## An Introduction to UML

# The Logical Model

by Geoffrey Sparks

*All material (c) Geoffrey Sparks 2000*

www.sparxsystems.com.au

# Table of Contents

**THE LOGICAL MODEL ...... 3**

INTRODUCTION TO UML ...... 3
THE LOGICAL MODEL ...... 3
*Objects and Classes ...... 3*
*Class Features ...... 4*
*Interfaces ...... 7*
*Relationships ...... 8*
*Constraints, Requirements, States and Scenarios ...... 10*
PUTTING IT TOGETHER ...... 11
*Example Class Diagram ...... 11*
*Example State Chart ...... 13*
*Packages ...... 14*
*Recommended Reading ...... 15*

© Geoffrey Sparks 2000

# The Logical Model

The Logical Model in UML is used to model the static structural elements. It captures and defines the objects, entities and building blocks of a system. Classes are the generalised templates from which run-time objects are created. Components (discussed in The Component Model) are built from classes. Classes (and interfaces) are the design elements that correspond to the coded or developed software artefacts. This paper will describe some features of the class model, look at the UML notation for describing classes/objects and give an example of the notation's usage.

## *Introduction to UML*

The Unified Modelling Language (UML) is, as its name implies, a modelling language and not a method or process. UML is made up of a very specific notation and the related grammatical rules for constructing software models. UML in itself does not proscribe or advise on how to use that notation in a software development process or as part of an object-oriented design methodology.

UML supports a rich set of graphical notation elements. It describes the notation for classes, components, nodes, activities, work flow, Logicals, objects, states and how to model relationships between these elements. UML also supports the notion of custom extensions through stereotyped elements.

The UML provides significant benefits to software engineers and organisations by helping to build rigorous, traceable and maintainable models, which support the full software development lifecycle.

This paper focuses on the logical or static model.

You can find out more about UML from the books mentioned in the suggested reading section and from the UML specification documents to be found at the Object Management Groups UML resource pages: http://www.omg.org/technology/uml/ and at http://www.omg.org/technology/documents/formal/

## *The Logical Model*

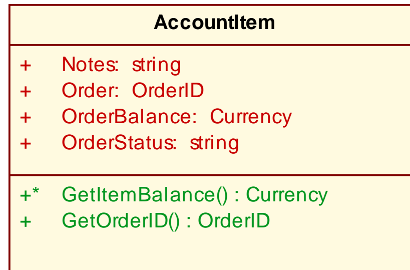
A logical model is a static view of the objects and classes that make up the design/analysis space. Typically, a Domain Model is a looser, high level view of Business Objects and entities, while the Class Model is a more rigorous and design focused model. This discussion relates mainly to the Class Model.

### Objects and Classes

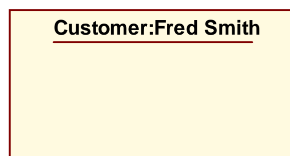
A Class is a standard UML construct used to detail the pattern from which objects will be produced at run-time. A class is a specification - an object an instance of a class. Classes may be inherited from other classes (that is they inherit all the behaviour and state of their parent and add new functionality of their own), have other classes as attributes, delegate responsibilities to other classes and implement abstract interfaces. The Class Model is at the core of object-oriented development and design - it expresses both the persistent state of the system and the behaviour of the system. A class encapsulates state (attributes) and offers services to manipulate that state (behaviour). Good object-oriented design limits direct access to class attributes and offers services which manipulate attributes on behalf of the caller. This hiding of data and exposing of services ensures data updates are only done in one place and according to specific rules

- for large systems the maintenance burden of code which has direct access to data elements in many places is extremely high.

Classes are represented using the following notation:




And objects with the notation below:

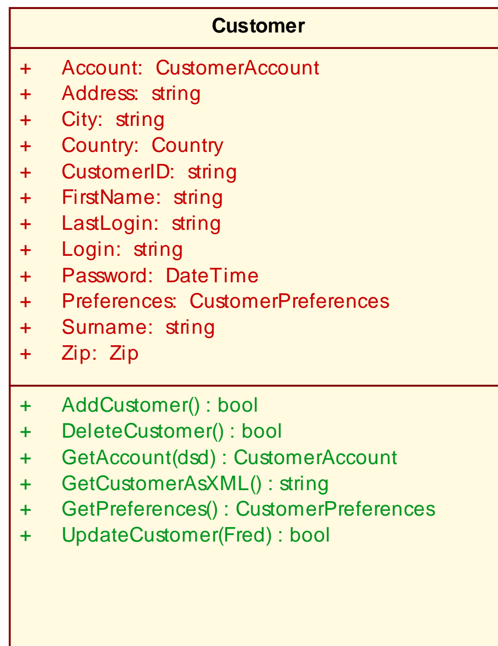



## Class Features

Note that the class has three distinct areas:

