DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

ANALYSIS AND DESIGN TECHNIQUES FOR OBJECT-ORIENTED DATABASES

by
Selim BUDAKOĞLU

September, 2005
İZMİR

ANALYSIS AND DESIGN TECHNIQUES FOR OBJECT-ORIENTED DATABASES

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of
Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering, Computer Engineering Program.

by
Selim BUDAKOĞLU

September, 2005
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"ANALYSIS AND DESIGN TECHNIQUES FOR OBJECT-ORIENTED DATABASES"** completed by **Selim BUDAKOĞLU** under supervision of **Prof. Dr. Alp KUT** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Alp KUT
Supervisor

(Jury Member)

(Jury Member)

Prof.Dr. Cahit HELVACI
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGMENTS

First, I would like to thank my supervisor, Prof. Dr. Alp KUT, for his guidance and assistance in this thesis.

I also thank my friends Volkan SALGAR, Şerife KAPUKAYA and A.Selçuk ÖZTÜRK for their support and motivation.

Finally, I would like to thank my whole family for their endless support and encouragement.

Selim BUDAKOĞLU

ANALYSIS AND DESIGN TECHNIQUES FOR OBJECT-ORIENTED DATABASES

ABSTRACT

Object-oriented analysis and design techniques for object-oriented database management systems are rapidly evolving and these techniques same as for the object-oriented programming languages. None of these techniques has achieved the status of a widely recognized standard on the order of the conventional techniques. These techniques will continue to evolve, as did conventional techniques before them.

These major techniques reviewed in these project are Object-Oriented Analysis and Object-Oriented Design from Coad and Yourdon, Designing Object-Oriented Software from Wirfs-Brock, Object Modeling Technique from Rumbaugh, Object-Oriented Analysis and Design with Applications from Booch, Object Lifecycles from Shlaer and Mellor, Object-Oriented Analysis and Design from Martin, The Fusion method from Coleman and Object-Oriented Software Engineering from Jacobson. All these techniques have common properties which are truly object-oriented, extensively described in a textbook, widely available and regarded a popular method.

Keywords : object-oriented analysis, object-oriented design, object-oriented database management system

NESNE YÖNELİMLİ VERİTABANLARINDA ANALİZ VE DİZAYN TEKNİKLERİ

ÖZ

Nesne-yönelimli veritabanı yönetim sistemleri için nesne-yönelimli analiz ve dizayn teknikleri hızlı bir şekilde gelişiyor. Bu tekniklerin tamamı nesne-yönelimli programlama dillerindeki teknikler ile aynı teknikler. Bu tekniklerin herhangibiri henüz geleneksel teknikler gibi geniş alanda tanınan bir standart haline gelemediler fakat geleneksel tekniklerin daha önce yaptığı gibi gelişmeye devam edecekler.

Bu proje dahilinde incelenen ana teknikler; Coad ve Yourdon imzalı Nesne-Yönelimli Analiz ve Nesne Yönelilmli Dizayn, Wirfs-Brock imzalı Nesne-Yönelimli Yazılım Dizaynı, Rumbaugh imzalı Nesne Modelleme Tekniği, Booch imzalı Uygulamalı Nesne-Yönelimli Analiz ve Dizayn, Shlaer ve Mellor imzalı Nesne Yaşamdöngüsü, Martin imzalı Nesne-Yönelimli Analiz ve Dizayn, Coleman imzalı Birleşme Metodu ve Jacobson imzalı Nesne-Yönelimli Yazılım Mühendisliği. Bu tekniklerin hepsi ortak özelliklere sahipler. Bu özellikleri sıralamak gerekirse hepsi tam anlamıyla nesne-yönelimli, kapsamlı bir şekilde kitaplarda tanımlanmış, geniş alanda mevcut ve popüler bilinen teknikler.

Anahtar Sözcükler : nesne-yönelimli analiz, nesne-yönelimli dizayn, nesne-yönelimli veritabanı yönetim sistemleri

CONTENTS

Page

CHAPTER ONE
INTRODUCTION

1.1 Object-Oriented Database

1.1.1 Purpose and Origin

Object-oriented databases (OODBs) evolved from a need to support object-oriented programming and to reap the benefits, such as system maintainability, from applying object orientation to developing complex software systems. The first OODBs appeared in the late 1980s. OODBs are based on the object model and use the same conceptual models as Object-Oriented Analysis, Object-Oriented Design and Object-Oriented Programming Languages. Using the same conceptual model simplifies development; improves communication among users, analysts, and programmers; and lessens the likelihood of errors.

1.1.2 Technical Detail

OODBs are designed for the purpose of storing and sharing objects; they are a solution for persistent object handling. Persistent data are data that remain after a process is terminated.

There is no universally-acknowledged standard for OODBs. There is, however, some commonality in the architecture of the different OODBs because of three necessary components: object managers, object servers, and object stores. Applications interact with object managers, which work through object servers to gain access to object stores.

OODBs provide the following benefits:

- OODBs allow for the storage of complex data structures that can not be easily stored using conventional database technology.

- OODBs support all the persistence necessary when working with object-oriented languages.
- OODBs contain active object servers that support not only the distribution of data but also the distribution of work (in this context, relational database management systems (DBMS) have limited capabilities).

In addition, OODBs were designed to be well integrated with object-oriented programming languages such as C++ and Smalltalk. They use the same object model as these languages. With OODBs, the programmer deals with transient (temporary) and persistent (permanent) objects in a uniform manner. The persistent objects are in the OODB, and thus the conceptual walls between programming and database are removed. As stated earlier, the employment of a unified conceptual model greatly simplifies development.

1.2 Object-Oriented Analysis

1.2.1 Purpose and Origin

Object-oriented analysis (OOA) is concerned with developing software engineering requirements and specifications that expressed as a system's object model (which is composed of a population of interacting objects), as opposed to the traditional data or functional views of systems. OOA can yield the following benefits: maintainability through simplified mapping to the real world, which provides for less analysis effort, less complexity in system design, and easier verification by the user; reusabilityof the analysis artifacts which saves time and costs; and depending on the analysis method and programming language, productivity gains through direct mapping to features of Object-Oriented Programming Languages.

1.2.2 Technical Detail

An object is a representation of a real-life entity or abstraction. For example, objects in a flight reservation system might include: an airplane, an airline flight, an

icon on a screen, or even a full screen with which a travel agent interacts. OOA specifies the structure and the behavior of the object- these comprise the requirements of that object. Different types of models are required to specify the requirements of the objects. The information or object model contains the definition of objects in the system, which includes: the object name, the object attributes, and object relationships to other objects. The behavior or state model describes the behavior of the objects in terms of the states the object exists in, the transitions allowed between objects, and the events that cause objects to change states. These models can be created and maintained using CASE tools that support representation of objects and object behavior.

OOA views the world as objects with data structures and behaviors and events that trigger operations, or object behavior changes, that change the state of objects. The idea that a system can be viewed as a population of interacting objects, each of which is an atomic bundle of data and functionality, is the foundation of object technology and provides an attractive alternative for the development of complex systems. This is a radical departure from prior methods of requirements specification, such as functional decomposition and structured analysis and design.

1.3 Object-Oriented Design

1.3.1 Purpose and Origin

Object-oriented design (OOD) is concerned with developing an object-oriented model of a software system to implement the identified requirements. OOD can yield the following benefits: maintainability through simplified mapping to the problem domain, which provides for less analysis effort, less complexity in system design, and easier verification by the user; reusability of the design artifacts, which saves time and costs; and productivity gains through direct mapping to features of Object-Oriented Programming Languages.

1.3.2 Technical Detail

OOD builds on the products developed during Object-Oriented Analysis (OOA) by refining candidate objects into classes, defining message protocols for all objects, defining data structures and procedures, and mapping these into an object-oriented programming language (OOPL). Several OOD methods describe these operations on objects, although none is an accepted industry standard. Analysis and design are closer to each other in the object-oriented approach than in structured analysis and design. For this reason, similar notations are often used during analysis and the early stages of design. However, OOD requires the specification of concepts nonexistent in analysis, such as the types of the attributes of a class, or the logic of its methods.

Design can be thought of in two phases. The first, called high-level design, deals with the decomposition of the system into large, complex objects. The second phase is called low-level design. In this phase, attributes and methods are specified at the level of individual objects. This is also where a project can realize most of the reuse of object-oriented products, since it is possible to guide the design so that lower-level objects correspond exactly to those in existing object libraries or to develop objects with reuse potential. As in OOA, the OOD artifacts are represented using CASE tools with object-oriented terminology.

CHAPTER TWO
THE OBJECT-ORIENTED ANALYSIS AND DESIGN METHOD BY COAD & YOURDON

2.1 Background

Coad & Yourdon mention 7 key motivations and benefits in favor of OOA/OOD instead of using traditional analysis methods.

These motivations and benefits are:

- Tackle more challenging problem domains
- Improve analyst and problem domain expert interaction
- Increase the internal consistency across analysis, design and programming
- Explicitly represent commonality between classes and objects
- Build specifications resilient to change
- Reuse OOA, OOD and OOP results
- Provide a consistent underlying representation for analysis, design and programming

According to Coad & Yourdon the main result of Object Oriented Analysis and Design (OOA/OOD) comes from a reduction of complexity of a problem and the system's responsibilities within it. The OOA/OOD method is based on a number of general principles for managing complexity. According to Coad & Yourdon the Object Oriented approach is the only way to provide all these principles.

The Object Oriented approach is based on a uniform underlying representation (Classes and Objects) which according to Coad & Yourdon leads to a number of major implications:

- No major difference between analysis and design notations
- No 'transition' from analysis to design

- No waterfall model has to be followed, spiral and incremental are also possible
- There are still different skills and strategies needed for analysts and designers
- There is a uniform representation from OOA to OOD to OOP (OO programming)

For Coad & Yourdon an object oriented approach consists of classes, objects, inheritance and communication with messages.

2.2 General Approach

The OOA part of the method consists of five major activities:

- Finding Class & Objects
- Identifying Structures
- Identifying Subjects
- Defining Attributes
- Defining Services

It is mentioned at several places in the OOA/OOD method that these steps are not sequential steps. According to these major activities, an OOA model that is built during Analysis consists of five layers:

- A Subject layer
- A Class & object layer
- A Structure layer
- An Attribute layer
- A Service layer

The OOD part adds to the five layers four different components that have to be designed for these layers:

- Human Interaction Component
- Problem Domain Component
- Task Management Component
- Data Management Component

These design steps are also not sequential steps.

2.3 Concepts and Constructs

2.3.1 Concepts

- **Object**: An abstraction of an entity (tangible or intangible) about which information has to be kept; an encapsulation of Attribute values and their exclusive services.
- **Class**: A description of one or more Objects with a uniform set of Attributes and Services, including a description of how to create new objects in the class.
- **Attribute**: An attribute is some data (state information) for which each Object in a Class has its own value.
- **Class & Object**: A term meaning "a Class and the Objects in that Class"
- **Subject**: Subjects are a mechanism for partitioning large, complex models. Subjects are also helpful for organizing work packages on larger projects, based upon initial OOA investigations.
- **Service**: A Service is a specific behavior that an object is responsible to exhibit.
- **State**: The State of an Object is the value of its Attributes
- **Transition**: Transition is the change of a state
- **Condition**: If/precondition/trigger/terminate action
- **Text Block**: Text
- **Loop**: While/Do/Repeat/trigger/terminate action

2.3.2 Relationships between Objects

2.3.2.1 Whole-Part Structures

In Whole-Part relationships one object (the object representing the Whole) is decomposed of other objects (the Parts). There can be three variations of the whole-part relationship:

- Assembly-Parts: A car has wheels, an engine,...
- Container-Contents: A box consists of a number of nails
- Collection-Members: An organization has a number of analysts

Whole-Part relationships may be associated with specific amounts or ranges, like the cardinality concept in ER modeling.

2.3.2.2 Instance Connections (Association Relationship)

An instance connection is a connection that one object needs with other objects in order to fulfill its responsibilities.

For example: An object Person works at an object Office
For Instance Connections an amount or range can also be denoted.

2.3.2.3 Message Connections

A message connection models the processing dependency of an Object, indicating a need for Services in order to fulfill its own responsibilities (services).

For example: For fulfilling the responsibility (service) "video tape return" the Object Store Assistant needs the Service "access" from the object Rental.

2.3.3 Relationships between Classes

2.3.3.1 Generalization-Specialization (Gen-Spec) Structures (Inheritance Relationships)

Gen-Spec relationships between classes define an inheritance hierarchy for classes that are specializations of other classes. A class may both inherit from one superclass (single inheritance) or from more superclasses (multiple inheritance).

For example: a Diesel Engine is a specialization of an Engine.

2.3.4 Operations and Communication

The types of services the OOA method provides are:

- Algorithmically simple services like Create, Connect, Access and Release
- Algorithmically complex services:
    - Calculate services to calculate a result from the attribute values of an object.
    - Monitor services to monitor an external system or device.

In the OOA method objects communicate by sending messages. The sender object "sends" a message, that is "received" by the receiver object that takes some action and returns a result to the sender object. In the OOA diagram this communication is denoted by Message Connections.

2.4 Techniques

The result of applying OOA/OOD results in one main OOA diagram consisting of the five layers:

- Subject layer as a partitioning mechanism

- Class & Object layer to capture Classes and Objects
- Structure layer to capture inheritance and whole-part structures
- Attribute layer to capture attributes and instance connections between Class & Objects
- Service layer to capture methods and message connections between Class & Objects

The dynamic behavior of Classes can be captured in Object State Diagrams, a restricted form of State Transition Diagrams. The algorithms that have to be applied for services can be described by Service Charts, which are a kind of flowcharts.

The connection between a Service from a Service Chart and the States from an Object State Diagram can be established by a Service/State Table. Object State Diagrams, Service Charts and Service/State Tables are not described very well or extensively in the OOA/OOD books.

2.5 Analysis and Design Processes

For the steps in the OOA/OOD method it is an important point to check previous OOA results in the same and similar problem domains. This is a key point because reuse is one of the main issues in the object oriented approach.

It is also important that the steps described below do not have to be followed in the given sequential order. For large systems it is often better to refine the problem domain into several preliminary subjects, and then start with identifying Class & Objects.

2.5.1 Object Oriented Analysis (Activity 1)

2.5.1.1 Identifying Class & Objects (Activity 1.1)

The first step in finding Class & Objects is studying the problem domain (activity 1.1.1).

Finding potential Class & Objects (activity 1.1.2) can be done by looking at:

- Structures
- Other systems
- Devices
- Things or events remembered
- Roles played
- Operational procedures
- Sites (Physical locations)
- Organizational units

When potential Class & Objects are found and named (activity 1.1.3) they have to be challenged against the following criteria (activity 1.1.4):

- Needed remembrance
- Needed behavior
- (Usually) multiple attributes
- (Usually) more than one object in a class
- Always-applicable attributes
- Always-applicable services
- Domain-based requirements
- Not merely derived results

After this is done the Class & Objects can be added to the OOA diagram (activity 1.1.5).

2.5.1.2 Identifying Structures (Activity 1.2)

There are two kinds of structures: Gen-Spec and Whole-Part structures.

To find Gen-Spec structures (activity 1.2.1), each class has to be considered as a Generalization and the next questions have to be asked:

- Is it in the problem domain?
- Is it within the problem's responsibilities?
- Will there be any inheritance?
- Will the Specialization/Generalization meet the Class & Objects criteria?

Next, each class has to be considered as a Specialization and the above questions have to be asked again.

For identifying Whole-Part structures (activity 1.2.2) the next appearances are possible:

- Assembly-Parts
- Container-contents
- Collection-members

Each object has to be considered as a whole and as a part and the next questions have to be asked:

- Is it in the problem domain?
- Is it within the problem's responsibilities?
- Does it capture more than just a status value?
- If not: include attribute within the whole
- Does it provide a useful abstraction?

After identifying Gen-Spec and Whole-Part structures multiple structures can be identified (activity 1.2.3). Multiple structures include various combinations of Gen-Spec, Whole-Part structures or both. After identifying multiple structures the structures can be added to the OOA diagram (activity 1.2.4).

2.5.1.3 Identifying Subjects (Activity 1.3)

Selecting possible Subjects (activity 1.3.1) is done by promoting the uppermost class in each structure upwards to a Subject. Then promote each Class & Object not in a structure upwards to a Subject.

Refining Subjects (activity 1.3.2) is done by searching for minimal interdependencies and minimal interactions between Class & Objects in different Subjects. Next the Subjects can be constructed (activity 1.3.3) and added to the OOA diagram (activity 1.3.4).

2.5.1.4 Identifying Attributes (Activity 1.4)

To find attributes (activity 1.4.1) the next questions have to be asked from the perspective of an object:

- How am I described in general?
- How am I described in this problem domain?
- How am I described in the context of this system's responsibilities?
- What do I need to know?
- What state information do I have to remember over time?
- What states can I be in?

In defining Attributes the Gen-Spec structure has to be kept in mind, attributes have to be placed as high as possible in the Inheritance Structure (activity 1.4.2).

After identifying attributes the Instance Connections between objects can be identified (activity 1.4.3).

