Software Quality Journal, 13, 31–69, 2005

Towards Formalised Guidelines for Migrating Structured Designs to UML: A Case Study

ELLI GEORGIADOU e.georgiadou@mdx.ac.uk
Middlesex University, School of Computing Science, Trent Park, Bramley Road, London N14 4YZ, UK

ELENI BERKI eleni.berki@cc.jyu.fi
University of Jyväskylä, Department of Computer Science and Information Systems, P.O. Box 35 (Agora), FIN 40014 Jyväskylä, Finland

MARIA DEL BREZO CORDERO
Valladolid University, Faculty of Science, Higher Technical School of Computer Engineering, Department of Informatics, Valladolid, Spain

MARGARET ROSS and GEOFF STAPLES margaret.ross@solent.ac.uk
Southampton Institute, Faculty of Systems Engineering, East Park Terrace, Southampton, Hampshire SO14 0YN, UK

Abstract. This paper provides evidence that it is possible to migrate structured system designs to UML models. Legacy structured designs can be converted to object-oriented systems without losing data or functionality. The reason for choosing UML is that it is the new standard notation, which tries to build on and integrate object-oriented notations. Extracts from a case study are presented together with the proposed guidelines for the re-engineering process.

Keywords: re-engineering, structured designs, OO, UML, isomorphic models, reuse, testing

1. Introduction

Over the last 40 years a plethora of information systems development methodologies have been developed and adopted aiming to address the issues of managing the software development process. Addressing the issue of software failures has resulted in the development of thousands of innovative approaches to building software, from new technologies to progressive processes and frameworks (Avison and Fitzgerald, 1995; Jackson, 1994; Jayaratna, 1994). With such immense variety of methods and tools the selection of an appropriate methodology has been the subject of debate, argument, investigation by researchers and practitioners alike for the last 30 years (Law, 1998; Law and Naeem, 1992; Georgiadou and Sadler, 1995; Jayaratna, 1994; Berki et al., 1997; Manninen and Berki, 2004).

According to Wilkie (1993) "Structured methods make a division between data, process and behaviour, highlighting one of them in the function of the method used. In information systems, data are emphasised and the analysis of process requirements takes second place. On the other hand, in real-time applications, processing is the highest priority and data associated are laid back."

Object orientation brought the promise of a single, unified approach, joining data and processes. It brings coherence and integration to the software development process. Object-orientation has always been associated with particular programming languages, such us Smalltalk, C++ and Java. However, OO is a philosophy, a way of organising our thoughts. It does not have to be related to software systems, it can describe every type of system and every type of operation.

The advent of the millennium bug proved that legacy systems can engender problems some of which manifest themselves many years down the line (Ross et al., 1997). Many legacy systems, especially in the government and corporate sectors have been designed using structured methods. SSADM, for example, has been the government standard within the UK for 20 years, Yourdon and Information Engineering have been widely used across the USA and MERISE has been the French standard.

The development of OO technologies and methods, and more recently Component Based Development marked a paradigm shift (Allen and Frost, 1998). Although new systems tend to be developed using this paradigm, existing systems need continuous maintenance. The question is whether it is advisable to maintain structured designs or to re-engineer them into equivalent OO designs. In order to answer this question we need to not only make a management decision as to the financial viability of the exercise but most importantly we need to ensure that re-engineering will not result in loss of either data or functionality. Using a standard notation makes the design easy to communicate, reuse and optimise. Designers should be willing to carry out this migration in order to give consistency and uniformity to the designs.

2. Re-engineering for reuse

Reuse and design are facets of the same activity. When commencing a new project, available reusable resources in terms of libraries of object classes could be identified, and then reused either directly or by means of inheritance. The development should be easier when tested stable components are already available.

The representation of the design information should allow the design to be captured in a form that is machine processable, and the components to be smoothly integrated. Software development tools should control the management of this development process enabling application development top down through to code generation, and bottom up to include and provide for re-engineering of existing code. In order to achieve this it has to be fully integrated with the development environment.