1. The class name (and stereotype if applied)
2. The class attributes area (that is internal data elements)
3. The behaviour - both private and public

The following image shows these three areas (Attributes in red, Operations in green):



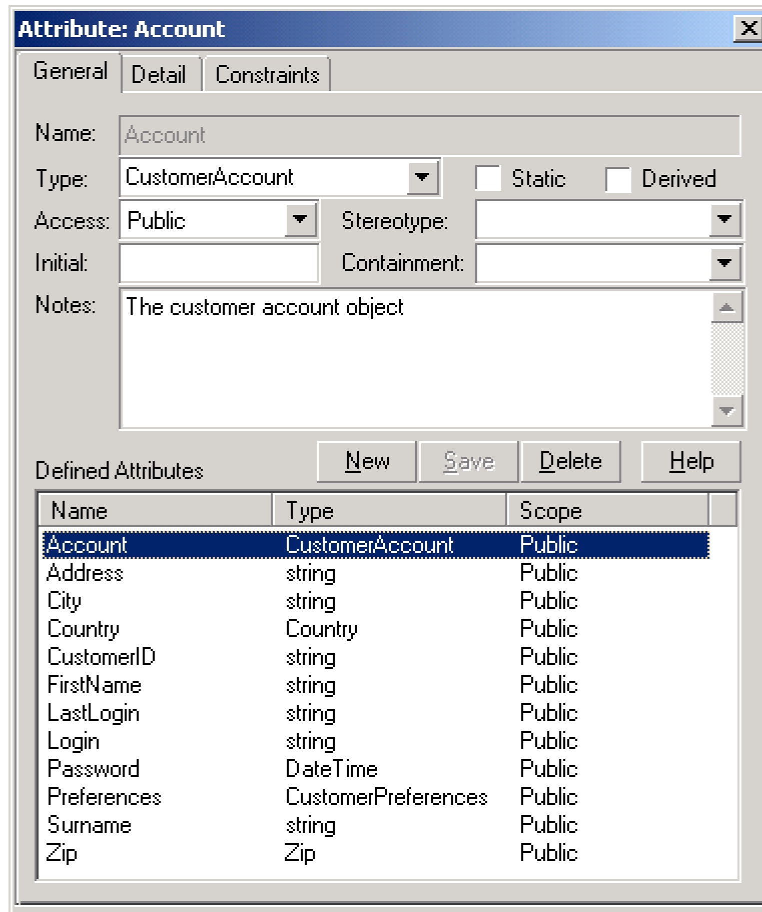

Attributes and methods may be marked as:

- Private, indicating they are not visible to callers outside the class
- Protected, they are only visible to children of the class
- Public, they are visible to all

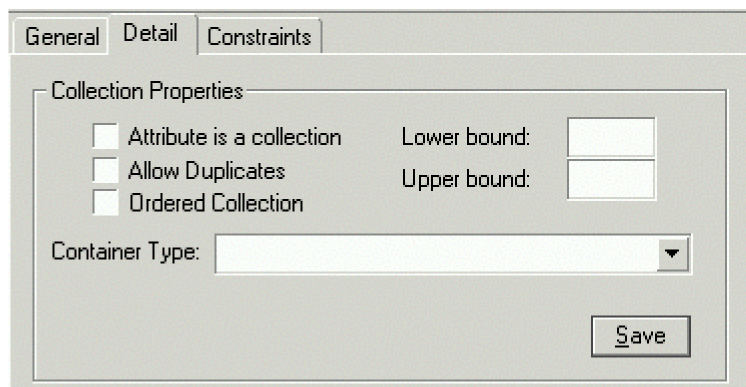
The following textual conventions apply to the notation:

| Symbol | Meaning |
|---|---|
| + | Public |
| - | Private |
| # | Protected |
| $ | Static |
| / | Derived (attribute - non standard) |
| * | Abstract (operation - non standard) |