This is done by adding connection lines for each object reflecting mappings within the problem domain. The Gen-Spec structures have to be examined to put the Instance Connections at the right level.

The identified Attributes and Instance Connections have to be checked for several special cases (activity 1.4.4).

These cases are:

- Attributes with a 'non-applicable' value
- Class & Objects with only one Attribute
- Attributes with repeating values
- Many-to-many Instance Connections
- Instance connections between Objects of a single Class
- Multiple Instance Connections between Objects
- Additional needed Instance Connections
- One connecting Object having a special meaning

After checking these cases the Attributes have to be specified, and if needed additional constraints have to be specified for an Attribute and the Attributes and Instance Connections can be added to the OOA diagram (activity 1.4.5).

2.5.1.5 Identifying Services (Activity 1.5)

The first step in defining Services is identifying the Object States (activity 1.5.1) by describing the States and Transitions in an Object State Diagram.

Then the required Services can be identified for each Class & Object (activity 1.5.2). For the algorithmically complex Services like Calculate and Monitor a

Service Chart can be drawn to depict the algorithmical behavior of the Service. If needed Service/State Tables can depict the connection between Services and States.

Simple Services are not denoted in the OOA diagram. After identifying Services Message Connections can be identified (activity 1.5.3). For identifying the Message Connections the next questions have to be asked for each object:

- What other Objects do I need services from?
- What other Objects needs Services from me?

Then follow each Message Connection and repeat these questions. Next the Services can be specified and Services and Message Connections can be added to the OOA diagram (activity 1.5.4).

2.5.1.6 Prepare Documentation (Activity 1.6)

The last step of the OOA part of the method is putting the OOA documentation together.

This activity consists of:

- Drawing of the complete OOA diagram (activity 1.6.1)
- Specification of Class & Objects (activity 1.6.2)
- Addition of supplemental documentation if this is needed (activity 1.6.3) which can consist of:
    - Table of critical threads of execution
    - Additional system constraints
    - Services/State tables or diagrams

2.5.2 Object Oriented Design (Activity 2)

2.5.2.1 Designing the Problem Domain Component (Activity 2.1)

By designing the problem domain component the Object Oriented Analysis is carried into Object Oriented Design. The OOA results are improved, and additions to it are made during designing the problem domain component.

The first step is searching for previous Designs and Classes that can be reused (activity 2.1.1).

To use so called off-the-shelf Classes there may be the need for some changes in the OOA results. If possible the changes have to be kept small.

Next Problem Domain specific Classes can be grouped together (activity 2.1.2) by adding a Root Class as mechanism for grouping Classes. Also Generalization Classes have to be added to establish a protocol, the naming of a common set of Services (activity 2.1.3).

With regard to the programming language that will be used for implementation it can be necessary to change the OOA Gen-Spec structures to accommodate the level of inheritance the programming language offers (activity 2.1.4). Coad & Yourdon give guidelines for moving a multiple Inheritance structure into single Inheritance structures, and how to flatten single Inheritance structures in zero-Inheritance, although this is not recommended, because of the possibilities inheritance offers.

The next step is changing the Problem Domain Component to improve performance aspects (activity 2.1.5). Speed can be improved by measuring the speed of actual code, the perceived speed of the system can be approved by caching interim results.

Support of the Data Management Component (activity 2.1.6) can be achieved in several ways: each Object can store itself, or it can be saved by the Data Management Component. For both approaches specific Attributes and Services (or new Classes) have to be added to the Problem Domain Component. A third approach is the use of an Object Oriented Database Management System (OODBMS) which takes care of storing Objects.

For convenience during design and programming, lower-level components can be isolated in separate Classes (activity 2.1.7). The last step of designing the Problem Domain Component is reviewing and challenging the additions that have been made to the OOA results (activity 2.1.8).

2.5.2.2 Designing the Human Interaction Component (HIC) (Activity 2.2)

For the Human Interaction Component prototyping is recommended.

Designing the Human Interaction Component starts with classifying the people who will be the users of the system by 'putting yourself in their shoes' (activity 2.2.1).

Classification can be done on the next criteria:

- Skill level
- Organizational level
- Membership in different groups (for example staff or customer)

Each defined category of people has to be described, including a task scenario (activity 2.2.2). The next things have to be written down for each:

- Who is it?
- Purpose
- Characteristics (like age, education etc.)

- Critical success factors (needs/wants and likes/dislikes/biases)
- Skill level
- Task Scenarios

After doing this a command hierarchy can be designed for the system (activity 2.2.3) by studying existing human interaction systems.

To refine this hierarchy the next guidelines have to be considered:

- Ordering of services
- Chunking whole-part patterns and breadth/depth guidelines
- Minimizing number of steps to get a service done

After this is done the detailed interaction can be designed (activity 2.2.4).

Criteria to use are:

- Consistency in human interaction
- Minimizing of number of steps
- Giving meaningful timely feedback to users
- Use small steps to accomplish something
- Provide 'Undo' functions
- Don't rely on human memory storage for remembering things
- Keep learning time and effort small
- Pleasure and appeal of system to users is important

To check if the Human Interaction Component meets this criteria there is a need for Prototyping (activity 2.2.5). After the detailed interaction is found to serve the needs, the human interaction Classes have to be designed (activity 2.2.6), including windows, fields, graphics and selectors. Whenever possible, standard graphical user interfaces (GUI) should to be used.

2.5.2.3 Designing the Task Management Component (Activity 2.3)

The first activity is looking if there is a need for tasks in the system (activity 2.3.1).

The next kinds of systems demand a need for tasks:

- Systems with data acquisition and control responsibility for local devices
- Human interaction with multiple windows that run simultaneously
- Multi-user systems
- Multi-subsystem software architectures
- Single processor systems needing tasks to communicate and coordinate
- Multiprocessor systems
- Systems needing communication with other systems

If not needed, tasks should not be designed, because they increase complexity.

There are several kinds of tasks that can be identified (activity 2.3.2):

- Event-Driven Tasks that trigger upon an event (for example a mouse-click)
- Clock-Driven Tasks that are triggered on a specified time interval
- Priority tasks and Critical Tasks

When three or more tasks are needed it is recommended to add a special coordination task.

After defining the tasks these must be challenged against a number of criteria (activity 2.3.3) to keep the number of used tasks at a minimum and to keep the tasks understandable.

After this each task can be defined (activity 2.3.4) by specifying:

- What the task is
- How the task coordinates
- How the task communicates

2.5.2.4 Designing the Data Management Component (Activity 2.4)

First an approach for the Data Management Component has to be chosen (activity 2.4.1); a choice has to be made whether Flat Files, a Relational Database Management System or an Object Oriented Database Management System will be used.

According to this choice different aspects have to be considered like identification, normalization etc. For the chosen approach the possible Data Management Tools have to be assessed using a list of criteria (activity 2.4.2).

The last step is designing the Data Management Component for the chosen approach and tool(s) (activity 2.4.3). This consists of designing the data layout and designing the corresponding services. How this has to be done depends on the chosen approach, Flat File, RDBMS or OODBMS.

Coad and Yourdon give a number of guidelines for Object Oriented Design to measure the quality of the design. These guidelines include the next topics:

- Coupling
- Cohesion
- Reuse
- Additional criteria:
    - Clarity of design
    - Generalization-Specialization depth
    - Simplicity of Classes & Objects
    - Simplicity of protocols
    - Simplicity of services

- Volatility of the design
- System size

To evaluate the design the authors suggest Walk-throughs and evaluation of the design with Critical Success Factors.

CHAPTER THREE
THE DESIGNING OBJECT-ORIENTED SOFTWARE (DOOS) METHOD BY WIRFS-BROCK

3.1 Background

According to DOOS the main objective in the Object Oriented approach is to manage the complexity of the real world by using abstraction.

The knowledge of the real world is abstracted and encapsulated in objects.

The difference between the traditional and the OO approach is that OO talks about the what of the system, where the traditional approach talks about the how of the system.

DOOS starts with searching objects, and then describing the responsibilities of these objects. In this view the world can be seen and modeled as a system of collaborating objects. The software is seen as a living thing (anthropomorphic view).

When the objects are chosen and defined carefull these can be used again, so reuse of objects is one of the possibilities of the Object Oriented approach. Compared with the traditional software lifecycle the Object Oriented lifecycle dedicates more time to design (analysis and design) and less time to implementation and testing (when an Object Oriented programming language is used) according to Wirfs-Brock.

3.2 General Approach

DOOS is divided into two parts:

- An initial exploratory phase (Analysis in the software lifecycle): The main topic of the initial exploratory phase is determining the objects, responsibilities and collaborations of objects that play a role in the real world.

- A detailed analysis phase (First part of Design in the software lifecycle): In the detailed analysis phase the results from the first part are refined and streamlined. At this point a full specification can be made.

3.3 Concepts and Constructs

3.3.1 Concepts

- **Object**: An object is a thing that retains certain information and knows how to perform certain operations.
- **Class**: A class is a collection of objects which share the same behavior.
- **Abstract class**: An abstract class is a class that does not create instances (objects), it only specifies behavior (or a template for behavior) for its subclasses.
- **Concrete class**: A concrete class is a class that can be instantiated.
- **Responsibility**: The knowledge an object maintains and the actions an object can perform.
- **Private responsibility**: A responsibility that represents behavior that can be requested by other objects.
- **Public responsibility**: A responsibility that represents behavior that cannot be requested by other objects.
- **Collaboration**: An object collaborates with another object to fulfill it's own responsibilities.
- **Client-Server-Contract**: The collaboration between objects in DOOS is done in a client-server way: one object (the client) requests a responsibility from another object (the server). A contract is a cohesive set of responsibilities which a client can request. The contract must be fulfilled by the server.
- **Subsystem**: A set of classes (and possibly other subsystems) collaborating to fulfill one or more contracts. Conceptually a subsystem can be treated like a Class on a higher level of scale.

- **Method**: A step-by-step algorithm executed in response to receiving a message. A method is a private part of the object.
- **Signature**: The name of a method, the types of the parameters and the type of object that the method returns.
- **Protocol**: The signatures for the methods that each Class will implement.

Objects and classes are treated as the same throughout the DOOS method.

3.3.2 Relationships between Objects

Collaborations: An object collaborates with another object to fulfill it's own responsibilities.

3.3.3 Relationships between Classes

DOOS provides only one kind of relationships between Classes that is treated as a relationship.

3.3.3.1 "is-kind-of" Relationship (Inheritance Relationship)

The "is-kind-of" relationship describes a Super/Subclass relationship between Classes. DOOS provides both Single Inheritance (a subclass inherits from only one superclass) and Multiple Inheritance (a subclass inherits from more than one superclass).

The other mentioned kinds of relationships are only examined to find additional responsibilities and collaborations, and are not treated as real relationships in the DOOS method.

3.3.3.2 "is-analogous-to" Relationship

The "is-analogous-to" relationship is examined for finding missing superclasses, when two classes are analogous they might share common responsibilities

3.3.3.3 "is-part-of" Relationship (Whole-Part Relationship)

The "is-part-of" relationship is examined to determine where responsibilities should be placed, either at the Part or at the Whole Class.
This relationship is also examined to find collaborations; a 'whole' class may require collaboration(s) with its 'part' classes. This causes a distinction between:

- Composite classes and objects that compose them
- Container classes and the elements it contains

For the first type, collaborations between Whole and Part classes will be frequent, the second type may or may not require collaborations between Whole and Parts.

3.3.3.4 "has-knowledge-of" Relationship

With this relationship additional collaborations between classes are determined.

3.3.3.5 "depends-upon" Relationship

Classes that depend upon other classes require either a "has-knowledge-of" relationship with another class, or a third class is needed to form the connection. For example: in a drawing system with elements some elements can be behind or in front of others. It's clear that these elements depend upon the other elements. The choice can be made to give the elements knowledge of other elements, or to create an ElementsInfo class.

3.3.4 Operations and Communication

DOOS does not make difference between different types of operations. In the DOOS method communication between Objects is done by sending messages. One object sends a message to another object, the receiver receives the message, performs the requested operation and (possibly) returns some information.

3.4 Techniques

The techniques that are used in the DOOS method are:

- Class-Responsibility-Collaboration card (CRC card) on which a Class with its Super/Subclasses, Responsibilities and Collaborations is denoted.
- Subsystem cards on which Subsystems, Contracts and Delegations are denoted.
- Class Hierarchy Graphs to show the classes and their Inheritance hierarchies.
- Venn Diagrams to examine the chosen Inheritance hierarchies for classes.
- Collaborations graphs to show Classes, Subsystems and the Client-Server Collaborations between them. The clients and servers for contracts are denoted.

3.5 Analysis and Design Processes

There are a few remarks about the DOOS method: the results of applying this method are never final, so reiteration is an important part of the method, the guidelines that the method provides are not rigid guidelines and it is suggested that validation can be achieved at some points of the design by 'Walk-throughs'. A 'Walk-through' means that several valid and invalid execution paths from the real world should be walked-through in the design by the analyst/designer to see if the design meets the requirements.

3.5.1 The Initial Exploratory Phase

3.5.1.1 Identify Classes (Activity 1)

Identifying Classes starts with reading and understanding the requirements specification (activity 1.1). A first list of candidate classes is made by extracting the nouns and possibly hidden nouns from the specification (activity 1.2).

The following guidelines have to be applied to the found candidate classes (activity 1.3):

- Model physical objects
- Model conceptual entities
- Use a single term for each object
- Control if adjective-nouns objects differ from the noun objects
- Control sentences with missing/misleading subjects
- Model categories of objects (possible superclasses)
- Model interfaces to the system
- Model values of object's attributes (i.e. not height but integer)

After identifying Classes, candidate Abstract Superclasses can be identified and named (activity 1.4) by grouping classes with common attributes. To name these superclasses the next guidelines can be applied:

- Enumerate the shared attributes and derive the name from these
- Divide the elements in smaller, more clearly defined categories

If this still yields no name the group of Classes has to be discarded.

Next try to look for potential missing Classes (activity 1.5) by expanding categories of already identified Classes. It is stated that this is not an easy job, and that only experience makes it easier.

After these activities are performed every class has to be recorded on CRC cards, including a description of each Class (activity 1.6).

It is important to keep in mind that this first step is very preliminary and will have to be reiterated some times, the analyst shouldn't decide to throw a candidate class away to soon. It's better to maintain it and perhaps later decide to discard it.

3.5.1.2 Identify Responsibilities (Activity 2)

The process of finding responsibilities starts with looking at the requirements specification (activity 2.1).

Candidate Responsibilities are:

- Verbs extracted from the requirements specification
- Responsibilities extracted from the purpose of the candidate Classes
- Responsibilities extracted from a walk-through the system.

Next the candidate Responsibilities have to be assigned to Classes they belong to (activity 2.2). This can be done by applying the next guidelines:

- Evenly distribute system intelligence
- State responsibilities as generally as possible
- Keep behavior with related information; if an Object keeps information, it also has to perform the operations upon this information
- Keep information about one thing in one place. If this is not possible three solutions can be chosen from:
    - Create a new Object as a repository of the information
    - Reassign the responsibility to the Object whose principal responsibility is to maintain the information
    - Collapse the Objects needing the information into one Object
- Share compound responsibilities among related classes

If it's hard to decide to which Class a Responsibility has to be added, the different possibilities have to be taken into account and examined with a Walk-through. Then choose the most natural or most efficient way. It is also possible to let a problem domain expert do a Walk-Through.

For finding additional responsibilities (activity 2.3) three kinds of relationships are especially important to examine:

- 'is kind of' relationships to identify responsibilities shared by Subclasses
- 'is analogous to' relationships to identify missing Superclasses with its Responsibilities
- 'is part of' relationships to identify whether responsibilities belong to the whole Class or the Part class, and to find Responsibilities the Whole Class must have in respect to the Part Class (For example the Responsibility to know about its Parts).

After this is done the identified Responsibilities can be added to the CRC cards (activity 2.4).

3.5.1.3 Identify Collaborations (Activity 3)

Classes can fulfill Responsibilities either by performing the necessary computations themselves or by collaborating with other classes.

To find Collaborations (activity 3.1) ask the next questions for each Class:

- Is the Class capable of fulfilling this Responsibility itself? If not, what does it need and from what other class can it acquire what it needs?
- Who needs to use the Responsibilities of this Class?

For finding additional Collaborations (activity 3.2) the next relationships have to be examined:

- 'is part of' relationship. This can either be:
    - Composite Classes -> Collaboration between part and whole
    - Container-element Classes -> not always Collaborations between container and element
- 'has knowledge of' relationships. This can imply a Collaboration between the Class that has the knowledge and the other Class.
- 'depends upon' relationships. This can either be a 'has knowledge of' relationship, or a relationship with a third Class forming the connection.

The classes that do not collaborate with other classes and are not collaborated with, have to be discarded (activity 3.3). Next the identified Collaborations can be added to the CRC cards (activity 3.4).

To control if the chosen Collaborations are right a 'Walkthrough' is suggested (activity 3.5).

With this step the initial exploratory phase is ended, and the result is a preliminary design which has to be turned into a solid and reliable design in the following detailed analysis phase.

