A Comparison of two outstanding Methodologies for Object-Oriented Design

Ulrich Frank

FIT.CSCW

GMD Sankt Augustin

Abstract

Although the promises of object-oriented software development are based on solid grounds there is still a great amount of confusion on when and how to invest in this new technology. This is partially due to the fact that a great number of special approaches and methodologies have been proposed during the last years. The report gives an overview of more than 30 approaches and presents a framework that helps to compare them. The framework is then applied to two methodologies which gained remarkable attention so far and are well documented at the same time: "Object Modelling Technique" by Rumbaugh et al. and the methodology proposed by Booch. Both methodologies are described in detail and evaluated according to the criterions presented in the framework. Since design methodologies can hardly be evaluated without regarding specific context variables the report does not aim at an "objective" score for each approach. Instead it is intended to ease the decision between the two methodologies by pointing to important differences.

Keywords

Object-oriented analysis, object-oriented design, object-oriented modelling, object model, dynamic model, functional model, design methodology

1. Introduction

It is widely accepted that the development of computer-supported information systems requires the design of a suitable conceptual model. A conceptual model aims at documenting the essential properties and constraints of an information system in an illustrative way - last but not least by abstracting from implementation issues. Within the current software-engineering practice conceptual modelling is primarily done by data modelling where the Entity Relationship Model is of special importance. During the last years object-oriented software development has become more and more popular. While object-oriented programming languages are still gaining attention an increasing number of software engineers gets aware of the fact that the development of object-oriented information systems requires special methodologies for conceptual modelling. An object-oriented approach promises to be very well suited to accomplish the goals commonly associated with conceptual modelling: Usually objects offer a more direct and natural correspondence to real world entities than data structures. Inheritance and encapsulation promote reusability of concepts and components. Furthermore an object-oriented model fosters integration: The various parts of a corporate wide information system can communicate on a higher level of semantics by referring to domain level classes instead of (structured) data types.

Although the promises of object-oriented software development are based on solid grounds there is still a great amount of confusion on when and how to invest in this new technology. One reason for this confusion is closely related to the attractiveness of object-oriented software: It motivates many manufacturers and vendors to stick the label "object-oriented" to their products without delivering substantial features - as King (1989, p. 24) puts it ironically: "If I were trying to sell (my cat) ... I would argue that he is object-oriented." The other reason is the great variety of research going on. On the one hand this variety results from the three roots of object-orientation: Artificial Intelligence, programming languages and data modelling. On the other hand it has epistemological reasons: Whenever there is a new attractive research area a lot of ambitious people feel motivated to work in this field - and to create their own approach. In a comment on the news-net forum "comp.obj" (October 1991) Berard describes this situation in the following way:

> "I have good news and bad news. The good news is that there has been a great deal of work in the area of 'object-oriented software engineering'. The bad news is that there has been a great deal of work in the area of 'object-oriented software engineering'".

For an executive or a system engineer who is in charge of deciding about a methodology to apply this situation is often even worse: Many companies only recently merged to Relational Database Management Systems and the Entity Relationship Model as the corresponding modelling methodology. In order to

protect this investment it makes sense to hesitate with merging to object-oriented technologies.

The purpose of this paper is not to argue for an object-oriented approach - the time that is most appropriate for turning to a new technology depends on a complex set of circumstances which have to be carefully considered in every single case. Instead we will try to ease the decision between different methodologies for object-oriented design. Although there is already a large number of methodologies available we will not consider a complete set - which is changing from day to day anyway. In practice the number of relevant methodologies is much smaller. We will focus on two methodologies we found to be of outstanding importance - which does not necessarily mean that they are of outstanding quality. They are well known, well documented, supported by a number of tools, and relatively comprehensive.

First we will introduce a framework that supports the evaluation of methodologies for conceptual modelling. We will then give an overview of object-oriented design methodologies and finally we will compare the methodologies proposed by Booch and Rumbaugh et al.

2. A Framework for Evaluating Modelling Methodologies

Evaluating methodologies is usually a delicate job. This is specially true for complex engineering methodologies. The judgement of those who apply them is very likely to be biased: They have internalized the conceptualizations they professionally use and only for this reason they tend to prefer them from others. The different opinions that emerge in this context make is almost impossible for the proponents of different methodologies to agree on common criterions. An infamous example for such a situation is the sometimes passionate discussion on the quality of programming languages. It does not merely characterized by a lack of objectivity, instead it necessarily reflects the fact, that the evaluation of modelling techniques cannot be accomplished regardless of personal preferences and predispositions of the users. A methodology's quality heavily depends on its acceptance or in other words: how it contributes to a productive workstyle. The criterions described below however do not include such subjective attitudes, since they vary in a wide range. The criterions are deducted from the goals associated with conceptual modelling, mainly integration and reusability.

2.1 Important Aspects

General Criterions

- Clarity: A methodology's clarity or descriptiveness depends on the quality of the (graphical) presentations it provides. This is the case for both the basic

modelling constructs and the rules for composing them. It is desirable that the constructs as well as the models composed from them should "correspond directly and naturally to our own conceptualizations ..." (Mylopoulos/Levesque 1984, p. 11). The presentation of a model should contribute to show this correspondence. Since design, implementation, and use of corporate information systems are associated with a range of different perspectives (like those of system analysts, programmers, executives etc.) it is desirable that the methodology provides various levels of abstraction.

- Views: While integration recommends a holistic look at a system, it is nevertheless important that different contexts or views can be distinguished within a model. A view not only fosters a model's clarity by allowing to abstract from aspects which are less relevant, it is also a prerequisite for defining access rights and modelling aspects of the user-interface.
- Level of semantics: In order to establish a close correspondence to certain parts of an application domain it is important that the modelling constructs allow for capturing a high level of semantics. This is also required for the associations between the constructs. Furthermore a methodology should provide powerful concepts to generalize over invariant feature of a domain.
- Flexibility: Integrating the various parts of an information system requires a common semantic reference system. A modelling approach should allow to establish such a system for all parts of an information system - for instance for office documents as well as for accounting. Furthermore it should be suited to capture all essential aspects of design. That is not only static aspects but also dynamic or functional aspects as well as the user-interface.
- Formalization: The modelling constructs should be defined well enough to instruct or even generate implementation. The number of such constructs should be small, the composition rules should be straightforward.

Integrating the development phases

- Common references: In order to integrate the different phases within the software life-cycle it is recommended to use identical or at least similar constructs from early analysis to implementation. Since the those phases require different levels of abstraction it should be possible to look at the constructs at various levels of detail.
- Incremental formalization/specification: Sometimes it is not possible or desirable to specify a construct in early phases in a detailed way. Therefore the methodology should allow for specifying constructs in a preliminary way - not forcing the designer to maintain integrity all the time.
- Avoiding friction between phases: Modelling complex application domains is an evolutionary process. Therefore a methodology should allow for conveniently modifying a model. Transforming from one level of abstraction to

another (i.e. from analysis to design) should not imply the loss of semantics. Otherwise such transformations are not reversible and reconstructing semantics at later point in time are expensive and risky. A methodology should also support to maintain referential integrity between interrelated constructs used in different phases. Furthermore it should allow to define formal procedures to detect design errors not only on a syntactic but also on a semantic level.

Reusability

- Modularization: The components corresponding with the modelling constructs should be protected against side-effects.
- Domain specific frameworks: A methodology that provides generic domain models not only contributes to the design of specific models but also fosters the search for application level components.
- Runtime-support: Implementing information systems requires a number of general support functions, like interprocess-communication, management of data and functions, information-retrieval etc. It is desirable that a methodology is compatible with powerful runtime systems (like operating systems and DBMS) which provide those functions.

Promoting sophisticated architectures

- Distribution: Modelling constructs and the corresponding implementation components should allow for transparent distributed operations. In principal the chances for a transparent distribution are improving with an increasing amount of semantics contained in the constructs which are subject of communication. Mainly however the quality of a distributed architecture depends on the integration of open system components, in other words: on the level of standardization or commitment associated with modelling constructs or components.
- Integrity: The higher the level of semantics a model allows for the better are the chances to accomplish an architecture that promises a high degree of integrity.