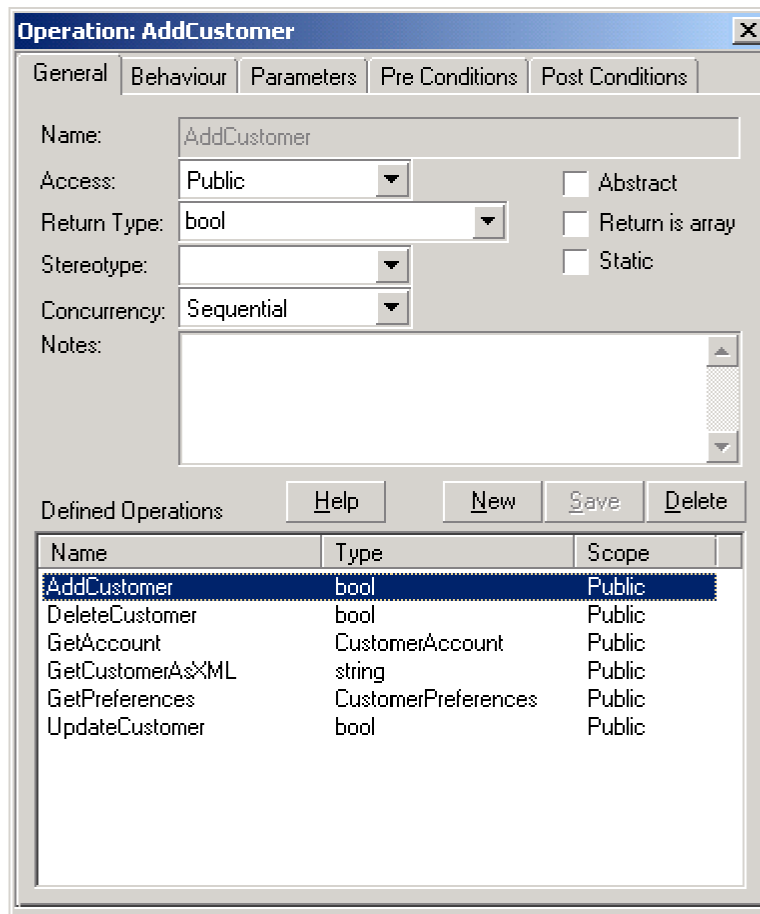
A CASE tool will allow you to set numerous attribute details, as in the example below:




And some additional details:

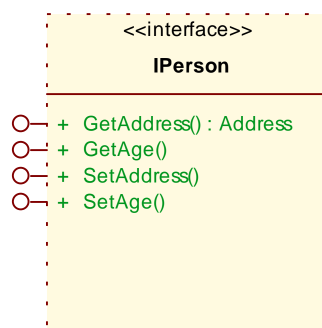


And likewise for class operations:




## Interfaces

Interfaces are abstract classes with a behavioural component only. They specify an operational contract that an implementor agrees to meet. For example, an interface may be defined called IPerson with methods to access Address and Age as in the example below:




This interface cannot be instantiated at run time - it is an abstract element. Instead, a class may be defined which 'implements' this interface. This class is then committed to

providing implementations of the interface methods with the exact signature as specified in the Interface definition. This allows many classes in different hierarchies with different parents and behaviour to all be treated as an IPerson type (if they implement this interface) at run time.

## Relationships

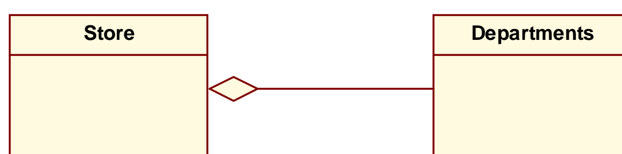
Logical elements may be related in a variety of ways. The following are the most common:

### Association

Association relationships capture the static relationships between entities. These generally relate to one object having an instance of another as an attribute or being related in the sense of owning (but not being composed of). For example a Salesperson has an association to their customers (as in the image below), but a salesperson is not 'composed of' Customers. Both ends of the association link may assume a 'role' as part of the association - for example the Salesperson may assume the role of 'Carpet Salesperson' if there are special conditions that apply to that role.