3.5.2 Detailed Analysis Phase

3.5.2.1 Identify Hierarchies (Activity 4)

- The first step for identifying Hierarchies is drawing Hierarchy Graphs (activity 4.1) to illustrate the inheritance relationships among Classes. At this stage it has to be identified whether classes are abstract or concrete (activity 4.2).
- Then Venn Diagrams can be drawn (activity 4.3) to show the shared responsibilities between Classes and Subclasses who inherit from the Superclasses. These Venn Diagrams show whether the Hierarchy is well chosen or not.

- To construct a good Class Hierarchy (activity 4.4) the next guidelines have to be applied:
- Model a 'kind-of' hierarchy. Subclasses have to be a 'kind-of' the Superclass and have to inherit all Responsibilities from its Superclasses.
- Common Responsibilities have to be modeled as high as possible in the Class Hierarchy. This may require the creation of new (additional) Superclasses.
- Abstract classes are not allowed to inherit from concrete Classes.
- Classes who do not add functionality have to be discarded.

After the Class Hierarchy is built Contracts can be identified (activity 4.5) using the following guidelines:

- Group Responsibilities used by the same clients to get a cohesive set of Responsibilities.
- Maximize the cohesiveness of Classes; a Class should support a cohesive set of Contracts.
- Minimize the number of Contracts in the design. This can be done by moving similar Responsibilities that can be generalized higher in the Class Hierarchy.

A way to apply these guidelines is to start with defining Contracts for Classes at the top of the Class Hierarchy and then go down in the Hierarchy to Subclasses. The contracts have to be noted on the CRC cards where every Contract gets a unique number, and Contract cards have to be written (activity 4.6).

3.5.2.2 Identify Subsystems (Activity 5)

Identifying Subsystems starts with drawing a complete Collaborations Graph (activity 5.1).

Subsystems can be identified by looking for frequent and complex collaborations between strongly coupled classes (activity 5.2).

The next guidelines have to be applied for identifying Subsystems:

- Classes in a Subsystem should collaborate to support a small and cohesive set of Responsibilities
- Classes within a Subsystem should be strongly interdependent

After identifying the Subsystems the patterns of Collaborations have to be simplified (activity 5.3) applying the next guidelines:

- Minimize the number of Collaborations a Class has with other Classes or Subsystems
- Minimize the number of Classes and Subsystems to which a Subsystem delegates
- Minimize the number of different Contracts supported by a Class or Subsystem

The Subsystems are denoted on Subsystem cards. The CRC cards have to be modified because Collaborations between Classes have become Collaborations between Classes and Subsystems. The changes in Classes, Collaborations and Contracts have to be recorded both in the Hierarchy Graphs and on the CRC cards. At this stage it is again suggested to perform a 'Walk-through' (activity 5.4) to check the design.

3.5.2.3 Identify Protocols (Activity 6)

The last step of the DOOS method starts with designing the protocols for each class (activity 6.1). Constructing Protocols is done by a refining the Responsibilities into sets of Signatures that maximize the usefulness of classes.

The next guidelines have to be used for constructing the protocols:

- Use a single name for each conceptual operation.

- Associate a single conceptual operation with each method name.
- Make it explicit in the Inheritance Hierarchy if Classes fulfill the same specific Responsibility.
- Protocols have to be generally useful.
- Provide default values for as many parameters as reasonable.

When this is done a design specification has to be written for each Class (activity 6.2), each Subsystem (activity 6.3) and each Contract (activity 6.4).

CHAPTER FOUR
THE OBJECT MODELLING TECHNIQUE (OMT) BY RUMBAUGH

4.1 Background

Rumbaugh et al. start their book with discussing the main difference between the traditional approach in software development and the Object Oriented approach. The main difference is the fact that the Object Oriented approach is not based on functional decomposition but on describing the real objects that play a role in the real world.

OMT describes Object Oriented as a way to organize software as a collection of discrete objects that incorporate both data-structure and behavior. The characteristics OMT finds essential for an approach to be Object Oriented are identity, classification (the grouping of Objects in Classes), polymorphism and inheritance, but they remark that there is some dispute about these characteristics.

The essence of Object Oriented development in the Object Modeling Technique (OMT) is the identification and organization of application-domain concepts, instead of implementation domain concepts.

According to Rumbaugh et al. the benefits of an Object Oriented approach comes from a greater emphasis on essential properties of an Object which forces the developer to think more carefully and deeply about what an Object is and does. They note that the main cost benefit is not found in a reduced development time but in future reuse of Classes/Designs and reducing errors and maintenance efforts.

For Rumbaugh et al. the main impacts of the Object Oriented approach are:

- Shifting of development effort into analysis
- Emphasis on data structure before functions. This provides stability
- A seamless development process

- An iterative rather than sequential approach. Features are added and clarified in iterations.

The OMT method has very rich notations, for the development of most systems only two/third of the possible notations will be used, but some systems need the more advanced modeling

4.2 General Approach

The OMT method describes Analysis as well as Design and Implementation. We will only focus at the Analysis and Design part.

OMT divides Analysis and Design in three parts:

- Analysis: Building of a model of the real-world situation starting with a problem statement.
- System Design: Design of the overall architecture of the system.
- Object Design: Refinement of the object structure towards efficient implementation, implementation details are added to the objects.

OMT mentions that the described method is not a sequential one, but iteration is needed to add and clarify features in a proper way.

In the OMT approach the system is described from three different views: an Object view, a Dynamic view and a Functional view. The object model describes the object to which something happens (the information flow), the functional model describes what happens (the process computations) and the dynamic model describes when it happens (the control flow).

4.3 Concepts and Constructs

4.3.1 Concepts

- **Object**: Concept, abstraction or thing in the world with crisp boundaries meaningful for the problem at hand to promote understanding of the real world and provide an asis for computer implementation. Objects have state and identity.
- **(Object) Class**: Group of objects with similar properties (attributes), behavior (operations), common relationships to other objects and common semantics.
- **Abstract Class**: An abstract class is a class which cannot be instantiated. From this it results that it cannot be a leaf Class in the inheritance tree structure.
- **Concrete Class**: A class that can have instances.
- Inheritance: The sharing of attributes and operations among classes based on a hierarchical relationship.
- **Delegation**: An implementation mechanism in which an Object, responding to an Operation on itself, forwards the Operation to another Object.
- **Attribute**: A property of a Class describing a data-value held by each object of the class.
- **Derived attribute**: An attribute that can be computed from other attributes
- **Operation**: A function transformation that may be applied to or by objects in a class.
- **Method**: A specific implementation of an operation by a certain class.
- **Signature**: The number and types of the arguments and the type of result for an Operation
- **Link**: A physical or conceptual connection between object instances.
- **Association**: A group of links with common structure and common semantics.
- **Derived association**: An association that is described in terms of other associations.

- **Role**: One part of an association
- **Module**: A module is a logical construct for grouping classes, associations and generalizations to capture one perspective or view of a situation. The boundaries of a module are somewhat arbitrary.
- **Subsystem**: A coherent subset of a system containing a tightly bound group of Classes and their Relationships. Thread of control A thread of control is a path through a set of state diagrams on which only a single object at a time is active.
- **State**: The values of the attributes and links of an object at a particular time.
- **Event**: Something that happens at a point in time.
- **Event scenario**: Sequence of events occurring during one particular execution of a system. (A scenario can be depicted as an event trace).
- **Process**: Something that transforms data values.
- **Data Flow**: A connection from output of an object or process to input from another object or process.
- **Actor**: An active object that drives the data flow graph by producing or consuming data values.
- **Data store**: Passive object that stores data for later access.
- **Action**: Operation with side effects on objects, which has no duration in time.
- **Activity**: Operation with side effects on objects, which has a duration in time.

4.3.2 Object Modeling Concepts

4.3.2.1 Relationships between Classes/Objects

Associations : There are associations between classes which consists of links. A link is for example Joe Smith works-at IBM.An association is for example person works-at company.Associations are always bi-directional and there can be rolenames given for each direction. Also cardinality (OMT calls this multiplicity) can be denoted as well as a specific ordering of objects of a Class.

It is possible to add an attribute to a link. (For example: Person works-at Company for Project). There are some special cases of associations:

Qualification association : This relates two objects and a qualifier (a special attribute that reduces the effective multiplicity of an association).

For example: A directory has many files; a file belongs to one directory. A filename specifies a unique file. A directory plus a filename yields a file and a file is a directory plus a filename. The association between directory and file which was one-to-many is by using the qualifier filename changed in one-to-one.

Aggregation (Whole-Part relationship) : This association relates an assembly class to one component class. For example: Engine is-part-of Car. The aggregation relationship is transitive and it is possible to use the aggregation relationship recursively.

Generalization relationship (Inheritance) : Generalization is a relationship between a class (superclass) and one or more refined versions of it (subclass) who share attributes and operations. These are denoted in the superclass from which the subclasses inherits.

A subclass may override features defined in a superclass like methods of operations and values of attributes.

In this way the subclass is an extension and a restriction of the superclass. A class can inherit from one other class (single inheritance) as well as from several other classes (multiple inheritance). It is mentioned that this can cause conflicts among attributes or methods with the same names that are defined in the different superclasses.

In the Analysis phase of OMT overlapping inheritance (Vehicle with subclasses land vehicle and water vehicle --> overlap for amphibian vehicle) is possible and

explicitly as overlapping denoted, it is described how this in the Design phase can be refined in a well structured inheritance tree without overlapping features.

Metadata :

- Metaclasses

  When a class is treated as an object OMT calls it a metaclass. This is meaningful when there are attributes (class attributes) who are common to an entire class of objects. Also class operations can be described; operations on the class itself (For example operations to create class instances).

In OMT it is possible to describe operations that are propagated from a starting object to objects lower in a generalization or aggregation structure. For example: the operation copy on the object document can be propagated to a copy operation on paragraph and from there to a copy operation on a character object.

OMT offers the possibility to denote candidate keys for an object and it's also possible to add constraints to objects, classes, attributes, links and associations. A constraint is for example: EmployeeSalary < BossSalary. A rich set of constraints can be denoted including subset constraints.

Because real-world concepts are highly redundant OMT offers the possibility to explicitly denote derived objects, links and attributes. (A derived attribute for example is age, which can be determined from birth-date and current-date). In the subsequent Design phase these derived features can be re-examined.

For complex applications it is possible to denote relationships between relationships, called homomorphisms.

4.3.3 Dynamic Modeling Concepts

The dynamic part of the system is captured in State Diagrams. For every class with important dynamic behavior a State Diagram has to be drawn.

A State Diagram consists of States and Events. A change of State caused by an Event is called a Transition.

For the State Diagrams the next concepts are important:

- Entry/Exit Actions : Entry/exit actions are actions that will take place when a State is entered/left.
- Guarded transition (Guard on Event) : A guard is a Boolean expression to denote a condition under which a transition will take place.
- Action on a transition : For a state diagram it is important to show not only the event that takes place, but also the action that results from an event. (OMT distinguishes between action and activity, an activity is an operation that takes time to complete where an action is an instantaneous operation).
- Output event on a transition : A transition can cause an event to take place.
- Internal actions : An event doesn't always have to cause an action which causes a state change. In this case there are internal actions. (This is not the same as a transition from the State to the State itself, in that case the defined Entry/Exit Action will take place).

For an Event an Attribute can also be denoted.

As well as Object Diagrams, State Diagrams also provide aggregation and generalization with Inheritance. States can be decomposed in Sub State Diagrams, for which the transitions of the Superstate are inherited by each of it's Substates.

It's also provided to denote an Event hierarchy to generalize/specialize Events. Sub-events inherit attributes from Super-events.

In Inheritance relationships the state diagram of a subclass should usually be an independent orthogonal concurrent addition to the state diagram inherited from it's superclass (this means disjoint attributes), because refinement of the State Diagram of the Superclass causes ambiguities.

4.3.4 Functional Modeling Concepts

The computations in the system are captured in functional models. These models describe how output values in a computation are derived from input values.

These functional models are denoted as Data Flow Diagrams (DFD's), with the next concepts:

- Dataflow diagrams hierarchically describe processes; the diagrams can be nested. The nesting terminates with simple functions at the lowest level.
- Recursion in DFD's is possible, a Process can call itself.

Because it's sometimes useful to describe some control flows with decision functions in a DFD, this can be done by adding control flows. These control flows have to be duplicates from the dynamic model.

4.3.5 Relationships between Modeling Techniques

The object model shows the structure of actors, data stores and flows in the functional model, the dynamic model shows the sequence in which processes are performed and the functional model shows supplier-client relationship among classes.

4.4 Techniques

OMT uses mainly three techniques:

- Object models to describe the static structure of objects in a system and their relationships. The object model consists of object diagrams, a certain kind of EER models.

- Dynamic models to describe the control aspects of a system. This dynamic part for every class is described in State Diagrams. To identify Events, States and Transitions first Event Flow Diagrams are drawn.
- Functional models to describe the computations within a system. This is done with Data Flow Diagrams.

4.5 Analysis And Design Processes

4.5.1 Construct Analysis Models (Activity 1)

4.5.1.1 Write a Problem Statement (Activity 1.1)

The construction of the Analysis models starts with writing a Problem Statement for the investigated problem domain.

4.5.1.2 Build Object Model (Activity 1.2)

Building an Object Model starts with identifying Objects and Classes (activity 1.2.1). Candidate Objects and Classes can usually be found by looking at the nouns in the problem statement; they can be physical entities as well as concepts. The correct Classes can be detected by following the next guidelines:

- Discard redundant Classes
- Discard irrelevant Classes
- Discard vague Classes
- Do not model Attributes as Classes
- Do not model Operations as Classes
- Do not model implementation constructs
- Do not model roles that objects play

The identified Classes are placed in a Data Dictionary (activity 1.2.2), in which the Classes are described including a description for every Class. After identifying

Objects and Classes the necessary Associations between Classes can be identified (activity 1.2.3). Associations can often be found by looking at the verbs or verb phrases in the problem statement. Verbs to look at are verbs that describe:

- Physical location
- Directed action
- Communication
- Ownership
- Satisfaction of some condition (For example 'works for')

For keeping the correct associations the next guidelines have to be applied:

- Discard associations between Classes that are eliminated
- Discard implementation associations
- Don't model actions
- Decompose ternary associations if possible into binary associations
- Don't model associations that can be derived

Next the identified associations have to be refined applying the next guidelines:

- Find a good name for every association
- Add rolenames to associations if this is necessary
- Use qualified associations whenever possible
- Specify multiplicity of associations
- Find missing associations

The next step is identifying Attributes (activity 1.2.4). Attributes can usually be found in nouns with possessive phrases and adjectives. For derived Attributes it's important to either label them as derived, or discard them. Link Attributes also have to be identified. Guidelines for identifying Attributes:

- Don't model Objects as Attributes

- Use qualifiers wherever possible (names can often be modeled as qualifier)
- Don't model identifier attributes that are made to identify (For example 'Person ID')
- Model link Attributes
- Don't model internal values (states) of an Object
- Don't model fine details
- Attributes that are very different from other Attributes of an Object often indicate the need for adding a new Class

Object Classes can be refined by using Inheritance (activity 1.2.5) to organize the Class Hierarchy. Common aspects of Classes like Attributes, Operations or Associations can be generalized into Superclasses and existing Classes can be specialized into Subclasses. Specialization can be found in noun phrases with adjectives. Multiple Inheritance is also possible but should only be used if it is really necessary.

To control the adequateness of the Object Model it is recommended to test several access paths through the Object model (activity 1.2.6) to find missing and unnecessary Classes, missing and unnecessary Associations and incorrect placement of Associations and Attributes.

The last step for building the Object model is the grouping of sets of Classes that capture a logical subset of the model into Modules (activity 1.2.7) For grouping Classes the analyst should look at cut points between Classes. The number of bridges between Classes in different Modules should be minimized. For grouping Classes into Modules it is possible to look at earlier made Modules, it may be possible to reuse previous made Modules.

4.5.1.3 Build Dynamic Model (Activity 1.3)

Building the dynamic model of the system starts with preparing Scenarios of typical dialogues in the system (activity 1.3.1). A scenario consists of sequences of

Events, the actors that cause the Events and the parameters of the Events.

First scenarios for normal cases have to be prepared, followed by scenarios for special cases as omitted input sequences, maximum and minimum values, repeated values and error conditions. These scenarios can be denoted as Event Traces.

From the identified scenarios the Events that play a role can be identified (activity 1.3.2). Events can be one of the following: signals, inputs, decisions, interrupts, transitions and actions to or from users or devices.

Events with the same effects on the flow of control must be grouped together with one name, also when the parameter values of the Events are different. Next each event must be allocated to the Object Classes that send and receives it. (Sender and Receiver can be the same Object). The events between Object Classes can be drawn on an Event Flow Diagram (activity 1.3.3). Next for each Class with nontrivial dynamic behavior a State Diagram has to be made (activity 1.3.4) to show the Events that these Objects send and receive. First the State Diagrams for the normal Events of an Object have to be drawn, then the boundary and special cases can be added. Normally a flat State Diagram will do, but if necessary the State Diagrams can be nested recursively.

The completeness and consistency of the State Diagrams can be checked by matching Events between Objects (activity 1.3.5). Every Event must have a sender and receiver, and States must have predecessors and successors, unless they are starting or termination States.