Looking at the various criterions listed above reveals that some of them are overlapping, others hardly allow for a degree of formalization that is required for a comparison. Furthermore there is a conflict between some of the criterions. For this reason we will further refine and restructure the criterions.

2.2 Refining and Enhancing the Framework

A conceptual model is an abstraction of a real world domain: Only those aspects of a domain are considered, which are of any relevance for the information system that is to be designed. Furthermore single instances of real world objects are neglected. Instead it is desirable to apply generalized concepts. Within the criteri-

ons listed above a model's level of semantics is of outstanding importance. The purpose of an information system is to manage information in a reliable way. The degree of a model's semantics depends on the permissible interpretations of the stored information: the more they are restricted the more semantics the model contains. In other words: semantics directly corresponds with the constraints which exclude non permissible system states. To become more specific about modelling we will first consider only static aspects. A conceptual model may then be characterized as being composed of a set of entity types and associations between them or the instances represented by them respectively.

An entity type or object type represents a set of entities, created by abstraction of real world objects. A model's integrity can then be characterized by an isolated definition of its entity types as well as by considering associations between different types.

Isolated view of entity types

- Constraints on permissible states of an entity type's instances.
- Constraints on permissible state transitions of an entity type's instances. Such constraints are important, whenever an arbitrary transition from one permissible state to another is not always allowed. For instance: The permissible values for a retail price may cover a certain range. Additionally however it may be appropriate to further restrict it by the value of the corresponding wholesale price.

Instances of one entity types in interaction with other instances (of the same or of other entity types)

- Constraints on permissible states of an entity type's state depending on the states of other instances. To give an example: The value range for an employee's salary may be restricted by the current value for his manager's salary.
- Constraints on permissible state transitions of an entity type's instances depending on state transitions of other instances. Sometimes the change of an instance's state requires a corresponding change of other entities in order to result in a consistent system state. This is the case for modelling transactions.
- Definitions of events which cause state transitions.

The constraints outlined above will be differentiated in the following way: *Static* aspects describe permissible structures or states, *functional* aspects are focused on functions which are to be performed by a system, while *dynamic* aspects capture a system's behaviour during its lifetime. The functional as well as the dynamic level control structures allow for representing control structures. They specify conditions under which certain functions may be executed and under which certain state transitions will occur. The following framework is based on

this differentiation and reflects the criterions discussed above. It is not restricted to methodologies for object-oriented design. Instead it may be applied to data modelling methodologies as well. For this reason we will avoid object-oriented terminology whenever possible and use more general terms instead - like entity vs. object.

Static Aspects			Description
Attributes	hardware-oriented	definable as value range	within a given, hardware-oriented set (for instance: fix point numbers from 1 to 100)
		by functional specification	again related to a given set (i.e. even fix point numbers)
	application-oriented	by enumeration	i.e. colours, names
		by entity type	An attribute's semantics may be specified by a costumized entity type.
	Cardinality	minimum = 0	Assigning a value may be declared as optional.
		maximum > 1	An attribute may have assigned more than one values.
	access privilege		There are different levels to distinguish access rights to an attribute.
	associations		Here we think of constraints which restrict the permissible values of an attribute in regard of the values of other attributes.
an entity's identity is determined by its state			An entity is solely identified by its state. That implies the constraint that at no point in time two entities may coexist. This is contrary to the case where an entity has a stable identity during its lifetime, regardless of its state.

Static Aspects				Description
History	general			It can be specified within a model if changes of an entities state should be recorded.
	partial			Recording of changes can be restricted to values of attributes.
Associations between entities	Cardinality	single		Cardinality is specified with one value (like 0, 1, *) for each of the associated entities.
		min-max	restricted	It can be assigned a minimum and maximum cardinality each - out of a given set of symbols (like for instance {0, 1} or {1, n }).
			free	Minimum and maximum cardinality can be specified using a wide range of values.
	Direction	role		In order to distinguish between associated entities it is possible to assign roles.
		inverse association		An association is directed. The meaning corresponding with each direction can be expressed by a domain level name and an inverse name as well.
	referential integrity			The existence of an entity can be defined to be dependend on the existence of other entities (in addition to cardinalities).
	aggregation			There is a constructor to characterize an association as an aggregation.
	miscellaneous			Further types of associations with special meaning (like "is version of") are provided.
Generalisation	single			Generalisation is possible using one common pattern. This allows for single inheritance.
	multiple			The methodology allows to generalize over multiple patterns. In other words: multiple inheritance can be expressed.

Static Aspects

		Description
Views		
Entity	specification of views	Views on an entity can be specified by selecting a set of attributes or operations.
	calibration of access privileges	Dito, accompanied however by assigning access rights to each view.
	user-interface	A prototypical user-interface may be assigned to each view.
Collection	specifikation of views	A view that is relevant for a certain working context can be composed out of different views on entities.
	user-interface	A prototypical user-interface may be assigned to such a working context.

Dynamic/functional Aspects

Operations/Functions		
specification of interfaces		The interface of an operation can be specified by refering to costumized entity types.
preconditions		A condition can be specified which is required to be satisfied in order to execute the operation.
postconditions		A condition can be specified that is required to be fulfilled when the operation terminates.
triggers		Tupels of events and corresponding actions can be defined within a model.
exception handling		The methodology supports the modelling of a unified exception handling.
system behavior		
Control structures	for entities	Constraints can be defined which restrict the permissible state transitions for entities of a certain type.
	for collections	Such constraints can be specified for a collection of interacting entities of different types.

Miscellaneous Aspects		Description
Modularization	coherence	Static and functional/dynamic aspects of an entity type are modelled together at one place.
	referencing	References can be established between corresponding parts of the static and the dynamic model. That requires that those parts can be uniquely identified.
Buildtime- and runtime-support		
DBMS	DDL	The model allows for an automated or at least partially automated transformation into the Data Definition Language of a DBMS.
	DML	The methodology promotes a Data Manipulation Language to allow for retrieval and manipulations on the instance level.
	standards	DDL and DML are widely accepted or standardized.
	controlling system behavior	Dynamic constraints defined in the model can be controlled by a DBMS.
	tool support	A set of mature tools to support the methodology is available.

Before we apply the framework to the two methodologies which we have selected, we will first have a look at the terminology of object-oriented software development in general. It is not intended to present formal definitions. Instead we will concentrate on a short description of the central notions of object-orientation. Such a terminological reflection is required since the increasing popularity of object-oriented technology goes along with a remarkable amount of confusion about the terminology and the future potential of this technology. Rentsch (1982, p. 51) characterized this situation for the 80s - where on my opinion one could also increment the years by 10:

> "My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."

3. Terminology

High-level programming or specification languages aim at providing the developer with means to write a program/specification in correspondence to the language used in the real world domain of interest. Although early programming languages allow to abstract from the characteristics of a specific processor, they are still hardware-oriented since the data types they provide are usually determined by the available internal representations. The operations which can be applied to instances of such data types are usually hardware-oriented as well. The introduction of abstract data types was motivated by the goal to provide constructs that reflect domain level semantics rather than hardware features. Abstract data types are created by defining a data structure together with a set of operations which can be applied to it. One essential characteristic of abstract data types is their contribution to modularization - and thereby to reusability. A compiler treats data and the operations associated with them as one unit. Furthermore it is possible to define that data may only be manipulated by operations provided by the abstract data type they are instantiated from. This allows for an effective protection against side effects.