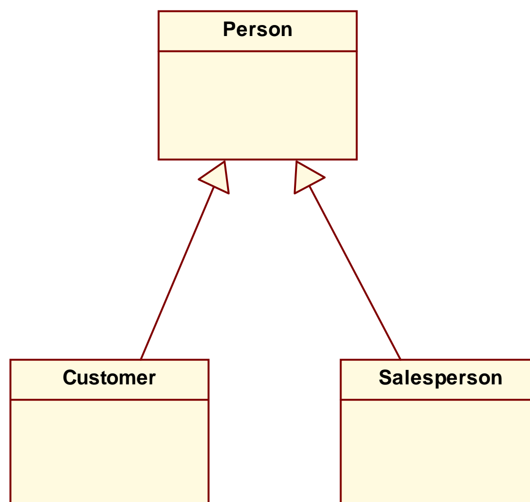
### Aggregation

Aggregation relationships define whole/part relationships. For example, a train journey may be composed of many journey legs. The overall journey then aggregates or is composed of the separate legs. The stronger form of aggregation is composition and infers that a class not only collects another class, but is actually composed of that collection. The example below indicates that a Store aggregates or collects together a number of Departments and that there is a whole/part relationship - that is the store is made up of (in part) this collection of departments.



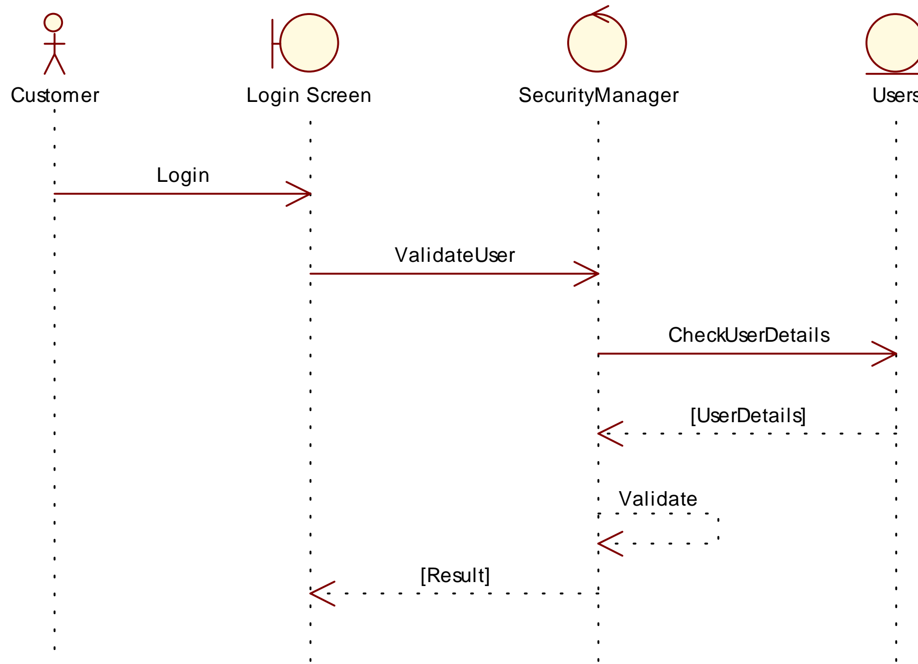
### Inheritance

Inheritance describes the hierarchical relationship between classes. It describes the 'family tree'. Classes may inherit attributes and behaviour from a parent class (which may in turn be the child of another class). This tree of inherited characteristics and behaviour allows the designer the ability to collect common functionality in root classes (ancestors) and refine and specialise that behaviour over one or more generations (children). Scope specifiers (public, protected and private) determine which elements may be inherited and which are not visible. The following diagram illustrates one level of inheritance:




### Dynamic

Dynamic relationships are the messages passed between classes (objects at run time). A Sequence Diagram illustrates this message passing and the sequence in which it occurs. The image below shows the sequence of messages passed over time when a user logs on to an on-line book store. These model elements have an association to each other reflected at run time by the passing of messages to each other.




## Constraints, Requirements, States and Scenarios

A logical model element and the attributes, associations and operations that it has may all be further specified with constraints, requirements and scenarios.

**Constraints**

These are the contractual rules that apply to an element and/or its features. Typically they fall into one of three types:

1. Pre-conditions - which must be true prior to the operation or existence of an element;
2. Post-conditions - which must be true after the use or destruction of an object
3. Invariants - which must always be true for the life and use of an object

**Requirements**

These specify what it is the class should be capable of doing, that is what information is persisted, what behaviour is supported and what relationships are required. For example, a person class may require the storage of Name, Age, Sex, Birth date etc.