4.5.1.4 Build Functional Model (Activity 1.4)

Building the functional model of the system starts with listing the input and output values of the system (activity 1.4.1), which are parameters of Events between the system and the outside world.

To show how each output value is computed from input values a DataFlow Diagram (DFD) is constructed (activity 1.4.2). This DFD can be nested to show different levels of detail. On a DFD Processes, Actors, External Stores and Data flows between these are shown. After drawing a fully refined DFD model the Functions that play a role in the system can be identified from the DFD and be described (activity 1.4.3). These description can be declarative or procedural, however the first is recommended.

At this point it is possible to identify possible constraints between objects. These constraints can be:

- Constraints between two Objects at the same time
- Constraints between Objects of the same Class at different times (invariant)
- Constraints between Objects of different Class at different times (usually described as input-output functions)
- Preconditions
- Postconditions

The last step in building the functional model is the specification of optimization criteria (activity 1.4.5). These criteria can be values to be minimized, maximized or optimized in another way.

4.5.1.5 Refine and Document the Three Models (Activity 1.5)

The first refinement of the models is the addition of Operations to the Object model (Activity 1.5.1).

Operations can be found by looking at:

- Events
- State Actions and Activities
- Functions from the DataFlow Diagram

- Additional operations from the 'Real-World'

Simple operations like Reading and Writing are not shown on the Object model. The identified Operations can be simplified by using Inheritance and placement of the Operations at the correct level in the Class Hierarchy. After doing this, an iterative step is necessary to refine all Analysis models. When the Analysis is complete the models have to be validated against the initial requirements. This can be done by restating the requirements from the models and look if they are consistent with the initial requirements. A problem domain expert should also be asked to verify the Analysis models.

4.5.2 Construct System Design (Activity 2)

4.5.2.1 Organize the System into Subsystems (Activity 2.1)

The first design step is dividing the system into a smaller number of components, named subsystems. A subsystem is a package of Classes, Events and Constraints, for which it is important to have a small interface with other Subsystems. Subsystems can be identified by looking at Services, groups of related functions that share a common purpose (For example arithmetics or I/O). For each Subsystem a well-defined interface to other Subsystems has to be established, which provides Designing different Subsystems independently. If necessary a Subsystem can be decomposed into other Subsystems, and on the lowest level into Modules.

The Decomposition of a Subsystem can be done in two ways:

- Layers: Each layer is built in terms of the one below it and provides as basis for the layers above it. (For example BitmapGraphics and WindowGraphics)
- Partitions: A vertical decomposition of a system into weakly coupled Subsystems providing several kinds of services. (For example Screen Dialogue, Arithmetics)

Often a systems decomposition is a mixture of Layers and Partitions. After identifying the Subsystems the information flow between the different Subsystems must be made explicit in a Data Flow Diagram (a different Data Flow Diagram than the one that captures the information flow between Objects).

4.5.2.2 Identify Concurrency in the Problem (Activity 2.2)

First the inherent concurrency in the system must be identified (activity 2.2.1). This can be done by examining the State Diagrams. It's also important to identify independent subsystems; these Subsystems can be assigned to different hardware units if possible and necessary. Sometimes the problem domain demands the need for independent Subsystems for example in the case of different physical location of the Subsystems.

To define concurrent Tasks (activity 2.2.2) the different possible threads of control in the system have to be examined. Often it's possible to merge several threads of control into one thread of control.

4.5.2.3 Allocate Subsystems to Processors and Tasks (Activity 2.3)

Allocating Subsystems to Processors starts with estimating the needed hardware resources (activity 2.3.1) which can be done in several ways. The designer must also decide which Subsystems should be implemented in hardware and which in Software (activity 2.3.2). After these decisions have been made the identified Tasks can be allocated to different Processors (activity 2.3.3).

Reasons to allocate Tasks to different Processors can be:

- Tasks are required at different physical locations.
- Improvement of system speed
- Distribution of system load between different Processors

After allocating Tasks to Processors the Physical connectivity between the different physical units can be established (activity 2.3.4) which include:

- Connectivity topology
- Topology of repeated equal units
- Connection channels and communication protocols

4.5.2.4 Choose Management of Data Stores (Activity 2.4)

At this point in the design decisions have to be made concerning the data stores. It has to be decided whether to use files or a Database Management System.

4.5.2.5 Choose Access to Global Resources (Activity 2.5)

The use of global resources like physical units, logical names and shared data must be identified and access to it must be defined explicitly. (For example whether to lock parts of a Database).

4.5.2.6 Choose Implementation of Control (Activity 2.6)

Control can be implemented in the software in several different ways. External control, the flow of external visible events among Objects in the system can be handled in three ways:

- Procedure-driven systems (like OO programming languages)
- Event-driven systems (like X-Windows)
- Concurrent systems (like concurrent OO programming languages)

Internal control, the flow of control within a process can be seen as procedure calls in a programming language. (The mentioned systems are not the only possibilities, also rule-based systems or other nonprocedural systems can be used).

4.5.2.7 Handle Boundary Conditions (Activity 2.7)

It is also important to describe the different boundary conditions, including:

- Initialization of the system
- Termination of the system
- Failure of the system

4.5.2.8 Set Trade-off Priorities (Activity 2.8)

At this point it is possible to make several decisions between possibilities, because not all goals for the system can be achieved. (For example memory use vs. response time etc.). The last part of the system design describes some common architectural frameworks which can be used during system design. These frameworks are for the next kinds of systems:

- Batch transformation
- Continuous transformation
- Interactive interfaces
- Real time systems
- Transaction manager systems.

4.5.3 Object Design (Activity 3)

4.5.3.1 Combine the Three Models (Activity 3.1)

The three models have to be combined to obtain Operations for the Object model. Operations must be defined for:

- Each Process in the functional model
- Each Event in the dynamic model

4.5.3.2 Design Algorithms for Operations (Activity 3.2)

The first step in designing algorithms is to choose algorithms for Operations that seem to be nontrivial. Algorithms must be defined to implement functions for which only a declarative description is given and to optimize functions. (For example an algorithm for a more sophisticated way to do sorting). To choose the best algorithm examine the:

- Computer complexity of the algorithm
- Ease of implementation and the understandibility of the algorithm
- Flexibility of the algorithm
- Fine tuning of the Object model (if the algorithm 'fits' in the Object model)

The next step is choosing suitable data structures (activity 3.2.2) like arrays, which will be used for implementation. It is often necessary to define additional internal Classes and Operations (activity 3.2.3) to implement higher level Operations with lower level Operations. This can also involve the creation of new Classes which consist of implementation details that are totally internal for the system.

For several Operations (mostly for Operations on internal Classes) it is necessary to assign these Operations to the right Classes. This needs implementation knowledge, because the involved Classes don't come from the real-world problem domain.

Placing the Operations at the right level can be done using the next guidelines:

- Associate the Operation rather with the target Object of the Operation than with the initiator
- Associate the Operation with the Object that is modified by the Operation
- Associate the Operation with the most centrally-located Object from the involved Objects
- Project the question to a real-world situation

4.5.3.3 Optimize Access Paths (Activity 3.3)

To make the implementation more efficient several optimizations can be used. The first is adding redundant Associations (activity 3.3.1) to get efficient access to Objects and to improve the system's response time. For this optimization the next guidelines can be applied:

- Examine each Operation and look at the Associations to traverse
- Look at the number of times an Operation is called and what this costs
- Estimate the cost of traversing a path
- Look at the number of Objects that meet specified selection criteria

Another improvement can be reached by rearranging the execution order of Operations in an algorithm (activity 3.3.2). Saving derived Attributes (activity 3.3.3) to avoid recomputation of often used data also improves the efficiency of the system. However, this requires the temporary update of the derived Attributes. This can be done in several ways:

- Explicit update
- Periodical recomputation
- Active values to register dependencies upon other values

4.5.3.4 Implementation of Control (Activity 3.4)

The State Diagrams now have to be turned toward an implementation, using the strategy chosen in the system design (see activity 2.6).

4.5.3.5 Adjust Class Structure to Increase Inheritance (Activity 3.5)

Operations on different Classes are often quite similar. These Operations and Classes can often be rearranged (activity 3.5.1) to decrease the overall number of

Operations and to increase the use of Inheritance, but the Operations that can be shared must have similar interface (signature) and semantics.

The signature of the Operations can often be adjusted to each other using the next guidelines:

- Missing arguments can be added to the signature and further ignored
- Operations that are specified versions of generalized Operations can call the general Operation
- Attributes in Operations can have a different name, but the same meaning
- Similar Operations can have different names
- Operations can be defined as No-Operations on Subclasses that don't use them

Once again the Object model can be examined to abstract out common behavior of different Classes which was yet unidentified (activity 3.5.2), involving the creation of new abstract superclasses.

Another possibility is to use delegation to share implementation (activity 3.5.3) as a kind of Inheritance of implementation for relationships who do not really model a 'is-a' relationship. It's for example better to build a stack by delegating Operations to a list than specializing a list to a stack.

4.5.3.6 Design Associations (Activity 3.6)

To design Associations in a proper way it has to be analyzed first how the Associations are used (activity 3.6.1). Then the implementation decisions can be made for the Associations (activity 3.6.2).

4.5.3.7 Determine Attribute Representation (Activity 3.7)

Determining the representation of Attributes involves the decision whether to add Classes for capturing Attributes or model them directly as primitive data-types.

4.5.3.8 Package Classes and Associations into Modules (Activity 3.8)

The packaging of Classes and Associations into Modules depends on the used Implementation language, the wanted degree of information hiding and the degree of coherence between entities.

For defining the information hiding aspect of Operations the next guidelines should be applied:

- Allocate to each Class the responsibility of performing Operations and providing information that pertains to it
- Use Operations to access Attributes belonging to Objects of another Class
- Avoid traversing Associations that are not connected to the current Class
- Define interfaces at a level of abstraction as high as possible
- Hide external Objects at the system boundary by defining abstract interface Classes
- Avoid applying a method to the result of another method

CHAPTER FIVE
OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS (OOADA) BY BOOCH

5.1 Background

Booch's book starts with an extensive discussion of the concepts of the Object-oriented approach and a discussion of complexity and the attributes of a well-structured complex system to manage this complexity. Model building is very important for the constructing of complex systems. Booch proposes four models of object-oriented development: a logical and physical structure and its static and dynamic semantics.

Next, he discusses the Object model, and shows that with the Object model systems are derived that embody the attributes of well structured complex systems. The hardest part of OO analysis and design according to Booch, is the identification of classes and objects. Proper classification thus is a fundamental issue in OO analysis and design; it deals with the problem of clustering and involves both discovery and invention.

5.2 General Approach

The OOADA method distinguishes between the logical and physical structure of a system and describes for both the static and dynamic semantics.

The method is not a sequential one like the waterfall model, the development of the design is evolutionary, it's an incremental and iterative method. This is called 'Round-Trip Gestalt Design': incremental and iterative development through refinement of logical and physical views of the system as a whole.

OOADA is divided into a micro and a macro process. The micro process basically represents the daily activities of the developer(s) and consists of four (non-sequential) major steps:

- Identifying classes and objects at a certain level of abstraction
- Identifying the semantics of the objects and classes
- Identifying the relationships among classes and objects
- Implementation of the classes and objects

The macro process is used for controlling the micro process. It addresses activities for the whole development team on the scale of months or weeks at a time. It discusses five activities:

- Conceptualization, in which the core requirements are established
- Analysis, in which a model of the desired behavior is developed
- Design, in which an architecture is created
- Evolution, in which the implementation is evolved
- Maintenance, in which postdelivery evolution is managed

Because OOADA addresses the total Design as well as Implementation the method describes the topics of visibility of objects to each other and synchronization of communication between objects including concurrency aspects.

5.2.1 Changes to the First Edition

The title of the book has been changed from 'Object-oriented Design with Applications' into 'Object-oriented Analysis and Design with Applications'. The development process and pragmatics have been greatly expanded.

In the notation, certain semantics were added and clarified. At some points, the notation has been simplified. An important aspect on notation is that a new unified notation of concepts was introduced for the benefit of standardization.

5.3 Concepts and Constructs

5.3.1 Concepts

- **Object**: An Object has state, behavior and identity.
- **Class**: A set of Objects that share a common structure and common behavior.
- **Abstract class**: A class with no instances.
- **Base class**: The most generalized Class in a structure.
- **Class category**: A collection of Classes (grouping of Classes).
- **Class utility**: A collection of free subprograms; i.e. procedures or functions that serve as nonprimitive operations upon one or more objects, belonging to the same or different classes.
- **Metaclass**: A metaclass is the class of a class; in other words, the class is treated as an object and thus as an instance of another class.
- **Note**: Textual description of assumptions and decisions. A note may be applied to any element of any diagram.
- **Friend**: A class or operation whose implementation may reference the private parts of another class.
- **Field**: A repository for part of the state of an Object.
- **Active/passive object**: Active Objects are Objects that encompass their own thread of control, passive Objects can only undergo a State change when they are explicitly acted upon. Because active Objects are autonomous, meaning that they can exhibit behavior without being operated upon by another Object they are the root of control in the system. Multiple threads of control in a system (concurrency) involves multiple active Objects, where sequential systems have only one active Object at a time.
- **Persistence**: The property of an Object through which its existence transcends time and/or address space. An object can have static persistence or dynamic persistence. An Object can thus exist after termination of a program.
- **State**: Static properties and the current (usually dynamic) values of each of these properties. A state can either be a start state, an intermediate state or a stop state.

- **History**: Indicator for the most recently visited state.
- **Transition**: A change of state.
- **Event**: An occurrence that may lead to a Transition.
- **Action**: Something that is performed when a transition occurs.
- **Method**: An operation upon an Object, defined as part of the declaration of a Class.
- **Operation**: An action an Object performs upon another Object to obtain a certain reaction. All Methods are Operations, but an Operation can either be a Method or a free subprogram (Class utility).
- **Message**: An operation that one Object performs upon another.
- **Protocol**: A specification of the how an Object may act and react; it deals with the entire static and dynamic outside view of an Object.
- **Module**: A unit of code that serves as a building block for the physical structure of a system. A module consists of an interface part and its implementation.
- **Subsystem**: A collection of Modules.
- **Process**: The activation of a single thread of control.
- **Processor**: Piece of hardware capable of executing programs.
- **Device**: Piece of hardware not capable of executing programs.
- **Main Program**: Root from which the program is activated.
- **Specification**: File that contains the declaration of entities.
- **Body**: File that contains the definition of entities.

5.3.2 Relationships between Objects

5.3.2.1 Links

An object collaborates with other objects through its links to these objects. A link denotes the specific association through which one object (the client) applies the services of another object (the supplier), or through which one object may navigate to another. Messages are passed through a link.

An Object can play three roles as attached to a link:

- **Actor**: An Object that can operate upon other Objects but can not be operated upon. (usually the active objects)
- **Server**: An Object that only can be operated upon by other Objects.
- **Agent**: An Object that can both operate upon other Objects and be operated upon by other Objects. An agent usually works on behalf of an actor or another agent.Whenever one object passes a message to another object through a link, the two objects are said to be synchronized. There are three forms of synchronization when one active object has a link to a passive object:
- **Sequential**: The semantics of a passive object can only be guaranteed if there is a single active object at a time.
- **Guarded**: The semantics of a passive object are guaranteed in a system with multiple threads of control, but mutual exclusion must be realized by the collaboration of active clients.
- **Synchronous**: The semantics of a passive objects are guaranteed in a system with multiple threads of control, and the supplier guarantees mutual exclusion.

During Design it is also important to address the visibility of Objects to other Objects. There are four ways that one object may have visibility to another:

- The supplier object is global to the client.
- The supplier object is a parameter to some operation of the client.
- The supplier object is a part of the client object.
- The supplier object is a locally declared object in some operation of the client.

5.3.2.2 Aggregation

Aggregation denotes a 'whole-part'-hierarchy with the ability to navigate from the whole (aggregate) to its parts (attributes).

For example: Licence plate (CA) 3GAZ544 'is-part-of' Buick Le Sabre.

5.3.3 Relationships between Classes

5.3.3.1 Associations

An association only denotes a semantic dependency. The direction of this dependency, and the exact way in which one class relates to another are left implicitly.

5.3.3.2 Inheritance Relationships

The inheritance relationship is the relationship among Classes in which one Class shares the structure and behavior defined in one other Class (in case of single inheritance) or in more other Classes (multiple inheritance). The Class that inherits is called the Subclass and the Class from which is inherited is called the Superclass. The inheritance relationship models a 'kind-of' hierarchy. The Subclass augments and/or restricts the structure and behavior of the Superclass.

5.3.3.3 Aggregation

Aggregation relationships between classes are analogous to aggregation relationships among objects corresponding to these classes. The difference between aggregation and inheritance is that an 'is-a' relationship implies inheritance. In other cases aggregation or another relationship should be used.

5.3.3.4 Using Relationships

Using relationships among classes parallel the links among the corresponding instances of these classes. A using relationship can be a refinement of an association in which we indicate which class is the client and which is the supplier of certain services.