The concept of a class is very much inspired by abstract data types: A class is also defined by a data structure and a set of operations applicable to them. Additionally to abstract data types classes can inherit features (data structures or operations) from a superclass. If a class can only inherit features of one superclass, we speak of single inheritance, otherwise we call it multiple inheritance. We consider an object to be an instance of a class. It is created by instantiating it from its class which means that the memory required to store its data is allocated and references to the operations defined in the class are established. Data protection is of crucial importance: data of an object are transparent to the external world, they are *encapsulated* within the object. Access to data is only possible via *services*. A service is defined by the interface of the operations an object offers to the external world. The set of services an object provides is sometimes referred to as protocol. Sometimes it is helpful to generalize over a set of classes although the resulting superclass does not represent any real world object. In this case we speak of an *abstract class*. An abstract class is a class that does not allow for instantiating objects from it.

Beside encapsulation and inheritance the notion of an object or a class we prefer includes one further essential feature: *polymorphism*. It simply means that operations defined in different classes, representing different semantics, may be identified by services with the same name. Polymorphism reflects the fact that communication between objects requires both the identification of an object and of one of its services. Selecting a service from an object is accomplished by sending it a *message* that includes the service identifier and - if needed - a list of parameters.

Sometimes "object identity" mentioned as one of the essential characteristics of object-oriented systems. It stresses the fact, that an object has an identity that does not depend on its state. Therefore this feature is primarily of importance for distinguishing Object-Oriented DBMS from Relational DBMS.

Beside those software engineering features it is of crucial importance for our point of view that a class or an object respectively allows for highly descriptive models of application domains. The underlying idea suggests that objects directly correspond with objects of a real world domain. Abstracting from a single occurrence of an object by introducing classes corresponds with common strategies for building concepts. This is also the case for generalisation over common features (introduction of superclasses) and specialisation from a given class (introduction of subclasses). Polymorphism reflects the common practice that similar operations, applied to different objects, have the same name.

In order to be independent from the specifics of a certain object-oriented programming language and to offer a more descriptive presentation at the same time, it is desirable for the development of large information systems to first design an object-oriented conceptual model. On the one hand it consists of the specification of classes which may include static as well as functional or dynamic aspects. On the other hand it serves to describe the associations between classes and between objects respectively.

4. Overview of Object-Oriented Methodologies

Analysing the literature reveals several methodologies for object-oriented modelling. The majority of those approaches however - usually published as research reports or conference papers - is only in a preliminary state. Only a few methodologies are documented in a comprehensive way within textbooks or manuals. Different to data modelling none of them can claim to dominate the scene. The following overview has been composed at the end of 1993. It aims at being complete but does not claim to be. It is only to give an overview and the corresponding references not to provide a comparison. The methodologies' description would not always allow for a detailed comparison. Monarchi/Puhr (1992) offer a structured, however not very detailed comparison of 23 methodologies. A more comprehensive, but still somewhat superficial comparison of four selected approaches is given by Hsieh (1992). Hong and Goor (1993) compare six methodologies by introducing a "supermethodology" on a meta level. Thereby they offer a neat comparison, which is however restricted to only a few criterions, which are not always selected in an appropriate way (sometimes they refer to features of tools, not of methodologies). Other studies which aim at a comparative evaluation are those conducted by De Champeaux and Faure (1992), Hewlett-Packard (1991) and Mannino (1987).

The number and variance of object-oriented methodologies indicated the dynamics of the research area. The approaches are sometimes differentiated in those which are to support analysis and those which are primarily thought to be applied for design. We will not apply this distinction since our focus is directed to the conceptualization of object models. Methodologies which support analysis mainly try to help with identifying the concepts that are of relevance for object-oriented modelling.

The methodologies usually include a meta model for object-oriented modelling - by which we mean a model that describes how to conceptualize object models. In this respect they are similar to meta models developed to serve as a basis for open architectures - like the one proposed by the OMG (1992). Despite this essential similarity meta models are not included in the overview.

The overview contains two methodologies where the authors (Coad/Yourdon, Shlaer/Mellor) have described their approach within two volumes. In these cases the two volumes are listed together. The year indicates the time when the methodology was published, not necessarily when it was first introduced. Only a few methodologies have been given a name by their authors.

Author	*Name*	*Year*	*Type of Publication*
Alabisco		1988	C
Ackroyd/Daum		1991	A
Berard		1986	M
Bailin		1989	A
Booch		1990	T
Buhr		1984	T
Cherry	PAMELA 2	1987	M
Coad/Yourdon	OOA/OOD	90/91	T
Cunningham/Beck		1986	C
Desfray		1990	C
Edwards	Ptech	1989	A
Embley et al.		1992	T
ESA	HOOD	1989	M
Felsinger		1987	T
Ferstl/Sinz	SOM	90/91	A
Firesmith		1992	T
Henderson-Sellers/Constantine		1991	A
Jacobson et al.		1992	T
Johnson/Foote		1988	A
Kadie		1986	R

Author	*Name*	*Year*	*Type of Publication*
Kappel/Schrefl		1991	C
Lee/Carver		1991	A
Liskov/Guttag		1986	T
Masiero/Germano		1988	A
McGregor/Sykes		1992	T
Mullin		1989	T
Nielsen		1988	M
Odell		1992 b	A
Page et al.		1989	C
Rajlich/Silva		1987	R
Seidewitz/Stark		1987	A
Robinson		1992	T
Shlaer/Mellor		88/92	T
Rumbaugh et al.	OMT	1991	T
Velho/Carapuca	SOM	1992	C
Wasserman et al.		1990	A
Wirfs-Brock/Wilkerson		1990	T

T Textbook
A Article in a Journal
M Manual
R Research Report
C Conference Paper

Fig. 1: Overview of object-oriented methodologies

While none of the methodologies listed above is extensively used in software engineering practice, a few of them have gained a high degree of popularity. Three methodologies are probably of outstanding prominence: Coad/Yourdon (1990, 1991), Booch (1990), and Rumbaugh et al. (1991). Coad and Yourdon benefit to a large extent from the fame they have gained with their conventional methodologies. Both textbooks aim at giving an easily understood introduction which is enriched by many examples. Furthermore they include a number of heuristics. They primarily focus on static aspects. Such a restricted view may help those professionals who are familiar with traditional data modelling to stepwise enhance their point of view. It does however not help to exploit the full potential of object-oriented software development. Furthermore both textbooks suffer from a sometimes superficial description and a lack of software engineering background. For this reason we do not include Coad and Yourdon's approach in

our comparison.

Different from Coad and Yourdon Booch and Rumbaugh et al. offer a detailed and comprehensive description of their methodologies. Although both approaches clearly have some important features in common they are structured so differently that they can hardly be described together. Therefore we will first look at Rumbas et al. and afterwards at Botch. Both methodologies do not imply a clear distinction between analysis and design. Instead it is regarded as a major advantage of an object-oriented approach that is allows to use the same constructs for conceptualizing a real world domain as well as an information system: “Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation.” (Rumbas et al. 1991, p. 21)

5. Object Modelling Technique

Rumbaugh together with a number of colleagues at a research department of General Electric developed a manual to instruct object-oriented design of information systems. In support of the methodology resulting from these efforts, called "Object Modelling Technique" (OMT), General Electric offers a corresponding CASE-environment. Rumbaugh et al. distinguish between three partial models: a static object model, a functional model, and a dynamic model. All the pages referred to are in Rumbaugh et al. (1991).

5.1 The Object Model

The object model serves as the foundation of OMT. Rumbaugh et al. (p. 6) characterize an object model as follows:

> "The *object model* describes the static structure of the objects in a system and their relationships. The object model contains object diagrams. An *object diagram* is a graph whose nodes are object *classes* and whose arcs are *relationships* among classes."

The graphical representation of an object model is accomplished by drawing object diagrams, which exist in two different occurrences: "class diagrams" and "instance diagrams". Instance diagrams contain specific instances. They merely serve to illustrate a model by pointing to examples. Therefore they are not an essential part of conceptual modelling, because - as Rumbaugh et al. (p. 22) put it themselves: "The notion of abstraction is at the heart of the matter." Class diagrams serve to describe classes. A class is specified by a set of attributes and operations. An attribute in turn is characterized by a data type. For our point of view it is important to note that Rumbaugh et al. do not allow attributes to be objects themselves, that is to be instances of a customized class.