**States**

Classes may be viewed as having different states over time (eg. A Customer is Browsing, Selecting, Buying & etc.). UML allows the designer to model these states using State Charts. These define the legal states an object may be in together with the transitions, conditions that initiate transitions and guard conditions that prevent transitions if not met.

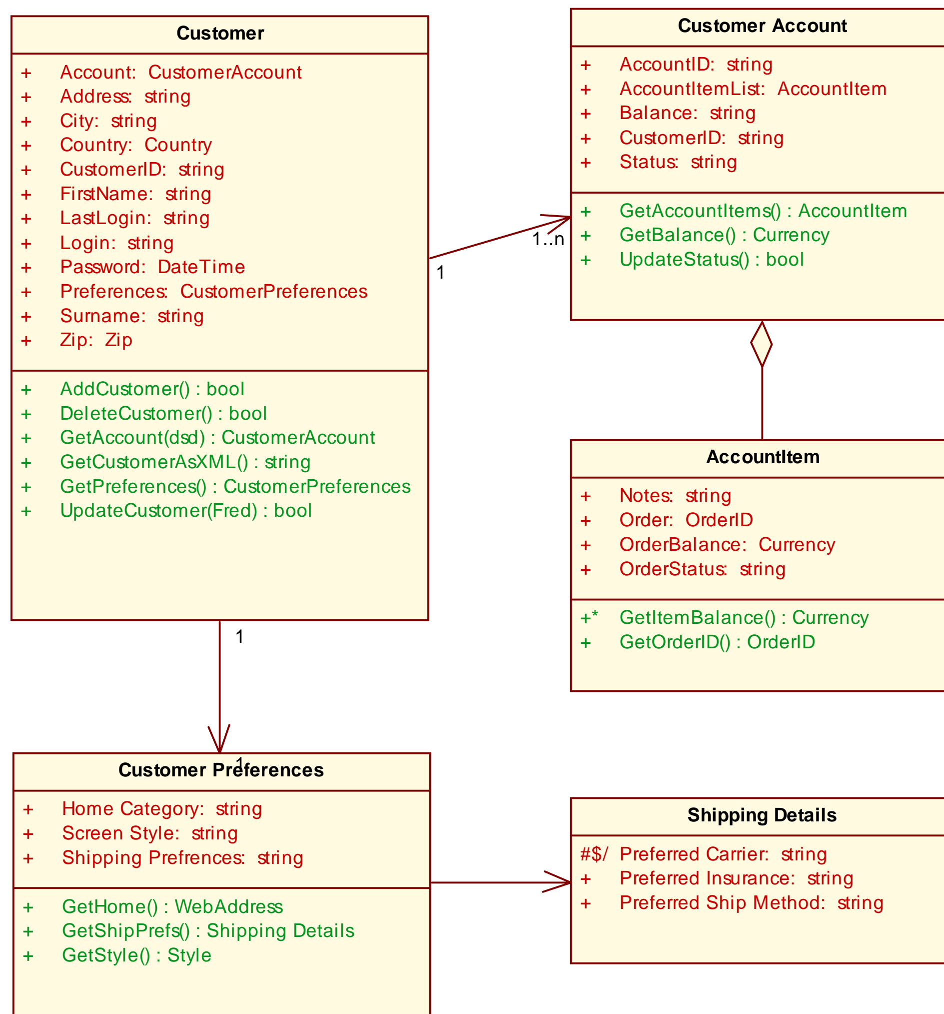
### Scenarios

Scenarios are descriptive textual (or diagrammatic) representations of how a class is used over time in a collaborative manner.