5.3.3.5 Instantiation Relationships

Instantiation relationships between Classes are provided for building container classes, which means Classes whose instances are (homogeneous or heterogeneous) collections of other Objects. These are often defined as parameterized classes who have to be instantiated (parameters filled in) before Objects can be created. For example: a Class Employee-Set can be defined as an instantiation of the Class Set (where as parameter Employee is filled in). The Class Employee-Set therefore uses the Class Employee. Parameter Classes provide Classes that are loosely coupled and because of this loose coupling they are more reusable.

5.3.3.6 Metaclass

A metaclass is a class of a class, e.g. a class is treated as an object that can be manipulated.

5.3.4 Operations and Communication

Operations in OOADA can be of five types:

- **Modifier**: An operation that changes the state of an object.
- **Selector**: An operation that accesses the state of an object, but does not change its state.
- **Iterator**: An operation that accesses all the parts of an object in some well-defined order.

- **Constructor**: An operation that creates an object and/or initializes its state.
- **Destructor**: An operation that frees the state of an object and/or destroys the object itself.

Message passing between Objects may take the following synchronization forms, from which the first will do for sequential systems and the other are needed for concurrent systems:

- **Simple**: In sequential systems message passing is simple.
- **Synchronous**: An operation commences only when the sender has initiated the action and the receiver is ready to accept the message. Sender and receiver wait for each other until both are ready to proceed.
- **Balking**: Like synchronous, but the sender will abandon the operation if the receiver is not ready.
- **Time-out**: Like synchronous, but the sender will only wait a specific amount of time for the receiver to be ready.
- **Asynchronous**: An operation commences regardless of whether the receiver is expecting the message or not.

5.4 Techniques

OOADA provides techniques for the four perspectives it describes; the logical and physical view and the static and dynamic semantics for these.

The techniques for the logical static view are:

- Object Diagrams for showing the existing Objects and the relationships among them, including visibility and synchronization aspects.
- Class diagrams for showing the existing classes and relationships among them including cardinalities, class utilities, concurrency and persistence and visibility aspects.
- For capturing the logical dynamic view:

- State Transition Diagrams for showing the States of an Object, the events that cause transitions and the actions resulting from transitions.
- Interaction Diagrams for describing how scenarios are executed in the same context as an Object Diagram. The difference is that the Interaction Diagram shows the dynamic aspects, while the Object Diagram shows the static aspects.
- For capturing the physical static view:
- Module Diagrams for designing the physical packaging of classes and objects into modules.
- Process Diagrams for the allocation of processes to processors. The processors and devices are the execution platform of the system.

5.5 Analysis And Design Processes

5.5.1 The Macro Process (Activity 1)

The macro process serves as a controlling framework for the micro process. It prescribes some measurable products and activities in such a way that the development team can assess risk and make early corrections to the micro process, in order to put more emphasis on the team's analysis and design activities. The macro process represents the activities of the entire development team on the scale of months or weeks at a time.

5.5.1.1 Establish Core Requirements (Conceptualization) (Activity 1.1)

Conceptualization tries to establish the core requirements for the system. The vision for the ideas is established for some application and its assumptions are validated. Conceptualization is a very creative process and should therefore not be involved with rigid development rules.

Prototypes are the primary products of this activity.

5.5.1.2 Develop a Model of the Desired Behavior (Analysis) (Activity 1.2)

The purpose of analysis is to model the world by identifying the classes and objects that form the vocabulary of the problem domain. The system's behavior is emphazised in this phase. By focusing upon behavior, function points of a systems are identified to denote the outwardly observable and testable behaviors of the system.

Analysis consists of domain analysis (activity 1.2.1) and scenario planning (activity 1.2.2). In domain analysis classes and objects that are common to a particular problem domain are identified. Scenario planning is the central activity in analysis. In this phase, the primary function points are identified, scenarios are storyboarded and documented. Where life-cycles are clear, state machines for classes are developed. In general, scenarios represent behaviors that can be tested.

5.5.1.3 Create an Architecture (Design) (Activity 1.3)

Design has to lead to an architecture for the evolving implementation, and to establish tactical issues that must be used by disparate elements of the system. Design can be divided into architectural planning (activity 1.3.1), tactical design (activity 1.3.2) and release planning (activity 1.3.3).

In architectural planning, the aim is to create very early in the life cycle a domain-specific application framework that can be successively refined. It encompasses devising the layers and partitions of the over-all system. During tactical design, decisions are made about the common policies.

Release planning aims to organize architectural evolution; an identification is made of a controlled series of architectural releases. Also, a formal development plan is yielded, which identifies the stream of architectural releases, team tasks and risk assessments.

5.5.1.4 Evolve the Implementation (Evolution) (Activity 1.4)

The purpose of evolution is to establish growth and change in the implementation through successive refinement, until the production system is reached. Evolution consists of application of the micro process (activity 1.4.1) and change management (activity 1.4.2).

Application of the micro process begins with an analysis of the requirements for the next release, after which it leads to the design of an architecture, and then classes and objects are invented that are necessary to implement this design. The main product is a stream of executable releases representing successive refinements to the first release of the architecture. Behavioral prototypes are also produced for exploring alternative designs or to further investigate unknown parts of the systems' functionality.

Change management attempts to recognize the incremental and iterative character of the object-oriented system. Undisciplined change to class hierarchies and protocols, or mechanisms, should be possible, but unrestrained change is a threat for the strategic architecture and the development team.

5.5.1.5 Manage Postdelivery Evolution (Maintenance) (Activity 1.5)

Maintenance is the activity of managing postdelivery evolution. This phase is mainly a continuation of evolution. However, now more localized changes are made to the system as new requirements are added and bugs are being eliminated. Due to the parallel with evolution, maintenance activities are application of the micro process (activity 1.5.1) and change management (activity 1.5.2). In addition to these, maintenance involves a planning activity that prioritizes tasks on a punch list (activity 1.5.3).

5.5.2 The Micro Process (Activity 2)

The micro process is largely driven by the stream of scenarios and architectural products that result from and that are refined by the macro process. It mainly represents the daily activities of the individual developer or a small development team.

In the micro process, the traditional phases of analysis and design are intentionally blurred, and the process is under opportunistic control.

5.5.2.1 Identify Classes and Objects (Activity 2.1)

Identifying Classes and Objects is done by finding the significant classes and objects in the problem space (activity 2.1.1) and the mechanisms that provide the behaviour required of objects that work together to achieve some function (activity 2.1.2).

This can be done by object-oriented analysis, behavior analysis and/or use-case analysis. This activity results in a data dictionary of candidate classes and objects and a document describing object behavior.

5.5.2.2 Identify the Semantics of Classes and Objects (Activity 2.2)

The aim is to establish the state and behavior of each abstraction identified in the previous phase. Three activities are performed in this phase: storyboarding (activity 2.2.1), isolated class design (activity 2.2.2) and pattern scavenging (activity 2.2.3).

Semantics are represented in a top-down way in storyboarding and, where it concerns system function points, strategic issues are addressed. Isolated class design results in a bottom-up identification of semantics. Isolated class design means good class design, not architectural design. It is therefore more tactical in nature than

storyboarding. In the third activity, pattern scavenging, commonality in patterns of behavior are discovered, because it may contribute to reusability.

5.5.2.3 Identify the Relationships Among Classes and Objects (Activity 2.3)

The purpose of identifying the relationships among classes and objects is to solidify the boundaries of each abstraction and to identify co-operating classes and objects. This step consists of three activities: The specifications of associations (activity 2.3.1), the identification of various collaborations (activity 2.3.2) and the refinement of associations (activity 2.3.3).

The identification of associations is mainly an analysis and early design activity. The resulting product of this activity is a class diagram. The identification of collaborations is primarily a design and classification activity. Collaborations are documented in object- and module diagrams. Class diagrams are refined to show tactical decisions about inheritance, aggregation, instantiation and use. The third activity, the refinement of associations, is performed both in the analysis and design phase. The result is a more specified description of semantics and relationships.

5.5.2.4 Implementation of the Classes and Objects (Activity 2.4)

During analysis, the purpose of implementing classes and objects is to refine existing abstractions and to discover new classes and objects at the next level of abstraction, which can then be iterated in the micro process. During design, the purpose is to create representations of abstractions in support of the successive refinement of the executable releases in the macro process.

There is one step in this activity: the selection of the structures and algorithms that provide the semantics of the earlier identified abstractions. The first three phases of the micro process discuss the outside view of abstractions; this final step focuses upon their inside view. Representational issues of each abstraction and the mapping of these representations to the physical model are the yielded products.

CHAPTER SIX
OBJECT LIFECYCLES (OL) BY SHLAER AND MELLOR

6.1 Background

'Object Lifecycles: Modeling the World in States' is a complete revision of the previous book 'Object-Oriented Systems Analysis: Modeling the World in Data', which was published in 1988.

Shlaer & Mellor developed the Object-Oriented Systems Analysis (OOSA) method because there was no method by which they could lay out candidate definitions of conceptual entities and examine the implications of those definitions. The main technique Shlaer & Mellor use is called Information Modeling, which is used to aid in the formalization of knowledge.

Since OOSA, guidelines have been improved to partition actions into processes with desirable properties. Also, higher-level diagrams have been introduced for the benefit of scale-up issues concerning large projects.

6.2 General Approach

OL divides System Development into OO Analysis and OO Design. OO Analysis is described in three steps:

- Information modeling
- In this step, the focus is on abstracting the conceptual entities in the problem domain in terms of objects and attributes. The associations that exist between the entities are formalized as relationships that are based on the policies, rules, and physical laws that prevail in the real world.
- State modeling
- This step is concerned with behavior of objects and relationships over time. State models are used to formalize the lifecycles of both objects and relationships. The state models, which consist of state transition diagrams and

tables, communicate with each other by means of events. State models are defined by multi-layers of state transition diagrams to make the model of communication orderly and understandable.

- Process modeling
- The actions of the state models, which contain all of the required processing, are dissected into fundamental and reusable processes and are expressed by an enhanced form of the traditional data flow diagram - the Action DFD. The processes that are so derived can then be converted directly into operations of object-oriented design.

6.2.1 Object-Oriented Design

Design is expressed in terms of a single program. Each program is made up of:

- A main program, responsible for intertask communicating, the invocation of operations, and the initialization of application classes
- Four architectural classes that provide mechanisms required to initialize and traverse state machines and implement the Timer object of the analysis phase
- Some number of application classes that are derived from the objects and state models as yielded during analysis.

Shlaer and Mellor use OODLE: a language-independent textual and graphical notation for object-oriented design.

6.2.1.1 Changes to the First Edition

- Information models
- Key letters were assigned to objects and a numbering of relationships has been introduced to make correlations between elements easier and to increase understandibility of the graphical representation of the information model. The correlation table for facilitating understanding about the associative

object has been abandoned. Instead, a more general associative object is introduced.

- State models
- The monitor state model, previously used to formalize dynamics of a competitive relationship, has been generalized, and is now a special case of the more general Assigner state model. The matter of lifecycles for subtype and supertype objects has been clarified. The Timer object now appears as an element object. Finally, an alternative labeling of events has been developed to keep track of events better.
- Process models
- This modeling step has been changed significantly. Guidelines have been formulated for casting the actions into separate processes, for naming the processes and for describing them. Action data flow diagrams describe the order of process execution, and conditional outputs are shown explicitly.Attention is paid to the problem of how to label a data flow between two processes that have entirely different perspectives on the meaning of the data.

6.3 Concepts and Constructs

6.3.1 Concepts

- **Object**: An object is an abstraction of a set of real world things that have same characteristics and must conform to same rules.
- **Attributes**: An attribute is the abstraction of a single characteristic possessed by all the entities that were themselves abstracted as an object.
- **Identifier**: A set of one or more attributes which uniquely distinguishes each instance of an object.
- **Domain**: The set of values an attribute can draw from.
- **Relationship**: The abstraction of a set of associations that exist between different kinds of things in the real world.

- **Associative object**: An object that holds references to the identifiers of the instances of a many-to-many relationship.
- **Subsystem**: Section of the information model in which clusters, i.e. groups of Objects that are interconnected by many relationships, remain intact.StateA stage in the lifecycle of instances of Objects.
- **Transition**: Change of State.
- **Event**: Something that indicates that a progression is happening and that causes a transition.
- **Action**: Something performed upon entering a state.
- **Timer**: A mechanism that can be used by an action to generate an event at some time in the future.
- **Data flow**: Input data flowing to a process or output data flowing from a process.
- **Event data flow**: A data flow pointing into a process from nowhere.
- **Process**: Something that manipulates data.
- **Data Store**: Information Store.
- **Accessor**: An accessor accesses data in a single object data store.
- **Published operation**: A method or function that is made generally available for use from outside a Class.
- **Deferring operation**: A polymorphic operation in which the name of the operation and input/output parameters are specified.
- **Class**: The translation of an object from the analysis phase into the design phase. OL does not give a definition of a Class.
- **Logical component**: Describes (part of) the data type of the class, e.g. 'month' is a logical component of the data type 'date'.
- **Main program**: The main program has three responsibilities: invocation of initialization operations on classes, generation of external events that initiate or continue a thread of control and generation of timer events.
- **Module**: A piece of code that is invoked by one or more other modules and, when complete, returns control to the calling module.

6.3.2 Relationships between Objects

6.3.2.1 Super/Subtype Relations (Inheritance Relationship)

Generalization mechanism to factor out common attributes of two or more objects. It has the form: subtype "is-a" supertype. An object can participate in multiple subtype-supertype relationships.

6.3.2.2 Binary and ternary relations between objects (Association relationships)

For example: Object person works at Object office.

6.3.3 Relationships between Classes

These relationships are called dependencies. There are two types of dependency:

- **Client-server**
  A client-server relationship occurs between two classes when a module of one class (call it Class A) invokes a published operation of another class (call it Class B). The invoking class (Class A) is called the client, while the invoked class (Class B) is called the server.
- **Friends**
  When a module of one class (Class A) either invokes an internal operation of another class (Class B) or makes direct access of the data of the other class, Class A is said to be a friend of Class B. The same holds for modules.

Other relationships in OL are Data Flows on the Action Data Flow Diagram to connect processes with other processes and data stores.

6.3.4 Operations and Communication

Shlaer and Mellor describe the following operations:

- **Published** : A published operation can access and manipulate the logical components of the data type. There are two categories:
    - Instance-based operations, in which the caller of the operation supplies an instance of the underlying data structure. The operation then manipulates that data structure.
    - Class-based operations, in which the caller does not supply a particular instance of the underlying data structure. Queries, iterators and create operations are examples of class-based operations.
- **Deferred** : A deferred operation is polymorphic. It is supplied in classes that inherit from the class in which the deferring operation was specified.
- **Run-time bound** : The operation is carried out at run-time.

In the Object Communication Model only a few patterns of communication are observed. A distinction between types of events that may occur, will clarify the patterns of communication:

- External events are generated by an external entity and received by a state model.

  There are two types of external events:
    - An unsolicited event is not caused to occur by some previous actions of the system.
    - A solicited event is generated in response to some previous activity of the system.
- Internal events are generated by a state model within the system.

Two patterns of communication can be described:

- **Top-driven pattern** : In this pattern a top-layer object receives unsolicited events from operators (or similar intelligent external entities) to initiate significant operations for the system as a whole.
- **Bottom-driven pattern** : In this case a bottom-layer object receives an unsolicited event from e.g. a physical device. The object has insufficient context to determine the appropriate response. It therefore adds information and generates an internal event upwards until an object with sufficient knowledge and context is reached.

6.4 Techniques

OL uses the following techniques in the analysis phase:

- Information structure diagrams for the information modeling. This drawing is sometimes called an Entity-Relationship Diagram or simply as the Information Model.
- State Transition Diagrams for describing the Object lifecycle.
- Object Communication Model for showing asynchronous event communication between state models and external entities such as operators, physical devices and objects in other subsystems.
- Thread of Control Chart for showing the order of events and states that belong to the instances in a thread of control. A thread of control is defined as a sequence of actions and events that occurs in response to the arrival of a particular unsolicited event when the system is in a particular state (p.94).
- Action Data Flow Diagrams for the representation of the units of processing within an action and the communication between those units of processing.
- Object Access Models provide the complementary view of the object communication model. It shows the synchronous communication between state models and object instance data.

For the information modeling it is stressed that for every object, attribute and relationship in the model, a non-graphical specification can be made. Each state

model not only consists of a state transition diagram, but also it provides a state transition table, containing the transition rules and a description of each action on the state transition diagram (required only if the actions were too long to put on the state transition diagram). In addition, an event list is provided showing all the events that have been defined for all the state models.

Additional work products of the process modeling phase include a state process table which contains the action data flow diagrams for all the actions, and process descriptions which are a compilation of any process descriptions produced.

During design we use:

- Class Diagrams, depicting the external view of a single class.
- Class Structure Charts, showing the internal structure of the code of the operations of the class.
- Dependency Diagrams, depicting the client-server (invocation) and friend relationships that hold between classes.
- Inheritance Diagrams, showing the inheritance relationships that pertain between the classes.