Henderson-Sellers et al. (1999) illustrated three alternative routes to achieving the target software system. Evidently developing from scratch involves minimal reuse relating to systems level resources. The other two options require search, identification and matching of existing components in terms of functionality. If the aim is to maximise productivity and minimise costs more studies are required to at least demonstrate the benefits of such a decision (Ezran et al., 2002). There are clear benefits in terms of reliability particularly due to the improvement in testing assembled/integrated

systems made from already tested, well behaved, predicable and controllable components.

By far the most frequently used route is the re-engineering/restructuring of existing code which has been successfully carried out for accommodating changes of programming language and/or migration to new environment. Reuse of designs has been the focus of some research by several researchers including Sutcliffe and Carroll (1999) and Mohamed-Bakry (1999).

3. Migrating from a structured designs to UML

3.1. A case study from Yourdon

Brezo (1998) provided an example of converting a structured design to an object-oriented design in UML. The reason for choosing the Yourdon Press case study was not only that it is an exponent of structured design but also that it is completely developed. It consists on an introduction with the specifications and requirements for the system, followed by the environmental model and the behavioural model. Additional development can easily be added due to the modular and incremental development allowed by UML.

Appendix A gives details of the migration method and presents all the designs and their UML equivalent. In the following sections we present part of the original case study (the complete environmental model, the first Data Flow Diagram (DFD), some of the DFD's and some part of the data dictionary when needed. The rest of the orginal system can be found in (Yourdon, 1998).

3.2. Modelling static and dynamic aspects

Structured Techniques with the exception of Entity Life Histories model static aspects of systems. Berki and Georgiadou (1996) discussed the deficiencies of DFDs and used Finite State Machines to re-model systems including dynamic aspects.

The behavioural aspects of the system can be shown in UML by means of several diagrams:

Use case diagrams. These have already been applied them (see appendix). They give the capability of establishing that the new system provides the same services than the previous one. They can be a contract between the software engineer and the client.

Interaction diagrams. They address an interaction and may be used to model flows within use cases (Booch et al., 1999). They show how the objects interact to carry out operations, these diagrams will be the proof that the system does what is supposed to be done.

There are two types of interaction diagrams, namely sequence diagrams, with the emphasis on the time ordering of the messages and collaboration diagrams, with the emphasis on the structured organisation of the objects that send and receive the messages.

The operations in the process specifications of the Yourdon Press model were converted into messages, operation calls, among or within the objects involved in the use case. The authors traced the DFD's and demonstrated that there is no missing functionality.

Through this process the authors found new operations that the objects must support. These were included in the class diagram. Modeling the dynamic behaviour is also an analysis technique. It is difficult to determine every object service when defining the static structure of the system. Additional attributes and operations were found. Through this incremental approach the authors gained further understanding of the system. Finally the new models were validated using interaction diagrams.

4. Guidelines for the migration of structured designs to UML

4.1. A step-by-step re-engineering process

Migrating from a structured design to an object-oriented design is not an easy task. It must be borne in mind that there must not be loss of data or functionality. The new system has to provide the same services and hold the same information as the legacy system.

The insights gained from the case study presented here (Appendix A) assisted the authors in generating a set of guidelines which can guide and inform the re-engineering process. At each step the deliverables are indicated and the software engineer is alerted to the pitfalls.

Step 1. Identify one service provided by the system and transform it to a use case. Services can be extracted from the statement of purpose. They are broken into smaller units. Use cases with small level of granularity are more likely to be reused than the ones with bigger level. They are also easier to model and provide a good structure for component-based development.

Step 2. Identify DFD's related to the use case. The event triggering the use case can be extracted from the event list. Follow this event through the Behavioural Model. Highlighting the processes is good practice to focus attention and to have a record of the parts of the system that have already been modelled.

Step 3. Describe the use case tracing the DFDs. Textual descriptions will help us to understand how the system works. When describing a use case, extract from the data flow diagrams the behaviour that is related to the actual use case. Also find optional flows that can be transformed into extending use cases.

Step 4. Trace the DFD's to create the class diagrams:

(a) Use two different models:

(i) Potential model, to which every feature identified is added, even though it is not related to the actual use case. It will prevent data or functionality being forgotten.

(ii) Accepted model, to which the features related to the actual use case are transfer from the potential model.