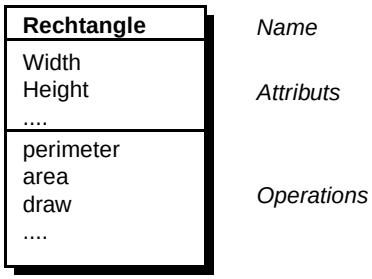

Fig. 2: Class diagram OMT. Depending on the level of detail that is desired additional information can be assigned - like the attributes' data types (see. pp. 23).

Rumbaugh et al. do not mention if attributes can be assigned a cardinality. However it looks like they assume the cardinality of an attribute implicitly to be one. Attributes do not have to be directly represented by data encapsulated within the object. Instead OMT also allows "derived attributes" (p. 26), which are deducted from the values of other attributes. We do not think this terminology is appropriate since it is very likely to confuse about the difference between attributes and operations. Within a class diagram the operations of a class are described by their names and the required parameters. Rumbaugh et al. recommend to distinguish between operations which only perform read-access and those that change data. OMT allows for introducing abstract classes as well as for multiple inheritance.

OMT allows to model associations between classes (pp. 27). For this purpose it would be appropriate to distinguish associations between classes from those between objects. Rumbaugh et al. however avoid such a distinction. Generalisation establishes an association between classes. OMT allows to mark classes as abstract classes. Associations on the instance level can be specified by various features. Cardinality (which is called "multiplicity" by Rumbaugh et al., see p. 30) defines how many instances of one class may be associated with instances of other classes. For this purpose OMT provides three symbols: 1:1, 1:n, and n:m to be used with the graphical representation of associations. Each class (representing a set of instances) within an association can further be assigned a role - similar to the Entity Relationship Model. For special constellations two additional features are available. Objects within an association may be characterized as "ordered". Rumbaugh et al. illustrate this by the association "visible on" between objects of class "Window" and others of class "Screen" (p. 35). Furthermore cardinalities can be restricted by introducing so called "qualifiers" (pp. 35). For instance: While the association between directory and file is usually characterized by a 1:n-cardinality, adding the qualifier "filename" and the cardinality 1:1 allows to express that a filename within a directory uniquely identifies a file. In order to provide illustrative examples OMT suggests to describe associations

between particular instances. Such associations are called "links".

Optionally OMT allows to model associations as classes, too (pp. 33). Such an option makes a lot of sense at first sight: An association may have features which are not original features of the objects it links together. For instance: If two objects of class "Person" are linked by an association "married with", a feature like "date of marriage" could be assigned to the association. This is definitely more appropriate than assigning such a feature to each of the involved objects. Nevertheless it has its pitfalls, too. An object should have an identity independently from other objects. This condition does not hold for an association. Furthermore it has to be taken into account that the distinction between objects and associations provides an important orientation for conceptualizing real world domains: Objects are thought to correspond to real world objects - which can be linked by associations. The heuristic power of this orientation is seriously endangered by allowing for the slogan "everything is an object". Furthermore it is accompanied by a confusing increase in complexity: If associations are objects it is possible that there are associations between associations.

OMT provides a construct to model associations as aggregation (pp. 36). An aggregation is a specialized association that implies transitivity and anti-symmetry.[1] In principle OMT allows associations to be established between objects of an arbitrary number of different classes. It is however recommended to use binary associations only. In accordance with the notion of an object used in Smalltalk OMT allows classes to be regarded as objects themselves. That implies the introduction of "metaclasses" (p. 71). Like any class a metaclass is characterized by a set of attributes and operations which however can only be applied to a class not to instances. Typical examples are operations that serve to instantiate objects from a class or that inform about the number of active instances of a class.

In order to add further semantics to an object model, OMT allows to specify additional "constraints" (pp. 73). They serve to restrict the number of permissible states of an object or attribute depending on the states of other objects or attributes. For all of the concepts mentioned above OMT provides a graphical notation which apparently is inspired by the Entity Relationship Model.

1. "A contains B" and "B contains C" implies "A contains C" (transitivity). Anti-symmetry is a constraint that does not allow "A contains B" and "A is part of B" to be true at the same time.

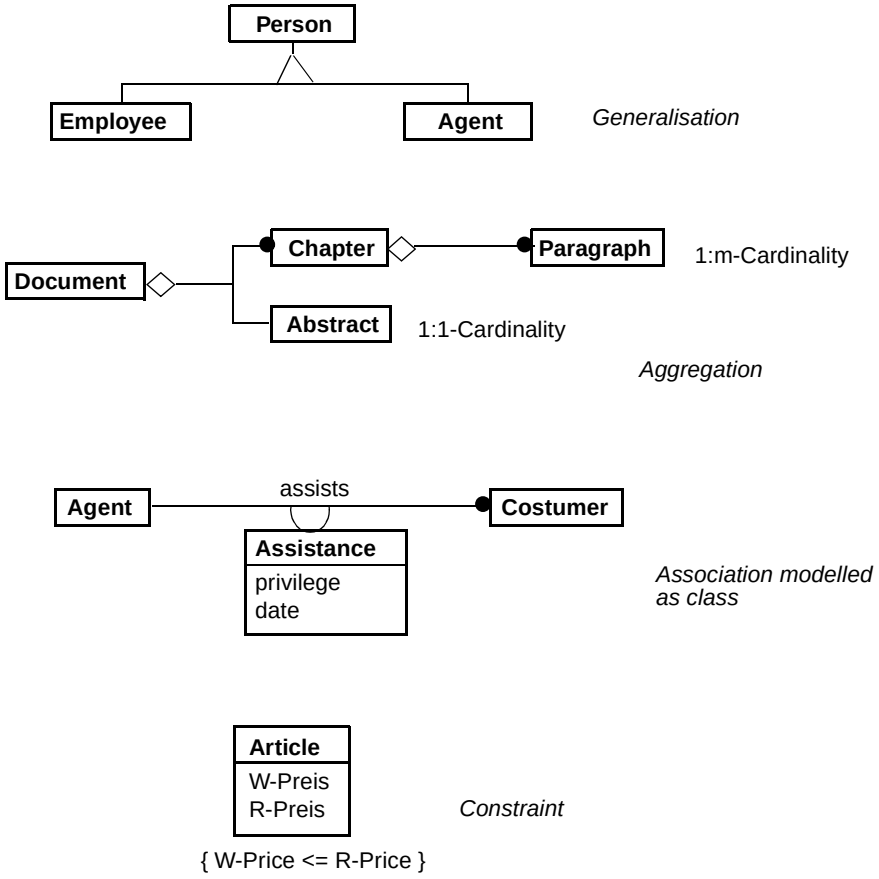

Fig. 3: Representation of selected concepts suggested by OMT

Classes within an object model which are closely related (for instance by forming a particular application) can be grouped within so called "modules" (p. 43). Since modules are less complex than the whole object model they foster maintainability. For this reason classes should be uniquely assigned to a module. For each module a set of interface for communicating with other modules is to be specified.

5.2 The Dynamic Model

The operations of a class are described within the object model. It does however not include any information on the conditions under which a service is invoked. The dynamic model is intended to provide for the mapping of such control structures (pp. 84). In other words: While the object model focuses on defining permissible object states, the dynamic model aims at describing conditions under

which certain state conditions occur. A dynamic model is composed of "state diagrams". A state diagram should be introduced for each class that allows for a behaviour of its objects that is not trivial. A state diagram is drawn as graph that is composed of symbols which represent events and states (and thereby implicitly: state transitions).