# Putting it together

## Example Class Diagram

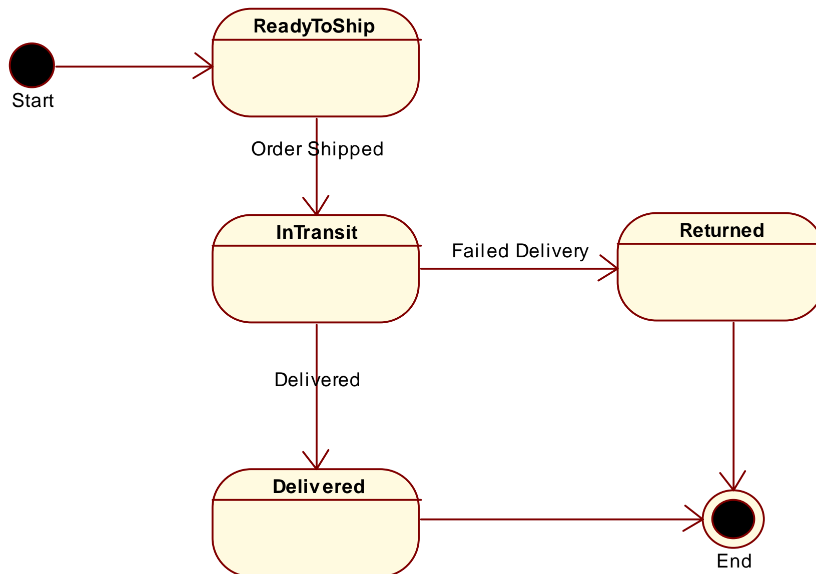
The following examples illustrate how the notation is used to design a static model. The first diagram shows the relationships between a customer and their Account and Account Items. The Customer preferences are also included. The notation reveals that a Customer may have one or more accounts and that accounts are composed of one or more Account Items. The attributes (or data) associated with each class is detailed (in red) and the behaviour supported (in green).



This second example illustrates similar elements, including an inheritance link from Publisher to Company (indicating that a Publisher is a kind of Company).

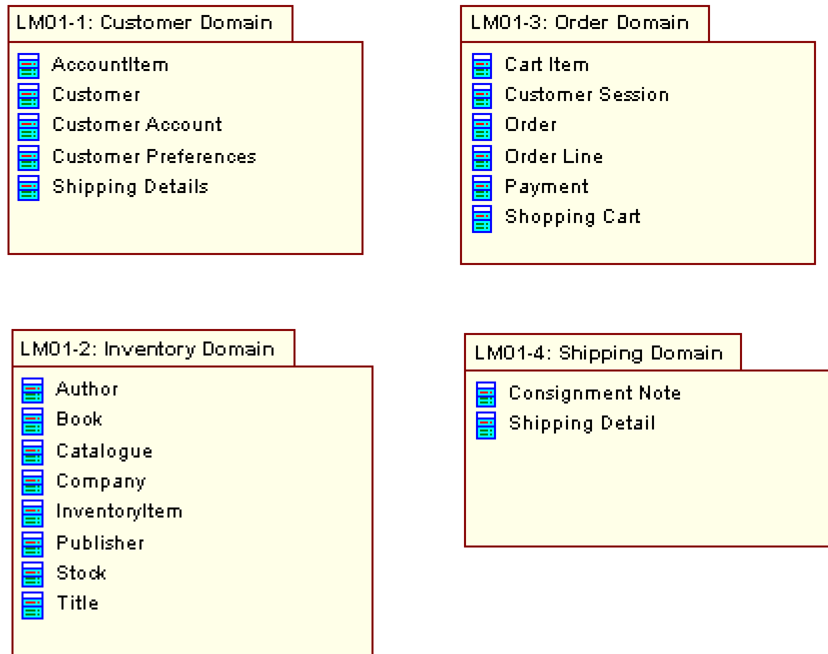
## Example State Chart

The diagram below illustrates the states an order being shipped may go through - from being ready to ship, in transit, delivered or returned.



## Packages

As the class model develops, classes (and objects and interfaces) may be organised into logical units or packages. These collect related elements (and in some implementations will govern the visibility of operations (and attributes) by elements outside the package. The diagram below illustrates the collection of classes into packages:



## Recommended Reading

Sinan Si Alhir, ***UML in a NutShell***.
ISBN: 1-56592-448-7. Publisher: O'Reilly & Associates, Inc

Doug Rosenberg with Kendall Scott ,***Logical Driven Object Modeling with UML.***
ISBN:0-201-43289-7. Publisher: Addison-Wesley

Geri Scheider, Jason P. Winters, ***Applying Logicals***
ISBN: 0-201-30981-5. Publisher: Addison-Wesley

Ivar Jacobson, Martin Griss, Patrik Jonsson, ***Software Reuse***
ISBN:0-201-92476-5. Publisher: Addison-Wesley

Hans-Erik Eriksson, Magnus Penker, ***Business Modeling with UML***
ISBN: 0-471-29551-5. Publisher: John Wiley & Son, Inc

Peter Herzum, Oliver Sims, ***Business Component Factory***
ISBN: 0-471-32760-3 Publisher: John Wiley & Son, Inc