6.5 Analysis And Design Processes

6.5.1 Constructing the Object-Oriented Analysis Model (Activity 1)

6.5.1.1 Constructing the Information Model (Activity 1.1)

The first activity of the OL method is creating an Information Model, to define conceptual units which are important in the real world and for the system. First, an identification of objects is made (activity 1.1.1), resulting in a list of candidate objects. Then objects are described (activity 1.1.2) and attributes are found and classified (activity 1.1.3). Identifiers are defined (activity 1.1.4) in order to uniquely distinguish each instance of an object. A description of each attribute and its domain

are made (activity 1.1.5) and relationships between objects are modeled and described (activity 1.1.6). These relationships are formalized by placing referential attributes in appropriate objects on the model (activity 1.1.7). After that, compositions of relationships are made, denoting that some relationships come about as a necessary consequence of the existence of other relationships (activity 1.1.8). Finally, subtypes and supertypes are described (activity 1.1.9).

6.5.1.2 Constructing the State Model (Activity 1.2)

State models are built to formalize the lifecycles of both objects and relationships. This lifecycle is described in a State Transition Diagram for every object which needs dynamic behavior. Additional work products include a State Transition Table, a description of each action on the State Transition Diagram and an event list.

6.5.1.3 Constructing the Process Model (Activity 1.3)

The aim is to dissect actions from the state models into fundamental processes. The processes are depicted in an Action Data Flow Diagram. The construction of the process model can be divided into four steps.

First, actions are partitioned into processes (activity 1.3.1). Then processes are named and described (activity 1.3.2), resulting in process descriptions. The third step is the construction of a State Process Table (activity 1.3.3), which provides a compact listing of the processes in the system and the actions in which they are used. Finally, an Object Access Model is constructed (activity 1.3.4) showing the synchronous communication between State Models in the system.

6.5.2 Constructing the Object-Oriented Design Model (Activity 2)

6.5.2.1 Produce Class Diagrams and Class Structure Charts (Activity 2.1)

Shlaer and Mellor suggest that Class Diagrams and Class Structure Charts should be produced in detail only as archetype diagrams - diagrams that indicate where specialization must be accomplished by name substitution. This phase is called 'application'.

6.5.2.2 Transform Application into Architecture (Activity 2.2)

In the application domain that has been described by archetype diagrams, a class diagram is produced for each class and action segments are produced of the Class Structure Charts. A Class Diagram is considered architectural, because it depicts the external view of a single class.

6.5.3 Transform Architecture into Implementation (Activity 2.3)

In this step the modules of the archetype Class Structure Charts are expressed as archetype code (code templates) in an implementation language. The templates can then be filled out either from the populated architecture database or from the specialized Class Structure Charts.

CHAPTER SEVEN
PRINCIPLES OF OBJECT-ORIENTED ANALYSIS AND DESIGN (OOAD) BY MARTIN & ODELL

7.1 Background

OOAD from Martin is an adapted version of OOAD from Martin and Odell which was published in 1992. In this new version nothing can be found of the background ideas that were mentioned in the 1992 issue of OOAD. Since no major changes in the OOAD method occurred, the same background ideas as the older edition will hold.

The OOAD method has as main background idea managing the complex world by abstraction (removing distinctions between objects to see commonalties), generalization (distinguishing Object types that are more general than others) and composition (forming objects from its component parts).

The method is grounded in a theoretical foundation, mainly logic and set theory, and all the concepts that play a role in the method are described extensively.

7.2 General Approach

OOAD from Martin (1993) does not describe any process steps. Therefore the approach is taken from OOAD from Martin and Odell (1992).

Although Object-Oriented Analysis is often divided into structural (static) and behavioral (dynamic) aspects, the OOAD method tries to integrate these two in their process part. However, they state that Object-Oriented Analysis should be founded on a behavioral foundation, so the main focus of the method is the behavioral aspects; structural aspects are identified and described as a derivation of the behavioral aspects.

Although the product part of the OOAD method doesn't describe it as a part of the method, it is emphasized several times that a complex realm can be broken down in

manageable chunks by first using Object Flow diagrams, for which no process part is available.

The OOAD method starts with identifying the goal of the realm, and this functions as a starting point. In a cyclic way the Events and Objects are defined, and for each new identified Event the cycle has to be repeated, until all Events are defined. At this point it can be necessary to define Events on a more detailed level, if so, the cyclic approach has to be repeated for the SubEvents of the higher level Event.

The way the system is analyzed can be done in the described top-down way, however, a bottom-up approach can also be followed, in which lower level Events are described first.

A real process part is only available for the Analysis part, it is only briefly described how the analysis models can be transformed into a design and an implementation.

7.2.1 Changes to the First Edition

Generalization-Specialization relationship (Super-subtypes) is now called 'Subtypes and inheritance'. Powertype relationships are now called 'Instantiations'. Three types of operations have been deleted:

- Component termination (termination of the part Objects when the Whole Object is terminated)
- Object coalesce (Two previous distinct objects become the same object)
- Object decoalesce (Opposite from coalesce)

A number of techniques have been added. A Class-communication diagram is an important one; it shows how classes pass requests among them. Composed-of diagrams have been introduced to show aggregation relationships. To show

generalization and instance connections, a Fern-diagram is used. State transition diagrams model object behavior in a simple way. Finally, generalization-specialization diagrams provide a detailed insight of the inheritance hierarchies that exist.

Each diagram may be expanded and contracted now. In still existing techniques, some notations have been changed or omitted. The following is a list of omitted aspects:

In Event Schemas:

- Function(s) that are attached to a trigger arrow
- Operations icons in 'Event Type partitions'

In Object Schemas:

- A function as a role name on Object relationships
- A computed function
- An associaion subtype
- A powertype association
- Mutable and Immutable composition

7.3 Concepts and Constructs

7.3.1 Concepts

- **Object**: An Object is any thing in which data can be stored, as well as operations that can manipulate that data.
- **Object Type**: An object type is a category of Objects.
- **Class**: A Class is an implementation of an Object Type. It has the Object Type's data structure and methods.
- **Attribute**: An attribute is a descriptor associated with an object type.

- **Operation**: An operation is a process that changes the state of an object.
- **Method**: A method is a specification of how operations are encoded in software.
- **Clock operation**: A clock operation is an operation that has a specified pattern of clock-tick events.
- **Event Type**: An event type (or event) is the successful completion of an operation (a change in state). Events cause other operations to occur.
- **Trigger rule**: A trigger rule defines the causal relationship between event and operation. It invokes a predefined operation after an event occurred.
- **Control condition**: A control condition is evaluated before an operation starts. If the conditions are true the operation is started.
- **Activity**: An activity is a process whose production and consumption are specified.
- **Product**: A product is the end result that fulfills the purpose of an activity.
- **External entity**: Something that lies outside the boundaries of the realm, but that is relevant for the functionality of the system.
- **Client**: A class that sends a group of requests to a server.
- **Server**: A class that receives a group of requests from a client.
- **Contract**: A set of requests that a client can make of a server. The server must respond to those requests.
- **Responsibility**: A service that the server provides to the client.
- **Request**: A grouping of responsibilities.
- **Subsystem**: A grouping of classes that fulfill a common overall purpose. A subsystem divides responsibilities among groups of classes.

7.3.2 Relationships between Object Types

Subtypes and inheritance (Generalization): Generalization occurs when an object type is considered more general, or inclusive, than another. All properties that hold for the generalized object type also holds for its subtypes. An instance of a subtype is also an instance of its supertypes.Subtypes can have one supertype (Single inheritance) or more supertypes (multiple inheritance).

- Composed-of relationships: In a composed-of relationship an Object type is formed of its component parts ('is-part-of relationship).
- General relations (Associations): Relations are object types whose objects are a set of tuples. Relations can be binary, ternary or n-ary.
- Instantiation: This is the relationship between an object and its object type. For example, the object 'John Smith' is an instantiation of the object type 'Person'.

7.3.3 Operations and Communication

The basic types of operations the OOAD method describes are derived from the state changes of an Object, resulting from event types. The basic types of operations are:

- Object creation
- Object termination
- Object classification (Object classified in an object type, e.g. an undergraduate student becomes a graduate student)
- Object declassification (Object declassified from an object type, e.g. a course is dropped from the curriculum)
- Object changing classification (Object declassified from one and classified in another object type, as one simultaneous action)
- An object's attribute is changed

Object types or classes can communicate with each other through requests. A set of requests that a client can make of a server is defined in a contract. The server must respond to those requests.

A contract also defines a cohesive set of responsibilities; a responsibility is something an object does for other objects by executing a method. There are two sorts of responsibilities:

- Public: One that can be requested by other objects
- Private: One that cannot be requested by other objects

7.4 Techniques

The OOAD method has several techniques:

- Object-relationship diagrams (Object schema) for describing the static Object types and their relationships.
- Class-communication diagrams to show multiple classes and the requests that pass among them.
- Event diagrams (a kind of Finite State Machines) for describing system behavior.
- Object flow diagrams to model processes with a strategic level approach, processes are described on a high-level, lower level processes can be described using Event diagrams and/or Object schema's. Object flow diagrams are similar to data flow diagrams. The Object flow diagram is an overview diagram that represents key enterprise activities linked by the products that activities produce and exchange.
- Composed-of diagrams are hierarchical diagrams that are used to show that an object type (or class) is composed of other object types (or classes). The composed-of relationship is often visualized in an object-relationship diagram.
- Fern-diagrams. Generalization is commonly represented with a Fern-diagram. This diagram indicates the direction of inheritance. It does not depict what is being inherited or how the inheritance mechanism works. Fern-diagrams support multiple supertypes.
- State transition diagrams for showing the sequence of states that an object passes. This technique shows object behavior.
- Generalization-specialization diagrams can also be referred to as an inheritance hierarchy or subtyping diagram. The 'is-a-subtype-of'-relations are often shown in the object-relationship diagram.

7.5 Analysis And Design Processes

Remark: OOAD from Martin (1993) does not describe process steps. Therefore, a description is given of the process part as described in Martin and Odell (1992). Steps 2 until 6 are performed in a cyclic pattern, one cycle for each identified Event Type.

7.5.1 Analyze Object Behavior (Activity 1)

7.5.1.1 Define Analysis Focus (Activity 1.1)

The first activity in the OOAD method is identifying the realm of interest or the Universe of Discourse for the problem at hand (activity 1.1.1). This realm should be identified by the analyst and the expert together. After identifying the realm the goal event type should be identified (activity 1.1.2) to describe clearly the purpose of the realm. The goal event type is the change in object state that should be reached in the realm. The first event type that will be examined in activity 1.2 is the goal event type.

7.5.1.2 Clarify Event Type (Activity 1.2)

Clarifying an event type starts with identifying the kind of event to which the examined event belongs (activity 1.2.1). It also has to be noted which object types are involved in the event (the object types of the event pre-state and the event post-state) for the object schema. This together states unambiguously the event type. Next, the event type can be named (activity 1.2.2). The name should document the name of the pre- or post-state object type and the process it represents.

7.5.1.3 Generalize Event Type (Activity 1.3)

To define whether the event type describes the accurate level of abstraction the event type with its pre- and post-states should be generalized and it should be decided which level of generalization is the most appropriate (activity 1.3.1). The identified object types for the pre- and post-states also have to be modified to the

object types of the most appropriate level for the object schema. After the generalization step it may be possible that this generalized event type was already specified. If so the event type may be integrated with the already specified event types (activity 1.3.2).

7.5.1.4 Define Operation Conditions (Activity 1.4)

After specifying the event type it is possible to identify which operation leads to the specified event type (activity 1.4.1). Then it can be determined whether this operation is internal (processing of operation occurs within the realm) or external (processing of operation occurs outside the realm) to the realm (activity 1.4.2). Next the control conditions for the operation have to be identified (activity 1.4.3). Thus it has to be determined under which conditions the operation may be started. It is imported to let the expert state this conditions in the experts language. The analyst then can transform this conditions in a normalized form, for example in a disjunctive normal form (activity 1.4.4). Then each of the terms of the control condition describes object types that should appear on the object schema.

7.5.1.5 Identify Operation Causes (Activity 1.5)

First the triggering event type must be determined (activity 1.5.1). This can be achieved by looking at the control conditions and identify what kind of events must occur to satisfy the control conditions. When these events are identified it can be determined which of these events trigger the operation. When this is done, it can be indicated in the Event diagram which control conditions are complex and which are not (activity 1.5.2).

Now it can be possible that different triggering events apply only to certain parts of the control condition. This causes a regrouping of the control conditions in several control condition blocks (activity 1.5.3).

The last activity is specifying the trigger rules (activity 1.5.4). This means an exact specification of the objects that are needed to invoke the operation. Also the functions that are needed to do the mapping to these objects have to be specified and the functions have to appear on the object schema.

7.5.1.6 Refine Cycle Results (Activity 1.6)

After identifying the triggering event types it has to be identified if these triggering event types can be generalized into a higher level event type (activity 1.6.1). It also has to be identified if it is possible and accurate to specialize the goal event (activity 1.6.2) to clarify its meaning. The last activity is checking the triggering event types to see if these are the same or a subclassification of the goal event type (activity 1.6.3). The duplicate events should be removed.

After activity 6 this cycle is ready, and the cycle from 1.2 to 1.6 should be repeated for identified (internal) events, until all internal events are specified.

If not all operations are at the 'basic' level of description, it may be necessary to specify these operations on a more detailed level in another event schema for which the cycle has to be repeated.

CHAPTER EIGHT
THE FUSION METHOD BY COLEMAN

8.1 Background

The Fusion method considers itself a 'second-generation' development method. It has integrated and extended existing approaches to provide a direct route from a requirements definition through to a programming-language implementation. It gives requirements of development for and with reuse and reengineering. The following methods or techniques have influenced Fusion:

- OMT (Rumbaugh et al., 1991): The Object model is almost similar to that in OMT. The Operation model is analogous to the functional model in OMT; the dataflow diagrams of OMTare not appropriate according to Coleman and have been formalized with pre- and postconditions.
- Formal methods: Pre- and postconditions are adopted to describe formally what a system does.
- Class Responsibility Collaborator (CRC): CRC extended with communication information has influenced Object interaction graphs.
- OO Design with Applications (Booch, 1991): Visibility graphs are influenced by visibility information on Booch's Object diagrams.

Fusion is based on a small but comprehensive set of well-defined diagramming techniques for describing analysis and design steps. Fusion consists of three phases: analysis, design and implementation. The steps in each phase order important decisions and the deliverables of each step are inputs for a following step. Associated with each step, Fusion also provides rules for checking the consistency and completeness of the developing models. In addition, Fusion can be adapted. Some guidelines are given to obtain a simplified version of Fusion in case not all techniques are fully required and it is described how Fusion can be combined with other methods.

8.2 General Approach

Fusion divides the development process in the phases analysis, design and implementation. As it is assumed that a customer will supply the initial requirements document, Fusion has no requirements phase.

In the analysis phase the intended behavior of the system is defined. Models in this phase describe classes of objects, relationships between classes, operations that can be performed on the system and allowable sequences of those operations.

In the design phase models are produced that show how system operations are implemented by interacting objects, references between classes, inheritance relationships, attributes of classes and operations on classes.

During implementation inheritance, reference and class attributes are implemented in programming language classes. Interactions between objects are programmed as methods belonging to a class. State machines are implemented to recognize allowed sequences of operations.

Fusion maintains a data dictionary, a repository where the different entities of the system can be named and described.

8.3 Concepts and Constructs of Fusion

8.3.1 Concepts

- **Object**: A concept, abstraction or thing that can be distinctly identified
- **Attribute**: A set of named values associated with an object or relationship
- **Class**: An abstraction that represents the idea or general notion of a set of similar objects
- **Abstract class**: A class without any direct instances
- **Relationship**: A tuple of objects <John Smith, Dept_1>
- **Invariant**: An assertion that some property must always hold

- **Role**: Name that qualifies the class participating in a relationship
- **Agent**: Entity in the system environment that invokes system operations (by sending events) and receives results (carried by events)
- **Event**: An instantaneous and atomic unit of communication between the system and the environment
- **Input event**: Input communication with which the environment requests the system to perform an operation
- **Output event**: Asynchronous output communication sent to an agent of a system
- **System operation**: An input event and its effect on a system. A system operation is invoked by an agent in the environment
- **Interface**: The set of system operations which it can receive and the set of events that it can output
- **State (of a system):** A set of objects that participate in relationships
- **Precondition**: A predicate that characterizes the conditions under which a system operation may be invoked
- **Postcondition**: A predicate that describes how the system state is changed by a system operation and what events are sent to agents
- **Controller**: Object (or an object interaction graph) responsible for responding to a system operation request
- **Collaborator**: Object (or an object interaction graph) that provides some functionality as a server to implement a system operation
- **Message**: The thing that is sent to an object to request it to perform an operation
- **Method**: The implementation of an operation (on a class).

8.3.2 Relationships between Classes

- Binary, ternary and n-ary relationships (also hold between Objects): A relationship is seen as a tuple of objects that may be labeled with an attribute.The arity of a relationship is the number of separate objects that

participate in the relationship. A cardinality constraint restricts the number of objects that may be associated with each other in a relationship.