Events can be organized into generalization hierarchies, too. They may be characterized by attributes and services and they can inherit from superclasses. Such an approach seems to be reasonable since it is always desirable to abstract from single occurrences and to build classification schemes. It is however confusing at the same time: The graphical representation of the generalization hierarchy (p. 98) suggests that events are modelled like objects or classes. An event is usually associated with the occurrence of certain state of an information system in general or of a particular object. The occurrence of an object's state however is conceptually different from an object. According to OMT an event does not have to cause a state transition. Instead a condition may be specified that has to be fulfilled for the event to trigger a state transition (pp. 91). For instance: The event "the heater has been turned on" will only cause a transition to the state "the heater is operating" if the condition "temperature less than 20 degrees" is fulfilled. Introducing conditions in state diagrams helps to reduce the complexity of a net. Furthermore OMT allows to represent *activities* and *actions* within a state diagram. An activity is associated with a state. In order to facilitate the integration of a dynamic model with an object model it is helpful to model an activity as an operation of the class the state diagram is assigned to. The examples given by Rumbaugh et al. are all related to a technical background. A typical example is the state "ringing" (of an object of class "Telephone") which is associated with the activity "ring bell". An activity starts with the state it is assigned to and terminates either together with the state or while the state still exists.

In order to keep state diagrams illustrative they may be drawn on different levels of detail ("nested state diagrams", pp. 94): Several different states may be aggregated to one more general state (like "document is being processed"). A general state in turn may be decomposed into more detailed states. Different from an activity an action does not have a relevant duration. An action may be something like opening a window on a screen or presenting a pop-up menu. Actions are triggered by events. They directly cause a state transition.

Integrating state diagrams to a dynamic model is accomplished by referring to "shared events". Shared events are events which occur in a number of state diagrams. Objects are supposed to be concurrent: In principal they can autonomously change their states. Furthermore OMT allows to characterize the operations of one object as being concurrent. Figure 4 shows the representation of relevant aspects of state diagrams. In analogy to object models state diagrams may

also be drawn for specific instances. In this case Rumbaugh et al. speak of "scenarios".

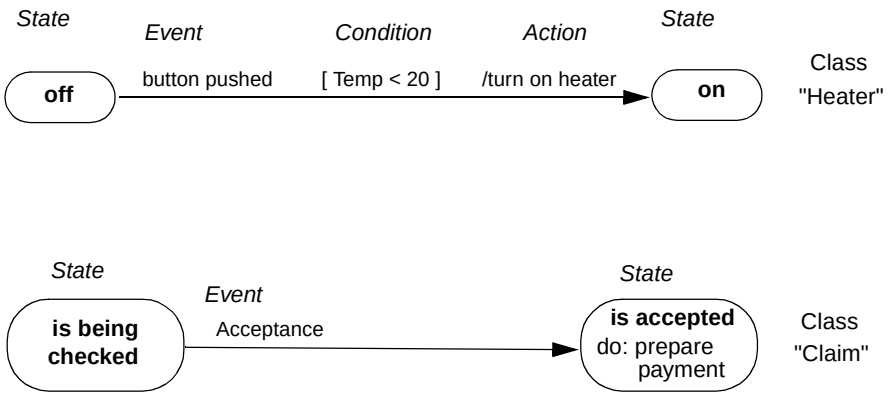

Fig. 4: Examples for concepts used with state diagrams in OMT

5.3 The Functional Model

Within the object model operations are merely defined by their interface. That is their name and - if necessary - a list of parameter types. The dynamic model informs about events which cause the execution of an operation and eventually about states which are associated with an operation. Neither the object model nor the dynamic model defines how an operation is done. For this purpose Rumbaugh et al. propose the so called "functiona model" (pp. 123). A functional model does not aim at an algorithmic description of an operation. Instead it serves to specify the input parameters and the resulting output of an operation. In order to give a more detailed description an operation may be decomposed into several smaller operations.

The functional model is represented by data flow diagrams (pp. 127). They are very similar to traditional data flow diagrams. The only differences: Interfaces of traditional data flow diagrams are called "*actor objects*", data stores are called "*data store objects*". The process represented by a data flow diagram corresponds with an operation of a particular class. Processes may be decomposed but not formally specified. Rumbaugh et al. only mention different approaches to allow for specification (like pseudo-code, pre- and postconditions, etc.).

As with the object model the functional model, too, does not consequently reflect object-oriented concepts. In correspondence to attributes being only defined by data types, the information flow between objects is restricted to data. This is a remarkable reduction of the message flow which may occur between objects. In

order to connect an object model to a functional an to a dynamic model, it is required that the three partial models are not developed independently. Instead it should be intended from the very beginning to name corresponding concepts within the three partial model in the same way. Coordinating the development of the partial models allows to establish references between corresponding parts: operations within a class diagram refer to a process represented in a data flow diagram. A data flow may refer to a set of attributes - of one or more classes. Activities and actions within the dynamic model may also correspond to operations within class diagrams. Figure 5 shows the relationships between concepts used in the three partial models.

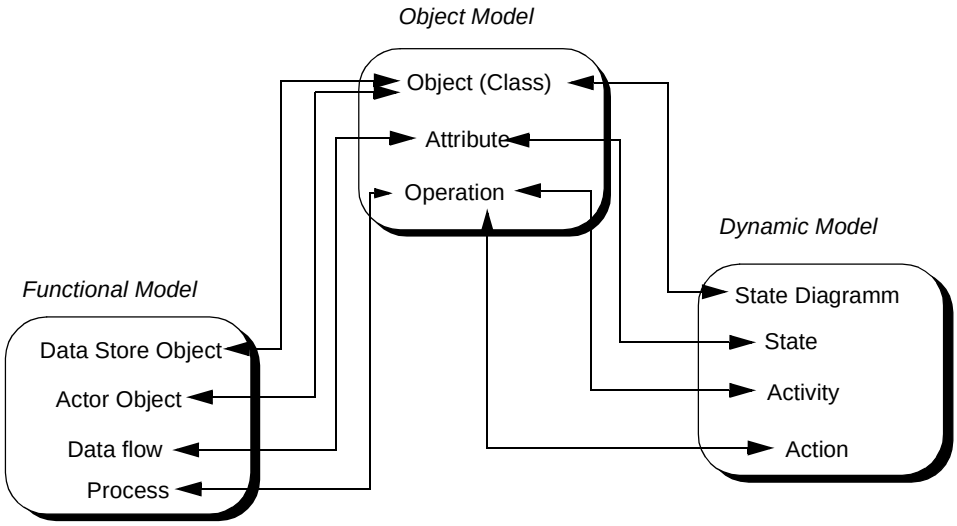

Fig. 5: Relationships between concepts used within the three partial models suggested by OMT

6. Booch's Methodology

Being a protagonist within the Ada-scene for long Booch is one of the pioneers of object-oriented modelling. After a number of early publications in the 1980s[1] he as presented a design methodology within an extensive textbook (Booch 1990). Although Booch conceptualizes objects similar to Rumbaugh et al. he does not distinguish between three partial models. His methodology is primarily focused on the specification of classes. In order to allow for a detailed description of classes Booch suggests various templates. The methodology not only covers

1. Booch (1982), Booch (1986 a), Booch (1986 b)

conceptual design ("logical view") but includes also techniques which are clearly associated with implementation issues ("module architecture", "process architecture"). The following description of Booch's approach is mainly focused on the logical view. All the pages referred to are in Booch (1990).

6.1 Class and Object Diagrams

Different from the object models of the OMT Booch distinguihes between class and object diagrams. This distinction is not to be confused with that between classes and instances: The elements used in object diagrams are abstractions from real world objects as well. The terminology Booch reflects different contexts and is certainly more consistent. This will become obvious when we get to associations which can be established between classes or between objects.

Classes can be interrelated by four types of associations (pp. 97): "*inheritance*", "*using*", "*instantiation*", "*metaclass*". Generalisation can be expressed via single or multiple inheritance. A "using relationship" between classes expresses that the specification of one class requires a reference to another class - for instance to declare parameters or for implementing an operation - i.e. if an operation needs the current time and therefore sends a message to class "Time". A using relationship can be assigned cardinalities. It is not question that managing such relationships is of crucial importance for a consistent maintenance of an object model. Furthermore one has to consider - although Booch does not explicitly mention it, that attributes (which are called "fields" by Booch) of a class also establish using relationships since they are defined by a class (not by a data type like in OMT).