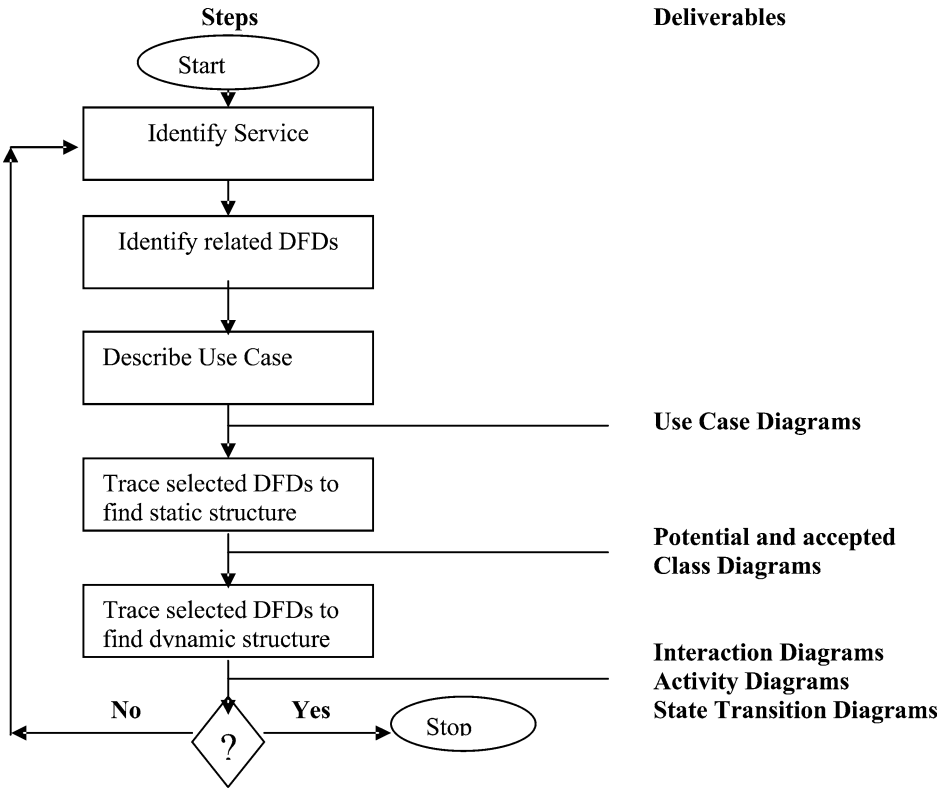


Figure 1. A schematic representation of the guidelines.

(b) Identify classes using language analysis. Searching for nouns in the process specifications will provide a list of potential classes, especially if they are an entity in the Entity-Relationship diagram.
(c) Apply patterns between them. Select the classes if they conform to patterns. They also help to identify the relationships among them. Also find any new classes that may have been confused with attributes.
(d) If the classes have related data dictionary entries, add the attributes to the potential mode. This is to preserve all the data from the structured model. The potential model will be larger than the accepted model in the first use cases but they will be similar in the end because the final model has to hold all the data, as information must not be lost.
(e) Trace the DFD; transfer the attributes found on it to the accepted model.
(f) Convert the functions to operations in the classes so that the new system will be able to perform the service.

Step 5. Use the interaction diagrams to check that the re-designed system behaves in the same way as the legacy system. Place the DFD's related beside the diagram and highlight the portion that is fulfilled in the diagram. Ensure that the model does not lose functionality and keep a record of the part of the system that

has been modelled. New classes, attributes and operations will probably be find.

Step 6. Use the activity diagrams to show concurrency and the state diagrams to show the change of status.

Step 7. Go back to the first step and apply incremental modeling untill all the services are covered.

The new model should then be checked to ensure that:

- Every data dictionary entry should match an attribute or a class in our system.
- Every function should match and operation or several of them.
- Every DFD should be able to be shown by means of interaction diagrams.

4.2. A schematic representation of the guidelines

The guidelines listed in Section 4.1 are presented in Figure 1.

5. Conclusions and further work

5.1. Problems and solutions

This case study has demonstrated that one can ensure that there is no loss of data or functionality but it does not mean that this is a good design. Object-oriented modeling requires a lot of experience and knowledge of the system. Ideally it would be helpful if one could interview the former designer but often they are not around and even if they are they do not necessarily remember the necessary details. To create a good design one can use different techniques from different methods. The use of patterns is recommended because patterns have proved to be good modeling ideas as they make systems reusable and more reliant. Patterns also help to organise and identify the classes. Partitioning the system is always good practice, especially for large systems. Packages make the models easy to validate or communicate and show dependencies between classes.

