method integration.

Secondly, it means to refine the existing (OMT) specification of the document objects, adapting their interfaces and the relationships between them with respect to the newly introduced method objects. Thirdly, it is necessary to introduce objects for developers and guarding processes which apply methods or control their application and to integrate these objects into the approach. In turn, this will lead to a refined life cycle model which can conveniently be extended to a full software process model.

References

[LoRe93] Löhr-Richter, P.; Reichwein, G.: Object Oriented Life Cycle Models Technical Report 93-05, Dept. of Computer Science, Technical University of Braunschweig, 1993

[Rumbaugh et al. 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design Prentice Hall, 1991

[Sernadas et al. 90] Sernadas, A., Sernadas, C., Gouveia, P., Resende, P., Gouveia, J.: OBLOG - OBject-oriented LOGic - An Informal Introduction INESC Research Report, Lisbon, 1990

References

[Bal90] H. Balzert, editor. *CASE - Systeme und Werkzeuge.* BI-Wissenschaftverlag, 1990.

[CY91a] P. Coad and E Yourdon. *Object-Oriented Analysis.* Englewood Cliffs, 1991.

[CY91b] P. Coad and E Yourdon. *Object-Oriented Design.* Englewood Cliffs, 1991.

[DE93] P. Demmenie and T. van Elzakker. *LOTOS - Een korte inleiding.* A study for the Computer Supported and Cooperative Work Seminar, RijksUniversiteit Leiden, Dec 1993. *Unpublished.*

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[Har87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems.* In Science of Computer Programming, volume 8, pages 231–274, 1987.

[ISO88] ISO IS 8807, Information Processing Systems, Open System Interconnection. *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior.*, 1988.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. *Object-Oriented Specification of Information Systems: The TROLL Language.* Technical Report 91-04, Technical University of Braunschweig, 1991.

[Kro93] K. Kronlöf, editor. *Method Integration - Concepts and Case Studies.* John Wiley & Sons, Chichester, 1993.

[LR93] P. Löhr-Richter. *Generische Methoden für die frühen Entwurfsphasen von Informationssystemen.* PhD thesis, Technische Universität Braunschweig, 1993.

[LRR93] P. Löhr-Richter and G. Reichwein. *Object Oriented Life Cycle Models.* Technical Report 93-05, Technical University of Braunschweig, 1993.

[LRS94] P. Löhr-Richter and G. Scherrer. *Software Engineering - Methoden und Vorgehensmodelle.* EMISA-Forum, Mitteilungen der GI-Fachgruppe, 2.5.2, 1994. *To appear.*

[MM88] D. Marca and C. McGowan. *SADT: Structured Analysis and Design Technique*, 1988.

[OHM+91] T. Olle, J. Hagelstein, I. MacDonald, C. Roland, H. Sol, F. Van Assche, and A. Verrijn-Stuart. *Information Systems Methodologies - A Framework for Understanding.* Addison-Wesley, second edition, 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Ros77] D.T. Ross. *Structured Analysis (SA): A Language for Communicating Ideas.* In IEEE Transactions on Software Engineering (SE-3), pages 16–34, 1977.

[Ste91] W Stevens. Software Design: Concepts and Methods. *Practical Software Engineering Series*, A. Macro, editor. Prentice-Hall, 1991.

[Wie91] R. Wieringa. Steps towards a method for the formal modeling of dynamic objects. In *Data & Knowledge Engineering*, volume 6, pages 509–540, 1991.