There are classes that describe objects which serve as "containers" for other objects, i.e. arrays, sets, bags, dictionaries, etc. With regard to an information system's integrity it is desirable to restrict the permissible classes of the objects which may be assigned to a container. Booch allows to express an association between a container class and the classes of the contained objects by introducing a so called "instantiation relationship". Initializing a container class implies the instantiation of objects of the associated class(es). Cardinalites may be assigned to express additional restrictions. Booch allows a class to be considered as an object, too. In this case a "metaclass relationship" exists between a class and its metaclass. It means that the metaclass includes operations for instantiating and initializing the associated class. In other words: The semantics of a class, which has a metaclass assigned to it, is flexible to a certain degree.

Classes an associations existing between them can be visualized by class diagrams. For this purpose classes are represented by so called "amorphous blobs". In order to distinguish between the different types of associations Booch suggests a characteristic arc for each type (see fig. 6). Additionally a cardinality can be assigned to each arc.

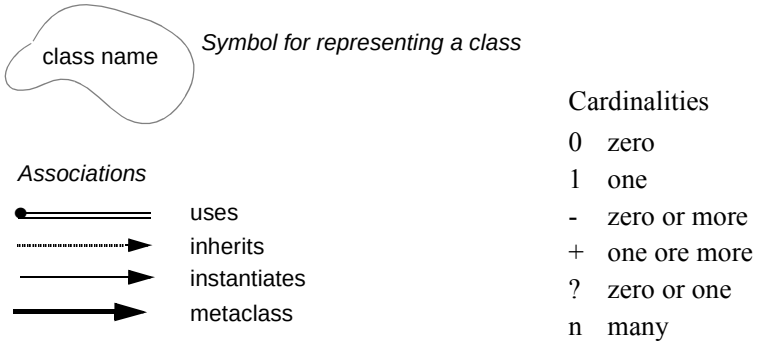

Fig. 6: Symbols for drawing class diagrams (S. 159)

Associations (or relationships) between classes define a linkage between concepts. Booch distinguishes those linkages from associations which exist between objects. Different from instance diagrams suggested by OMT Booch does not use specific instances. Instead an object within an object diagram represents any instance of a particular class. On this level Booch only distinguishes between two types of associations (pp. 88): aggregation ("containing relationship") and interaction ("using relationship"). A using relationship between two objects means that the objects should be able to interchange messages. In other words: They have to be able to identify each other. Objects within a using relationship may play one of three roles. An "*actor*" is an object, which uses services of another object without offering services itself. A "*server*" is an object that offers services without requesting any service. An "*agent*" is an object which can act as an actor and as a server at the same time. Aggregation is a using relationship that is specialized by adding transitivity and anti-symmetry.

For modelling the message flow between objects Booch gives up the distinction between interaction and aggregation and only considers the general case: "A relationship between two objects simply means that the objects can send messages to one another." (p. 170) Object diagrams are drawn in analogy to class diagrams. Objects are represented as clouds, too. Unlike for classes in class diagrams the clouds are drawn by full and not by dotted lines. A message name can be assigned to each arc representing an association between objects. The message names correspond with operations defined at another place in the model. Since an operation includes a definition of the required parameters and its output, an object diagram implicitly describes the permissible information flow between objects. That makes them comparable to the data flow diagrams used in OMT - with one important difference however: They reflect the actual way of information exchange in object systems - sending messages. This allows for a seamless integration with other parts of the model. Within an object diagram objects are

identified by the names of their classes. Beside the name of the corresponding operation a message can be characterized by further features. It can be assigned one of five synchronisation-types: "*simple*", "*synchronous*", "*balking*", "*timeout*", and "*asynchronous*". (pp. 126) Furthermore it can be specified whether a message is sent periodically or not. Since the description of classes (see below) includes similar characteristics they have to be checked for consistency.

6.2 Conceptualization of Classes

Booch allows to specify classes in a much more detailed way than OMT. For this purpose he introduces so called "class templates" which are structured as follows (p. 164):

Name:	identifier
Documentation:	text
Visibility:	exported | private | imported
Cardinality:	0 | 1 | n
Hierarchy	
Superclasses:	list of class names
Metaclass:	class name
Generic parameters:	list of parameters
Interface | Implementation (Public/Protected/Private):	
Uses:	list of class names
Fields:	list of field declarations
Operations:	list of operation declarations
Finite state machine:	state transition diagram
Concurrency:	sequential | blocking | active
Space Complexity:	text
Persistence:	static | dynamic

In order to reduce the complexity of large models, a set of classes can be grouped into categories (similar to the modules suggested by OMT). The feature *visibility* serves to define, whether a class should only be available within its category (which is surprisingly not included in the class template) or whether it may be imported by another category. *Cardinality* of a class describes how many instances may coexist (0: no instance has to exist, 1: exactly one, n: an arbitrary number). The permissible number of superclasses of a class depends on whether single or multiple inheritance has been chosen for a particular model. If the language used for implementation allows for it, a metaclass may be assigned. The use of "generic parameters" indicates - among others - that Booch is very much influences by his work on Ada. A generic parameter is a variable part of a generic class (as it may be defined in Ada or Eiffel). In order to create a usable class from a generic class, its generic parameters have to be instantiated (for instance with specific operations).

While a list of operations and a list of attributes ("fields") is essential for a class definition, a list of used classes seems somewhat redundant: Attributes (by their classes) and operations (by their parameter classes and other references) establish using relationships anyway. Different to the class itself Booch does not allow for assigning cardinalities to attributes. Attributes, operations, and associations can each be characterized by one of three access levels. "*Private*" means, that they cannot be used from outside, "*protected*" is to say that there has to be some sort of privilege in order to get access, while "*public*" means that any object can use them. The distinction between "interface" and "implementation" at this point is confusing and redundant anyway: During design one should not bother with details that do not add semantics to the conceptualization of a class but only to its implementation. Booch is aware of this fact and points out, that those specifications are optional (p. 165). As already mentioned above an attribute's semantics is defined by a class not by a data type. An attribute can be either characterized as a class variable (it is only available on the class level - if you consider a class as an object, too) or as an instance variable (available within every instance). While such a distinction may be important for implementation purposes (provided the implementation language allows for it) it is again kind of confusing on the conceptual level.

Assigning state transition diagrams allows to describe the permissible behaviour of a class. The slot named "concurrency" is intended to describe how communication with other objects is synchronized. While synchronisation is an inherent feature of sequentialized processes, concurrent processes may be synchronized in two ways: "Blocking objects" are objects, which are non-active, that is they can execute operations only after they have received a corresponding message. "Active objects" have an autonomous behavior, that is they may invoke their operation themselves. In other words: A passive object's operations are sequentialized (whereas a number of objects may execute operations in parallel), while active objects can execute their operations concurrently.

With regard to implementation issues "*space complexity*" allows to specify the memory required by one object of the class. It is doubtful if such information is helpful during analysis. The exact amount of memory depends on the implementation language and on the hardware anyway. An object's "*persistence*" can be characterized as "static" or "dynamic" - depending whether its lifetime is determined by the program it is used in or not. Again it can be argued that this may be a decision that is only relevant for implementation purposes.

Beside classes Booch allows to describe so called "*class utilities*". A class utility corresponds with similar concepts in programming languages like Ada and C++ ("free subprograms"). On my opinion they are not appropriate for an object-oriented design methodology since they to not correspond with the object-oriented

paradigm where operations are only provided by objects.