- Aggregation: Aggregation is a mechanism for structuring the object model. It is a relationship type that shows how a class contains other classes.
- Generalization and specialization: Generalization is defined when a class, called the supertype, is formed by taking common properties of several classes, called the subtypes. The attributes and the relationships of the supertype are inherited by all the subtypes. Each subtype is allowed to have additional attributes and participate in additional relationships. An important property of generalization is that all the objects of a subtype also belong to the supertype. Multiple generalization occurs when a class can be divided into two or more superclasses.

Specialization occurs when a new subtype is defined as a more specialized version of a supertype. Multiple specialization allows a new subtype to be defined as a specialization of more than one immediate supertype. As mentioned above, all the attributes and relationships of the superclasses are inherited by the subclass.

8.3.3 Operations and Communication

A system operation is an input event (sent by an agent to the system) and the effect it can have. When a system receives such an event it can cause a change of state and the output of events. At any point of time only one system operation can be active. A system operation may:

- create a new instance of a class
- change the value of an attribute of an object
- add or delete some tuples of objects from a relationship
- send an event to an agent

Communication between objects:

Communication paths are visibility relationships in the Fusion method. These are organized into four categories:

- Reference lifetime, also called dynamic reference, occurs when a client only needs to message a server in the context of a single method invocation. Access can be given through a parameter or by a local variable of the method.
- Server visibility. A server object may be exclusively used by a client that sends the server a message or in case the server may be shared by several clients.
- Server binding, if deleting one object implies deleting all related objects, we say the lifetimes of related objects are bound. Server binding thus means that if a client holds the only the reference to the server, then the server object should be deleted when the lifetime of the client comes to an end.
- Reference mutability. The mutability of a reference indicates whether it can be reassigned after initialization. Constant mutability means a reference is not assignable after initialization. If the mutability is variable, then it may be reassigned.

8.4 Techniques

The Fusion method uses several techniques. The analysis phase ends with operation models and design begins with making object interaction graphs and is finished by making inheritance graphs. A description of the techniques is summarized below:

- Object models, which define the static structure of the information in the system. Both the concepts in the problem domain and relationships between them are described. The object model notation is based on extended entity relationship notation.
- System object models. A system object model is a subset of an object model that relates to the system to be built. It is formed by excluding all the classes and relationships that belong to the environment.

- Interface models define the input and output communication of the system. The description is in terms of events and the change of state that they cause. The interface of a system is the set of system operations which it can receive and the set of events that it can output. Different aspects of behavior are described in two models:
    - 3a. Life-cycle models, which characterize the allowable sequencing of system operations and events. A life-cycle expression defines a pattern of communication by showing the allowable sequences of interactions that a system may participate in over its lifetime.
    - 3b. Operation models, which describe the effect of each system operation in terms of the state change it causes and the output events it sends.The operation model uses preconditions and postconditions.
- Object interaction graphs. These graphs are constructed for each system operation. It defines the sequences of messages that occur between objects to realize a particular operation.
- Visibility graphs show the reference structure of classes in the system. The following is identified for each class:
    - Objects that class instances need to reference
    - Appropriate kinds of reference to those objects
- Class descriptions. One class description is made for each class. It describes the methods, some data attributes and object-valued attributes. A class is a set of objects that has the same attributes and methods, some of which may be inherited from superclasses. The descriptions serve as foundations for implementation.
- Inheritance graphs show generalization-specialization relations between classes. There is one difference between generalization and specialization in the analysis phase and the inheritance structures of design. In the analysis phase, abstract subtype relationships between classes are identified that exist in the domain of the system and not of the system design or implementation. At design, the inheritance relationship is implemented in the system and is not necessarily identified in the domain. The inheritance relationships here are prescriptive, while in the analysis phase they are descriptive.

8.5 Analysis And Design Processes

8.5.1 Analysis (Activity 1)

During analysis an architectural description of the system is made. It is a user's perspective of what the system does. Therefore emphasis is put on the domain description and on externally visible behavior.

8.5.1.1 Develop the Object Model (Activity 1.1)

In the object model the concepts in the problem domain and relationships between them are described. The object model is constructed in the following steps:

- A list of candidate classes and relationships results from brainstorming (Activity 1.1.1).
- A data dictionary is filled with these classes and relationships (Activity 1.1.2).
- The object model is produced incrementally by searching for relationships, attributes, constraints and cardinalities (Activity 1.1.3).

8.5.1.2 Determine the System Interface (Activity 1.2)

The system interface is the set of system operations to which it can respond and the events that it can output. The data dictionary is filled with every agent, which invokes a system operation, every system operation and every output event. First the agents, system operations and events are identified (Activity 1.2.1), whereafter the system object model is produced (Activity 1.2.2). The system object model is a refinement of the object model, that is now traced out with interface boundaries.

8.5.1.3 Develop an Interface Model (Activity 1.3)

The interface model consists of a life-cycle model and an operation model. The life-cycle model which shows allowable sequences of interactions, is constructed in

two steps. First, scenarios are generalized and life-cycle regular expressions are formed (Activity 1.3.1). Life-cycle expressions are then combined to form the life-cycle model (Activity 1.3.2). The semantics of each system operation are captured in the operation model. For each system operation "Assumes" and "Results" clauses are developed (Activity 1.3.3). An "Assumes" clause is a precondition that holds before and during an operation is invoked. A "Results" clause is postcondition that has to be true in a final state that the operation has delivered. In the next step, "Sends", "Reads" and "Changes" clauses are extracted from the "Assumes" and "Results" clauses (Activity 1.3.4). A "Sends" clause is applied to an agent and all the events that an operation may send to the agent. A "Reads" clause is defined on items that the operation may only read and not change. A "Changes" clause is specified for items that may be accessed and changed.

8.5.1.4 Check the Analysis Models (Activity 1.4)

Analysis models are checked on completeness and consistency. First, completeness is checked against requirements (Activity 1.4.1). An example of such a check is that all possible scenarios must be covered by the life-cycle. Simple consistency is then checked (Activity 1.4.2), meaning that overlaps between models have to be consistent. Finally semantic consistency is checked (Activity 1.4.3), in which implications of the models have to be consistent.

8.5.2 Design (Activity 2)

In the design phase, abstract definitions produced in the analysis phase are transformed to software structures.

8.5.2.1 Object Interaction Graphs (Activity 2.1)

These graphs show how functionality is distributed across the objects of a system. One graph is made for each system operation. The construction of an object interaction graph takes four steps. The first step is to identify relevant objects that

are involved in computation (Activity 2.1.1). Secondly, the role of each object is established, resulting in the identification of the controller and the collaborators (Activity 2.1.2). In the third step a decision is made on messages between objects (Activity 2.1.3). It is described how all collaborators interact to carry out the operation. The fourth step is to record how objects interact. Consistency with analysis models is checked together with a verification of functional effects of objects in accordance to the functional specification of its system operation in the operation model.

8.5.2.2 Visibility Graphs (Activity 2.2)

In order to identify all visibility references, each message on an object interaction graph has to be studied; therefore all object interaction graphs are inspected (Activity 2.2.1). Then the kind of visibility reference is identified (Activity 2.2.2), whereafter a visibility graph can be drawn for each design object class (Activity 2.2.3). In the resulting visibility graph a check is made if for each relation in the system object model there is a visibility reference for the corresponding classes on the visibility graph. Mutual consistency is also checked, meaning that exclusive target objects and shared targets are not referenced by more than one class.

8.5.2.3 Class Descriptions (Activity 2.3)

For each class, a description is made of the internal state and the external interface. Information is extracted from the system object model, object interaction graphs and visibility graphs. From the object interaction graph methods and parameters are extracted (Activity 2.3.1). The system object model and the data dictionary provide data attributes (Activity 2.3.2), while object attributes are derived from the visibility graph for the class (Activity 2.3.3). Inheritance information is identified in the inheritance graph (Activity 2.3.4). A check is made for the completeness of all the derived information.

8.5.2.4 Inheritance Graphs (Activity 2.4)

Inheritance structures can be found by screening the classes for commonalties and abstractions. Superclasses and subclasses are identified. Three process steps are taken to construct inheritance graphs. The object model is inspected for generalizations and specializations (Activity 2.4.1), whereas common methods can be found in object interaction graphs and class descriptions (Activity 2.4.2). Finally, visibility graphs are inspected for finding common visibility (Activity 2.4.3).

8.5.2.5 Update Class Descriptions (Activity 2.5)

The class descriptions can now be updated with the inheritance information. In the system object model it is checked that subtype relations are preserved (Activity 2.5.1). Object interaction graphs and visibility graphs are inspected to verify that all classes are represented in an inheritance graph (Activity 2.5.2 and Activity 2.5.3), whereafter the updated class descriptions are checked on completeness and on consistency with the inheritance graphs.

8.5.3 Implementation (Activity 3)

The design can now be mapped into an effective implementation.

8.5.3.1 Coding (Activity 3.1)

Coding means transforming the outputs of design into code in an implementation language. It is concerned with three elements: the system life-cycle, class descriptions and the data dictionary. Coding the system life-cycle can be divided in the case there are life-cycles without interleaving and the case there are life-cycles with interleaving. When there is no interleaving, the life-cycle regular expressions are translated into a state machine (Activity 3.1.1a). The state machine is then implemented (Activity 3.1.2a). For life-cycles with interleaving, the interleaving-free subexpressions are implemented (Activity 3.1.1b), after which the resulting state

machines can be linked (Activity 3.1.2b). Coding class descriptions begins with specifying the representation and interface of a class: attention is paid to attribute- and method declaration and inheritance (Activity 3.1.3). Method bodies are implemented by viewing object interaction graphs and the data dictionary. Emphasis is put on error handling and iteration (Activity 3.1.4).

The data dictionary is coded by implementing functions, predicates and types that are found in the data dictionary and are used by methods. (Activity 3.1.5). If necessary, add new code to ensure that assertions will hold on methods (Activity 3.1.6).

8.5.3.2 Performance (Activity 3.2)

This step is optional and notices the importance of treating performance throughout all development steps. The system should be profiled as good as possible (Activity 3.2.1) and frequently used code should be optimized (Activity 3.2.2).

8.5.3.3 Review (Activity 3.3)

Inspections have to be made to see if there are defects in the software (Activity 3.3.1) and the software should be tested (Activity 3.3.2).

CHAPTER NINE
OBJECT-ORIENTED SOFTWARE ENGINEERING (OOSE) BY JACOBSON

9.1 Background

OOSE combines three different techniques which have been used for a long time. The first technique is object-oriented programming, which was developed during the 1960s and soon appeared to be usable in many application areas. Only during recent years it is widely adopted. From object-oriented programming, OOSE mainly uses the concepts of encapsulation, inheritance and relationships between classes and instances. Secondly, conceptual modeling is used to create different models of the system or organization to be analyzed. In OOSE they are extended with object-oriented concepts and with the possibility to model dynamic behavior. The models serve to understand the system and to obtain a well-defined system architecture. Block design, thirdly, originates from hardware design in the telecommunications area. It models a number of modules having their own functionality, that are connected together with well-defined interfaces. This vision had to be applied in software design too, because errors in a program could shut down the whole system. Block design implies greater changability and maintainability of software.

9.2 General Approach

OOSE has a so-called 'use case driven approach'. In this approach, a use case model serves as a central model of which all other models are derived. A use case model describes the complete functionality of the system by identifying how everything that is outside the system interacts with the system.

The use case model is the basis in the phases analysis, construction and testing. The aim of analysis is to understand the system according to its functional requirements. The objects are found, organized and object interactions are described. The operations of objects and the internal view of objects is described as well during analysis.

Construction encompasses design and implementation in source code. It is important that objects in the analysis phase can be found back during construction. This is called traceability. Besides traceability, components are important during construction. A component is an already defined piece of source code that can be used for implementing objects.

In testing the system is verified, meaning that the correctness of the system is checked according to its specifications.

9.3 Concepts and Constructs

9.3.1 Concepts

- **Actor**: An actor defines a role that a user can play in exchanging information with the system
- **Primary actor**: An actor who uses the system directly, performing one or some of the main tasks
- **Secondary actor**: An actor who supervises or maintains the system
- **Abstract actor**: An actor that describes a role that should be played against the system. Different actors may inherit from an abstract actor if they similar roles
- **Role**: The role is defined by the operations of an Object. It describes the purpose wherein one Object participates with another
- **User**: A user is the person who actually uses the system. A user is instance of the Actor class
- **Use case**: A use case is a complete course of events specifying all the actions between the user and the system
- **Abstract use case**: A description of commonality in other use cases. An abstract use case will not be instantiated on its own
- **Concrete use case**: A use case that really will be instantiated
- **Object**: An object is characterized by a number of operations and a state which remembers the effect of these operations

- **Entity object**: An object about which information is stored that lasts for a long time, even when a use case is completed
- **Interface object**: An object which contains functionality of the use cases that interacts directly with the environment
- **Control object**: An object that models functionality that is not in any other object, e.g. calculating taxes using several different criteria
- **Central interface object**: An interface object that contains other interface objects
- **Attribute**: An attribute contains information and its type
- **Class**: A group of objects that have similar behavior and information structures
- **Abstract class**: A class that is developed with the main purpose to be inherited by other classes
- **Concrete class**: A class that is developed with the main purpose to create instances of it
- **Stimulus**: An event that is sent from one object to another and that initiates an operation
- **Message: Intra-process stimulus**: a normal call inside one process
- **Signal**: Inter-process stimulus: it is sent between two processes
- **Operation**: An activity inside a block that may lead to a state change of the corresponding object.
- **Subsystem**: A defined group of objects in order to structure the system
- **Service package**: The lowest level of a subsystem that is to be viewed as an atomic change unit
- **Block**: Design object, which is an abstraction of the actual implementation. Blocks are later implemented as source code
- **State**: The union of all values describing the present situation
- **Transition**: The change of a state
- **Object module**: The implementation in source code of a block. A block may be implemented in more than one object module
- **Public object module**: An object module that is accessible from the outside of a block

- **Private object module**: An object module that is not accessible from the outside of a block

9.3.2 Relationships between Objects and Classes

A distinction is made between class associations and instance associations, denoting relationships between objects. Class associations are shown by dashed arrows, while instance associations are drawn with full arrows. All relationships are always unidirectional.

Class associations:

- Inherit association: The operations and the information structure of a superclass are inherited by the subclasses. A subclass may inherit from more than one superclass (multiple inheritance).
- Extension association: Extension means inserting one use case description into another use case description that must be a complete course in itself. This description is thus independent of any inserted description.
- Instance associations:
- Association (Relation): To model relationships between objects, OOSE mentions the role one object can play in relation to the other object. For example, if there is a relationship between the object 'Car' and the object 'Person' named 'driven by', then 'Person' plays the role of 'driver' in relation to the object 'Car'.
- Acquaintance association: An object is an acquaintance of another object if it knows that the other object exists. It is a static relationship, meaning that the objects cannot exchange information with each other.
- Consists-of association: This is a special type of acquaintance association denoting that is composed of other objects. The term aggregation is also used.
- Communication association: Communication associations are used to model communication between objects. Through communication associations

objects send and receive stimuli. The arrow denotes the direction of a stimulus.

9.3.3 Operations and Communication

- Operations that have to be carried out by an entity object may include:
- creating and deleting the entity object
- storing and fetching information
- behavior that must be changed if the entity object is changed

Objects communicate by sending each other stimuli. A stimulus causes an operation to be performed in the receiving object. A stimulus can either be a message, denoting synchronous intraprocess communication, or a signal, denoting synchronous or asynchronous interprocess communication.

9.4 Techniques

- Requirements model. This model delimits the system and defines its functionality. It consists of three parts:
    - use case model, which describes actors and use cases. Actors define roles that users can play in exchanging information with the system and use cases represent functionality inside the system. It's a complete course of events specifying all the actions between the user and the system (e.g. when an operator wants to generate a daily report, the operator is an actor and 'generate a daily report' is a use case).
    - problem domain object model, for developing a logical view of the system that can be used for making a noun list which can be supportive for specifying the use cases.
    - Interface descriptions. It is important that users are involved in making detailed interface descriptions. Therefore these descriptions should be made in an early stage. The interface has to capture the user's logical view of the system, because the main interest is the

consistency of this logical view and the actual behavior of the system. It can deal with both user interfaces and interfaces with other systems.

- The analysis model structures the system (the requirements model) by modeling three types of objects: interface objects, entity objects and control objects. The behavior that is modeled in the use cases is spread among the objects in the analysis model. The analysis model provides a foundation for design.
- The design model will refine the analysis model and will adapt it to the implementation environment. Interfaces of objects and semantics of operations are defined and decisions can be made about Database Management Systems and programming languages. Blocks are introduced for object types to hide the actual implementation. The design model consists of interaction diagrams and state transition graphs.
    - An interaction diagram is drawn for each concrete use case. It describes the use cases in terms of communicating objects. This communication is modeled as blocks sending stimuli to each other. Interaction diagrams support use cases with extensions. In this case, probe positions are added to an interaction diagram. A probe position indicates a position in the use case that is to be extended and often a condition is required for when the extension is allowed to take place.
    - State transition graphs are used to describe object behavior in terms of which stimuli can be received and what stimuli may cause. OOSE uses an extension of the SDL notation (Specification and Description Language which is a CCITT standard).
- The implementation model consists of the source code of the specified objects in the design model. It is desirable that a block can be easily translated into the actual object module. In a smooth implementation environment, this is typically the case.
- The test model is produced by testing the implementation model. Test specification, which is the test's class when a test is seen as an object, and test result, the execution of an instance from the test's class, are the main concepts. First, the lowest level of the system is tested (e.g. object modules

and blocks), later use cases can be tested and finally a test can be performed on the whole system. The requirements model can serve as a verification for the test model.