In order to instruct the specification of operations Booch suggests another template (p. 166):

Name:	identifier
Documentation:	text
Category:	text
Qualification:	text
Formal Parameters:	list of parameter declarations
Result::	class name
Preconditions:	PDL / object diagram
Actions:	PDL / object diagram
Postconditions:	PDL / object diagram
Exceptions:	list of exception declarations
Concurrency:	sequential | guarded | concurrent | multiple
Time Complexity:	text
Space complexity:	text

According to the grouping of methods of a class used in Smalltalk Booch allows to categorize the operations of a class. "*Qualification*" reflects a concept provided by programming languages like CLOS, where Operations can be qualified - like ":before", ":after", ":around". The list of parameter declarations contains the classes of the parameters which have to be included in the message that requests the operation. An operation's "*result*" is defined by the class of the returned object. Booch provides a number of different means to characterize an operation's flow of control. Non of them is mandatory. While pre- and postconditions only define additional constraints at certain points in time, "actions" serves to describe the flow of control. For this purpose Booch recommends to use a *programming description language* (PDL) - in other words: pseudo-code. Alternatively it can be referred to an object diagram (see above) at this place. An object diagram however does not define the control flow of an operation but only the permissible message flow. Booch does not specify the level of detail appropriate for the description of "actions": "... an appropriate level of detail for each operation should be chosen, given the needs of the eventual reviewer." (p. 167) Exceptions which may occur during the execution of an operation, should be characterized by an exception type, whereas exception types should be defined in a unified way for the whole model - like "printer out of paper", or "read error accessing disk" ... This is essential for implementing a unified exception handling for a whole information system.

"*Concurrency*" allows to define, whether an operation may be reentrant or not and how synchronisation should be accomplished. These specifications need to be compatible with those made about concurrency within the class template. "*Time complexity*" and "*space complexity*" serve to capture information about performance and the needed memory.

6.3 Dynamic Aspects

Although operations and message flow can be described on a very detailed level, that does not tell anything about the conditions that cause an operation to be invoked. In order to cover these temporal and behavioural aspects Booch suggests "*timing diagrams*" and "*state transition diagrams*".

Like OMT Booch uses state transition diagrams to describe permissible state transitions of the instances of *one* class. The states should possibly correspond to certain states of an objects attributes. The transitions are characterized by an event and a particular operation of the class.

In order to describe an information system's behaviour it is not sufficient to look at the behaviour of objects in an isolated way. Instead it is required to take into account the interaction of objects of a number of classes. The timing diagrams suggested by Booch allow to illustrate the sequence in which operations of various objects are executed. Such a diagram - like it is also used to illustrate the allocation of a processor's capacity within the process management of operating systems - serves to indicate to which degree operations can be executed independently. This is of special importance when objects can be assigned to one of many processors. A timing diagram however does not contain any information about the events that trigger the execution of an operation. Therefore they are to be seen as a supplement to state transitions diagrams. Their integration with other diagrams is straightforward. It can be done be using unified operation names.

7. Comparative Evaluation

Both OMT and Booch's methodology are suited to cover the essential aspects of system design. Booch's approach is certainly more sophisticated and supports a more consequent object-orientation. One the one hand this is due to the terminology he uses[1], on the other hand Booch provides a clearer orientation for integrating the different aspects of a model than Rumbaugh et al.: Classes and their features serve as common references for all aspects. This advantage however is reduced by the confusion that Booch produces by including concepts which are language specific (i.e. "class utilities", "generic parameters").

While OMT sometimes lacks an appropriate degree of precision and detail[2],

1. His definition of a class, however, is not convincing on my opinion. Booch (1990, p. 93) defines a class as "a set of objects that share common structure and a common behavior". Such a definition can easily be confused with the set of existing instances. Furthermore it can hardly be applied to abstract classes, which Booch explicitly allows for.
2. Hsieh (1992, p. 26) presents - however without giving convincing reasons - a quite different judgement: "By far we feel that Rumbaugh's methodology is the most comprehensive and complete one ...".

Booch's methodology is somewhat overloaded: It is not only restricted to design aspects but also includes implementation issues. This may be caused by the fact that Booch's perspective is clearly influenced by his work on the design of programming languages. It does not have to be a mistake to consider details, which may become relevant only during implementation, in time. In order not to harm the descriptive power of a conceptual model, I would prefer to introduce an additional level of abstraction for describing those details. In order to allow for a precise definition of associations it is certainly helpful to distinguish between classes and objects. However, the different types of associations Booch suggests are confusing sometimes. It is for instance questionable whether there is a difference between a "using relationship" on the class level and a "containing relationship" on the object level.[1]

The following comparison of both methodologies refers to the evaluation framework presented above.

Static Aspects

Evalutation

			Rumbaugh et al.	Booch
Attribute	hardware-oriented	definable as value range	yes	yes
		by functional specification	no	no
	application-oriented	by enumeration	yes	no
		by entity type	no	yes
	Cardinality	minimum = 0	no	yes
		maximum > 1	no	yes
	access privilege		no	yes
	associations		no	no
	an entity's identity is determined by its state		no	no

1. The second case is more special than the first. But I doubt that this difference is of any importance for system design.

Strukturelle Aspekte

Evaluation

				Rumbaugh et al.	Booch
History			general	no	no
			partial	no	no
Associations between entities	Cardinality		single	yes	yes
		min-max	restricted	no	yes
			free	no	no
	Direction		role	yes	no
			inverse association	no	no
	referential integrity			yes	yes
	aggregation			yes	yes
	miscellaneous			yes	yes
Genera-lisation	single			ja	yes
	multiple			yes	yes
Views					
Entity	specification of views			no	no
	calibration of access privileges			no	no
	user-interface			no	no
Collection	specifikation of views			no	no
	user-interface			no	no

Dynamic/functional Aspects Evalutaion

		Rumbaugh et al.	Booch
Operations/Functions			
specification of interfaces		yes	yes
preconditions		no	yes
postconditions		no	yes
trigger		no	no
exception handling		no	yes
System behavior			
Control structures	for entities	yes	yes
	for collections	no	restrictedly
Miscellaneous Aspects			
Modula-rization	coherence	yes	yes
	referencing	-	-
Buildtime- and runtime-support			
DBMS	DDL	no	no
	DML	no	no
	standards	no	no
	controlling system behavior	yes	yes
tool support		yes	yes

8. Concluding Remarks

We can summarize that both methodologies allow for designing comprehensive conceptual models on a high level of semantics, where certain restrictions apply to OMT which is in part more data- than object-oriented. Both approaches allow to integrate the different modelling aspects, Booch however provides better guidance in this respect. There is a little difference in preparing the transformation to the implementation level. Both methodologies are intended to support object-oriented programming languages. While the examples Booch provides cover a wider range of programming languages, Rumbaugh et al. give examples for the transformation into a relational model, too. General Electric as well as Booch offer tools to support their methodologies ("OMT-Tool" and "Rational Rose"). Additionally both methodologies are supported by other tools of independent vendors.

The design of user-interfaces is only supported on a technical level - for instance by using state transition diagrams - not on an application level. Thinking of the increasing number of libraries which support the implementation of graphical user-interfaces a mere technical view does not seem to be sufficient. Instead it could be referred to constructs like windows, widgets, etc. For this purpose a design methodology should provide an appropriate selection of such constructs and foster their integration with other aspects of the model.

Both methodologies promise obvious software engineering advantages over conventional approaches of conceptual modelling. Nevertheless many developers will probably find their contribution to be doubtful. Both methodologies are rather complex which is specially true for Booch's approach. Therefore it requires a remarkable effort to apply them - although the textbooks offer a comprehensive description. While a conceptual model is a document intended for software developers at first place it should ease communication with domain experts as well. The graphical notations used in both methodologies are hardly satisfactory in this respect.

Rumbaugh et al. consider the transformation of their model into the schema of a RDBMS. It is however no question that an OODBMS is required in order to exploit the potential of object-oriented design to its full extent. That in turn requires standardized object models. For this reason the work of organizations like the OMG is of crucial importance. However, the current state of the art (see for instance OMG 1992) indicates that there is still some substantial research to be done.

References

Ackroyd, M.; Daum, D. (1991): Graphical notation for object-oriented design and programming. In: JOOP, Vol. 3, No. 5, pp. 18-28

Alabisco, B. (1988): Transformation of data flow analysis models to object-oriented design. In: Meyrowitz, N. (Ed.): OOPSLA ’88 - conference proceedings, San Diego, Sept. New York, pp. 335-353

Bailin, S.C. (1989): An object-oriented requirements specification method. In: Communications of the ACM, Vol. 32, No. 5, pp. 608-623

Berard, E. (1986): An Object-Oriented Design Handbook. Rockville, MD

Booch, G. (1990): Object-oriented Design with Applications. Redwood

Booch, G. (1986 a): Object-Oriented Development. In: IEEE-Transactions on Software-Engineering. Vol. 12, No. 2, pp. 211-221

Booch, G. (1986 b): Software Engineering with Ada. Menlo Park, Ca.

Booch, G. (1982): Object-Oriented Design. In: Ada-Letters, Vol. 1, No. 3, pp. 64-76

Bourbaki, N. (1991): Toward a definition of object-oriented languages, part 1. In: JOOP, March/April, pp. 62-65

Buhr, R. (1984): System Design with Ada. Englewood Cliffs, NJ

Cherry, G. (1987): PAMELA 2: An Ada-Based Object-Oriented Design Method. Reston, Va.

Coad, P.; Yourdon, E. (1991): Object Oriented Design. Englewood Cliffs, NJ

Coad, P.; Yourdon, E. (1990): Object-oriented analysis. Englewood Cliffs., NJ.

Cunningham, W.; Beck, K (1986): A diagram for object-oriented programs. In: Meyrowitz, N. (Ed.): OOPSLA ’86 - conference proceedings, Portland, Sept.. New York, pp. 39-43

De Champeaux, D.; Faure, P. (1992): A comparative study of object-oriented analysis methods. In: JOOP, No. 1, Vol. 5, pp. 21-33

Desfray, P. (1990): A Method for Object Oriented Programming: The Class-Relationship Model. In: Bezivin, J.; Meyer, B.; Nerson, J.-M. (Ed.): TOOLS 2 - Technology of object-oriented languages and systems. Paris pp. 121-131

Edwards, J. (1989): Lessons learned in more than ten years of practical application of the Object-Oriented Paradigm. London

Embley, David W.; Kurtz, Barry D.; Woodfield, Scott N. (1992): Object-oriented systems analysis- a model-driven aproach. Englewood-Cliffs, NY

European Space Agency (ESA) (1989): HOOD Reference Manual. Issue 3.0. Nordwijk

Felsinger, R. (1987): Object-Oriented Design, Structured Analysis/Structured Design, and Ada for Real-time Systems. Mt. Pleasant, SC

Ferstl, O.K.; Sinz, E.J. (1991): Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM). In: Wirtschaftsinformatik, 33. Jg., Heft 6, pp. 477-491

Ferstl, O.K.; Sinz, E.J. (1990): Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM). In: Wirtschaftsinformatik, 32. Jg., Heft 6, pp. 566-581

Firesmith, D. (1992): Object-oriented requirements analysis and logical design. Chichester

Henderson-Sellers, B.; Constantine, L.L. (1991): Object-oriented development and functional decomposition. In: JOOP, Vol. 3, No. 5, pp. 11-16

Hewlett Packard (1991): An Evaluation of Five Object-Oriented Development Methods. Software Engineering Department, HP Laboritories. Bristol

Hong, S.; Goor, G. (1993): A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies. In: Nunamaker, J.F.; Sprague, R.H. (Ed.): Information Systems: Collaboration Technology, Organizational Systems, and Technolgoy. Proceedings of the 26th International Hawaii International Conferenc on System Sciences. Los Alamitos, pp. 689-698

Hsieh, D. (1992): Survey of object-oriented analysis/design methodologies and future CASE frameworks. Menlo Park, Ca.

Jacobson, I.; Christerson, M; Jonsson, P; Overgaard, G. (1992): Object-Oriented Engineering. A Use Case Driven Approach. Reading, Mass.

Johnson, R.E.; Foote, B. (1985): Designing Reusable Classes. In: JOOP, Vol. 1, No. 2, pp. 22-35

Kadie, C. (1986): Refinement Through Classes: A Development Methodology for Object-Oriented Languages. Urbana, Il.

Kappel, G.; Schrefl, M. (1991): Using an Object-Oriented Diagram Technique for the Design of Information Systems. In: Sol, H.G.; Van Hee, K.M. (Ed.): Dynamic Modelling of Information Systems. Amsterdam, New York u.a. pp. 121-164

King, R. (1989): My Cat is Object-Oriented. In: Kim, W.; Lochovsky, F.H. (Ed.): Object-Oriented Concepts, Databases, and Applications. New York, pp. 23-30

Lee, S.; Carver, D.L. (1991): Object-oriented analysis and specification: a knowl-

edge base approach. In: JOOP, Vol. 3, No. 5, pp. 35-43

Liskov, B.; Guttag, J. (1986): Abstraction and specification in program development. Cambridge, Mass.

Mannino, P. (1987): A Presentation and Comparison of Four Information System Development Methodologies. In: Software Engineering Notes, Vol. 12, No. 2, pp. 26-27

Masiero, P., Germano, F. (1988): JSD as an Object-Oriented Design Method. In: Software Engineering Notes. Vol. 13, No. 3, pp. 22-23

McGregor, J.D.; Sykes, D.A. (1992): Object-Oriented Software Development. Engineering for Reuse. New York

Monarchi, D.E.; Puhr, G. (1992): A Research Typology for Object-Oriented Analysis and Design. In: Communications of the ACM, Vol. 35, No. 9, pp. 35-47

Mullin, M. (1989): Object-Oriented Design with Examplex in C++. Reading, Mass.

Nielsen, K. (1988): An Object-Oriented Design Methodology for Real-Time System in Ada. San Diego, Ca.

Odell, J.J. (1992): Modelling objects using binary-and-entity-relationship approaches. In: JOOP, June 1992, Vol. 5, No. 3, pp. 12-18

Object Management Group (OMG)/Object Model Task Force (1992): OMG Architecture Guide 4. The OMG Object Model. Draft July, Framingham, Mass.

Page, T.W.; Berson, S.E.; Cheng, W.C.; Muntz, R.R. (1989): An Object-Oriented Modeling Environment. In: Meyrowitz, N. (Ed.): Object-Oriented Programming: Systems, Languages and Applications. New York, pp. 287-296

Rajlich, V.; Silva, J. (1987): Two Object-Oriented Decomposition Methods. Detroit

Rentsch, T. (1982): Object-Oriented Programming. In: SIGPLAN Notices, Vol. 17, No. 12

Robinson, P. (1992): Object-Oriented Design. London

Rumbaugh, J. et al. (1991): Object-oriented Modelling and Design. Englewood Cliffs, N.J.

Seidewitz, E.; Stark, M. (1987): Towards a General Object-Oriented Software Development Methodology. In: Peterson, G.E. (Ed.): Object-Oriented Computing - Tutorial. Vol. 2: Implementations. Washington, DC. pp. 16-29

Shlaer, S.; Mellor, S.J. (1992): Object lifecycles- modeling the world in states. En-

glewood Cliffs, N.J.

Velho, A.V.; Carapuca, R. (1992): SOM: A Semantic Object Model - Towards an Abstract, Complete and Unifying Way to Model the Real World. In: Sol. H. (Ed.): Proceedings of the Third International Working Conference on Dynamic Modelling of Information Systems. Delft, pp. 65-93

Wasserman, A.I.; Pircher, P.A.; Muller, R.J. (1990): The Object-Oriented Structured Design Notation for Software Representation. In: Computer 23, No. 3, pp. 50-63

Wirfs-Brock, R.J.; Wilkerson, B. (1990): Designing Object-Oriented Software.