9.5 Analysis And Design Processes

9.5.1 Analysis (Activity 1)

In analysis, two models are yielded: the requirements model and the analysis model. The requirements model describes all the functional requirements from user perspective and the way the system is to be used by end users. The requirements model can be structured into an analysis model. It is structured from a logical perspective into a robust and maintainable system. OOSE does not describe steps of how these two models can be obtained.

9.5.2 Construction (Activity 2)

Construction is divided in two steps, design and implementation.

9.5.2.1 Design (Activity 2.1)

Design consists of three phases. First, the implementation environment has to be identified (Activity 2.1.1). The consequences that the implementation environment will have on design are studied. The identification and investigation of the implementation environment results in strategic implementation decisions. Secondly, a first approach to a design model is developed (Activity 2.1.2) in which the analysis objects are translated into design objects fitting in the implementation environment. Finally, the interaction of objects (i.e. the stimuli) in each use case is described (Activity 2.1.3), resulting in object interfaces.

9.5.2.2 Implementation (Activity 2.2)

In this step each object is implemented in the programming language (Activity 2.2.1).

9.5.3 Testing (Activity 3)

Testing begins with test planning (Activity 3.1) in which it is examined whether existing test programs and data can be used, whether they can be modified or if new development has to take place. Also, decisions about subtests can be made, e.g. concerning standards for testing and required resources for the subtests. Information throughout the test process is stored in a test log, which serves also as the basis for further refinement of the test process and the planning of new tests.

After the test planning, tests are identified (Activity 3.2). A detailed description is made of what should be tested and how much resources are needed approximately. The third step is to specify the tests (Activity 3.3). It brings out an overview description of the test and its purpose. Besides that, a detailed procedural description of each step is made in the test. In the fourth step the tests are performed (Activity 3.4), each use case one by one. A decision table is used to store the test results of each subtest. A test result is called an outcome and can be 1 for OK and 0 for fail. By adding weight factors to subtests according to their importance, the overall test can be measured and can be compared to a limit. If the weighted sum of outcomes of the subtests exceeds the limit, then the test is approved. In case the test failed, the failure is analyzed (Activity 3.5). After analysis, tests are respecified and performed again until the test approves. In that case the test is finished (Activity 3.6). The equipment and test bed should be restored as well as the documentation made during preparation until the completion of the test. The test log is filed and can be used in next tests.

CHAPTER TEN
OBJECT-ORIENTED PROGRAMMING LANGUAGES VERSUS THE ANALYSIS AND DESIGN METHODS

10.1 Programming Considerations

This comparison focuses on how the concepts of an object-oriented method can be directly supported in selected object-oriented programming languages. A concept of a method is said to be directly supported by a programming language if an equivalent concept can be found in that language. This definition does not imply that this concept can not be implemented in that programming language. For example, the multiple inheritance concept can be implemented by a language that only supports single inheritance when a kind of simulation for multiple inheritance is implemented in the single inheritance programming language. However, the issue whether a concept of a method can be implemented is not well defined and heavily depends upon the language proficiency of a programmer. Hence, this issue is excluded from the comparison.

In the table that describes the connection between the concepts of methods and programming language, the following symbols are used:

- (String) This concept of the language that is equivalent to that of the method.
- (<) This concept of the method consists less than the concept of the language.
- (>) The concept of the method consists more than the concept of the language.
- (-) The concept of the method is not directly supported by the language.
- (*) The concept of the method has the same name as the concept of the language .

The String symbol may be combined with either '<' or '>' symbol in a table entry.

The selected programming languages include C++, Smalltalk, Java and Delphi because these are the most well-known and used object-oriented languages.

State transition diagrams and Dataflow Diagram concepts are not reviewed and compared, because we assume that these are implementation issues that are not related to a mapping of concepts between method and programming language.
We only reviewed Object-oriented Programming languages and didn't add Object-oriented Database Management Systems (OODBMS) to the comparison because an OODBMS should be an extension to an Object-oriented programming language which mainly deals with the issue of persistent Objects. Therefore we included an overview of the commercial available OODBMS products we know and describes which object-oriented programming languages they support.

10.2 Relationships between OOPL's and the Methods

Table 10.1 The OOA/OOD method by Coad & Yourdon

Method Concept	**Smalltalk**	**Java**	**Delphi**	**C++**
Class	*	*	Object type	*
Object	*	*	*	*
Attribute	Instance variable	Instance variable	Data field	Data member
Subject	-	< Files	< Unit	< Files
Service	Method	Method	Procedure/ function	Member function
Instance connection	< Collection classes	Object identifiers	Type pointers	Pointers
Single inheritance	Super- subclass	Super-subclass	Ancestor-descendant	Derived classes
Multiple inheritance	-	-	-	Derived classes

Whole-Part relationship	< Collection classes	< Collection classes	Embedded pointers	Pointer/ embedded classes
Message connection	Message	Message	Procedure/ function call	Function calls
Constructor operation	< Creation and initial. methods	Creation and factory method	Creation and initial. procedure	*
Destructor operation	(Implicit)	>	-	*

Table 10.2 The DOOS method by Wirfs-Brock

Concept of method	**Smalltalk**	**Java**	**Delphi**	**C++**
Class	*	*	Object type	*
Object	*	*	*	*
Abstract Class	< Class	< Class	< Object type	< Class
Concrete Class	Class	Class	Object type	Class
Subsystem	-	> Files	> Unit	> Files
Public responsibility	Public method	Public method	Function/ procedure	Public member function
Private responsibility	Private method	Private method	-	Private member function
Contract	-	-	-	-
Single Inheritance	Super-subclass	Super-subclass	Ancestor-descendant	Derived classes
Multiple Inheritance	-	-	-	Derived classes
Collaboration	> Message	> Message	> Function procedure call	Function calls

Table 10.3 The OMT method by Rumbaugh

Concept of method	Smalltalk	Java	Delphi	C++
Class	*	*	Object type	*
Object	*	*	*	*
Abstract Class	< Class	< Class	< Object type	< Class
Concrete Class	Class	Class	Object type	Class
Metaclass	*	Posing Class	-	-
Module	-	Files	Unit	Files
Attribute	Instance variable	Instance variable	Data field	Data member
Operation	Method	Method	Procedure/ function	Member function
Abstract operation	-	-	-	< Virtual function
Single inheritance	Super-subclass	Super-subclass	Ancestor-descendant	Derived classes
Multiple inheritance	-	-	-	Derived classes
Whole-Part relationship	< Collection classes	< Collection classes	Embedded pointers	Pointers/ embedded classes
Association	< Collection classes	Object identifiers	Type pointers	Pointers
Concurrency	*	-	-	-

Table 10.4 The OOADA method by Booch

Concept of method	Smalltalk	Java	Delphi	C++
Class	*	*	Object type	*
Object	*	*	*	*

Abstract Class	< Class	< Class	< Object type	< Class
Concrete Class	Class	Class	Object type	Class
Metaclass	*	-	-	-
Parametrized class	-	-	-	*
Class Utility	-	*	*	*
Export control: Public/protected/ private/implemen tation	Public/ private	Public/ private	Public	*
Class persistence	-	-	-	-
Field	Instance variable	Instance variable	Data field	Data member
Operation	Method	Method	Procedure/ function	Member function
Operation exceptions	*	?	-	*
Message	Message	Message	Procedure/ function call	Function call
Module	-	Files	Unit	Files
Single inheritance	Super-subclass	Super-subclass	Ancestor-descendant	Derived classes
Multiple inheritance	-	-	-	Derived classes
Part-of relationship	< Collection classes	< Collection classes	Embedded pointers	Pointer/ embedded classes
Using relationship	< Collection classes	Object identifiers	Type pointers	Pointers
Instantiation relationship	*	?	-	*

Concurrency	*	-	-	-
Constructor operation	< Creation and initial. methods	Creation and factory method	Creation and initial. procedures	*
Destructor operation	(Implicit)	>	-	*

Table 10.5 The OL method by Shlaer and Mellor

Concept of method	**Smalltalk**	**Java**	**Delphi**	**C++**
Object	*	*	Object type	*
Class	*	*	Object type	*
Attribute	Instance variable	Instance variable	Data field	Data member
Published operation	Method	Method	Procedure/ Function	Member function
Deferred operation	Method	Method	Procedure/ Function	Member function
Module	-	Files	Unit	Files
Relationship	< Collection classes	Object Identifiers	Type pointers	Pointers
Single super/ subtype	< Super-subclass	< Super-subclass	< Ancestor-descendant	< Derived classes
Multiple super/subtype	-	-	-	Derived classes
Message	*	*	Procedure/ function call	Function call

Table 10.6 The OOAD method by Martin and Odell

Method Concept	**Smalltalk**	**Java**	**Delphi**	**C++**
Object Type/Class	Class	Class	Object type	Class

Object	*	*	*	*
Attribute	Instance variable	Instance variable	Data field	Data member
Contract	-	-	-	-
Operation/ function	Method	Method	Procedure/ function	Member function
Relation	< Collection classes	Object identifiers	Type pointers	Pointers
Single super/ subtype	Super- subclass	Super-subclass	Ancestor-descendant	Derived classes
Multiple super/ subtype	-	-	-	Derived classes
Composition relationship	< Collection classes	< Collection classes	Embedded pointers	Pointer/ embedded classes
Instantiation relationship	-	?	-	-
Concurrency	*	-	-	-
Event Type	-	-	-	-
Creation operation	< Creation and initial. methods	Creation and factory method	Creation and initial. procedure	*
Destruction operation	(Implicit)	>	-	*

Table 10.7 The Fusion method by Coleman

Method Concept	**Smalltalk**	**Java**	**Delphi**	**C++**
Class	Class	Class	Object type	Class
Object	*	*	*	*
Abstract Class	< Class	< Class	< Object type	< Class

Attribute	Instance variable	Instance variable	Data field	Data member
Message	Message	Message	Procedure/ function call	Function call
System operation	Method	Method	Procedure/ function	Member function
Relation	< Collection classes	Object identifiers	Type pointers	Pointers
Single super/ subtype	Super- subclass	Super-subclass	Ancestor-descendant	Derived classes
Multiple super/ subtype	-	-	-	Derived classes
Aggregation	< Collection classes	< Collection classes	Embedded pointers	Pointer/ embedded classes
Creation operation	< Creation and initial. methods	Creation and factory method	Creation and initial. procedure	*

Table 10.8 The OOSE method by Jacobson

Method Concept	**Smalltalk**	**Java**	**Delphi**	**C++**
Class	*	*	Object type	*
Abstract Class	< Class	< Class	< Object type	< Class
Concrete Class	Class	Class	Object type	Class
Object	*	*	*	*
Attribute	Instance variable	Instance variable	Data field	Data member
Actor	-	-	-	-
Use case	-	-	-	-
Stimulus	Message	Message	Procedure/ function call	Function call

Contract	-	-	-	-
Block	-	-	-	-
Object module	-	Files	Unit	Files
Operation	Method	Method	Procedure/ function	Member function
Association	< Collection classes	Object identifiers	Type pointers	Pointers
Single super/ subtype	Super- subclass	Super-subclass	Ancestor-descendant	Derived classes
Multiple super/ subtype	-	-	-	Derived classes
Extension association	-	-	-	-
Consists-of association	< Collection classes	< Collection classes	Embedded pointers	Pointer/ embedded classes
Acquaintance association	-	-	-	-
Concurrency	*	-	-	-
Creation operation	< Creation and initial. methods	Creation and factory method	Creation and initial. procedure	*
Removing operation	(Implicit)	>	-	*

10.3 Relationships between OOPL's and Object-Oriented Database Management Systems

Table 10.9 Object-Oriented Database Management Systems

Product Name	Database Vendor	Object-oriented languages they Support
ObjectStore	Object Design	C++, Java
Versant	Versant Object Technology	C++, Smalltalk
ONTOS	ONTOS	C++, Java
Objectivity/DB	Objectivity	C++
GemStone	Servio Logic	C++, Smalltalk
OpenODB	Hewlett-Packard	C++, Smalltalk
ITASCA	Itasca Systems	C++, Common Lisp, Delphi
GBase	Object Databases Corp.	C++
O2	O2 Technology	C++
ArtBASE	ArtInAppleS	Smalltalk
EasyDB	Objective Systems	C++, Java
NeoAccess	NeoLogic	C++
POET	BKS Software	C++, Java
UNISQL/X	UNISQL Inc.	C++, Smalltalk

CHAPTER ELEVEN
CONCLUSION

In conclusion, object-oriented analysis and design techniques for object-oriented database management systems are rapidly evolving, but the field is by no means fully mature. None of the techniques reviewed here has achieved the status of a widely recognized standard on the order of the conventional techniques of Yourdon and Constantine or DeMarco. Object-oriented techniques for object-oriented database management systems will continue to evolve, as did conventional techniques before them, as subtler issues emerge from their use in a wide array of problem domains and project environments. Three areas-system partition-ing, end-to-end process modeling, and harvesting reuse - appear to be especially strong candidates for further object-oriented development work. In the meantime, adopters of current object-oriented techniques may need to develop their own extensions to contend with these issues, or alternatively, limit application of the techniques to problem domains where these issues are of lesser importance.

During analysis and design, all conventional techniques revert to a process-oriented view in establishing the architecture of program modules, and as a result, object orientation will likely be viewed as radical change by developers schooled in any of the conventional design methods. Since organizations will have to adopt object-oriented design methodologies to end up with object-oriented implementations. a move to an object-oriented environment in general may be seen predominantly as a radical change.

Object orientation is founded on a collection of powerful ideas; modularity, abstraction, encapsulation, reuse that have firm theoretical foundations. In addition, trends in computing towards complex data types and complex new forms of integrated systems seem to favor the object model over conventional approaches for object-oriented database management systems. Although little empirical evidence exists to support many of the specific claims made in favor of object orientation, the weight of informed opinion among many leading-edge practitioners and academics

favors object orientation as a “better idea” than conventional approaches for database development. Organizations that are able to absorb this radical change may well find themselves in a significantly stronger competitive position vis-a-vis those incapable of making the transition.

REFERENCES

Arnold, P., Bodoff, S., Coleman, D., Gilchrist, H., & Hayes, F. (1998). *An evaluation of five object-oriented development methods* (4th ed.). NY: HP Laboratories Press.

Batini, C., Ceri, S., & Navathe, S. (2001). *Conceptual database design : An entity-relationship approach* (3rd ed.). California: The Benjamin/Cummings Publishing Company Inc.

Booch, G. (1997). *Object-oriented analysis and design with applications* (2nd ed.). California: The Benjamin/Cummings Publishing Company Inc.

Budd, T. (1999). *An introduction to object-oriented programming* (3rd ed.). Massachusetts: Addison-Wesley.

Coad, P., & Yourdon, E. (1999). *Object-oriented analysis* (4th ed.). Retrieved Novamber 12, 2004 from http://www.yp.cmu/str/descriptions/ooanalysis.pdf.

Coad, P., & Yourdon, E. (1999). *Object-oriented design* (3rd ed.). Retrieved Novamber 10, 2004 from http://www.yp.cmu/str/descriptions/oodesign.pdf.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., & Jeremaes, P. (1994). *Object-oriented development: the fusion method.* New Jersey: Prentice-Hall.

Goor, G.P.M., Brinkkemper, S., & Hong, S. (2002). *A comparison of six object-oriented analysis and design methods* (2nd ed.). New Jersey: Yourdon Press.

Harmsen, F., & Brinkkemper, S. (2004). *Description and manipulation of method fragments for the assembly of situational methods* (3rd ed.). Sydney: Prentice-Hall.

Henderson-Sellers, B., & Edwards, J.M. (1994). *Book two of object-oriented knowledge: the working object.* Sydney: Prentice-Hall.

Jacobson, I., Christerson, M., Jonsson, P., & Övergaard, G. (1992). *Object-oriented software engineering*. NY: Addison-Wesley.

Martin, J., & Odell, J. (2003). *Object-oriented analysis and design* (4th ed.). New Jersey: Prentice-Hall.

Martin, J. (2003). *Principles of object-oriented analysis and design* (4th ed.). New Jersey: Prentice-Hall.

Pappas, C.H., & Murray, W.H. (1991). *Borland C++ handbook.* California: McGraw-Hill, Berkeley.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1997). *Object-oriented modeling and design* (2nd ed.). New Jersey: Prentice-Hall.

Shlaer, S., & Mellor, S.J. (2002). *Object-oriented systems analysis: modeling the world in data.* New Jersey: Yourdon Press.

Stroustrup, B. (2001). *The C++ programming language* (4th ed.). Massachusetts : Addison-Wesley, Reading.

Wirfs-Brock, R., Wilkerson, B., & Wiener, L. (2000). *Designing object-oriented software.* New Jersey: Prentice-Hall.

Yourdon, E. (1999). *Modern structured analysis*. New Jersey: Prentice-Hall.