UML is a powerful notation but it is not a technique, so a study of object-oriented design is needed to apply it. The creators of the standard oriented language have also created a method to apply it, "The Unified Method" which can help developers take advantage of the features in the language.

The proposed guidelines can be applied to legacy systems, also known as legacy assets. "A legacy asset is a software product, developed on the basis of older technologies, which is past its best but that it can not be replaced or disrupted without major impact on the enterprise" (Allen and Frost, 1998). Re-engineering is a risky task.

This set of guidelines for migrating from structured designs to UML notations is an efficient way to structure the designers thinking and actions.

5.2. Related work

Some enterprises will not be willing to change their old system or to invest in it. Alternative ways of upgrading, instead of changing a system is the use of wrappers; hiding its structured design using a wrapper. "A wrapper is a component that provides an object-oriented interface to non object-oriented software" (Allen and Frost, 1998).

However, re-engineering without testing the systems components and without taking into account future changes might be an obstacle, especially when the designer has to deal with safety critical systems, where security and accuracy need to be certified. UML notations do not provide testability and formal testing procedures are essential to be built in the design process. In this case, there are more abstract modeling (Berki and Georgiadou, 1996) and metamodeling techniques and tools (Berki, 2001), which provide testedness and more abstraction. These facilitate the design process in different metadesign levels and the testing of the systems specification itself.

In order to improve the quality of design specifications and subsequent implementations, a generic approach is often needed to guide understanding and to support both method specialisation and generalisation. This should focus on defining systems components' interaction and, in parallel, on projecting the system's properties expressively, and in an integrated and testable (for method reuse) way.

It is also important to test and establish in a scientific manner whether the adoption of a specific design method will be able to model static, dynamic and computational requirements for a system and to which degree. This is a question that can be answered only if one could decide on the method's elements during and after their use by viewing them at a different level of abstraction. That is, the use of an effective and suitable metamodel would allow one to reason about the method construction itself and finally test, refine and re-engineer the method's artefacts.

These organised sets of guidelines are used as frameworks or metamodels in order to provide a rigorous environment for reasoning and formally studying the properties of software design artefacts. For instance, finite state and transition diagrams were proposed as an alternative and *isomorphic computational model* to some techniques of Structured, Semi-formal and Object-Oriented methods (Berki and Georgiadou, 1999) It was also proposed to use possible extensions to existing methods in order to ensure the isomorphic nature of models and apply them in a number of case studies (Berki, 2001). Metamodeling using isomorphic models deals with the examination of the equivalence of models produced in the traditional way and their equivalent formal representations.

Finally this case study provides evidence and demonstrates by examples that it is possible to convert a well-documented Yourdon design into UML notation. The proposed set of guidelines aims to assist in the migration process. can be very useful to convert the plethora of legacy systems, that are developed with structured methods, to more dynamic and integrated UML notations.

Future work will concentrate on integrating the strengths of CDM FILTERS (Berki, 2001) with the practical knowhow derived from extensive use of the proposed guidelines for the production of automated CASE tools. It is aimed to automate the extraction of design information from existing models for succesful re-engineering of legacy structured designs to UML models.

Appendix A. Proposed guidelines applied to a case study

A.1. The environmental model

A.1.1. The statement of purpose and a context diagram The purpose of Yourdon Press Information System (YPIS) is to maintain information needed to sell books to customers. This includes order entry, invoicing, generation of shipping documents, inventory control, and production of royalty reports and accounting reports. Figure A1 is the context diagram (Yourdon, 1988).

A.1.2. The event list The event list for YPIS consists of 40 events. Most of the events are flow-driven, though most of the events involving the Accounting Department are temporal. A list of events includes:

Customer orders book (this includes special rush orders).
Customer sends payment.
Customer asks for book information (price, etc.).
Customer asks permission to return a book.
Customer asks about status of a book order.
⋮

A.2. The behavioural model

Figure A2 depicts the behavioural model.

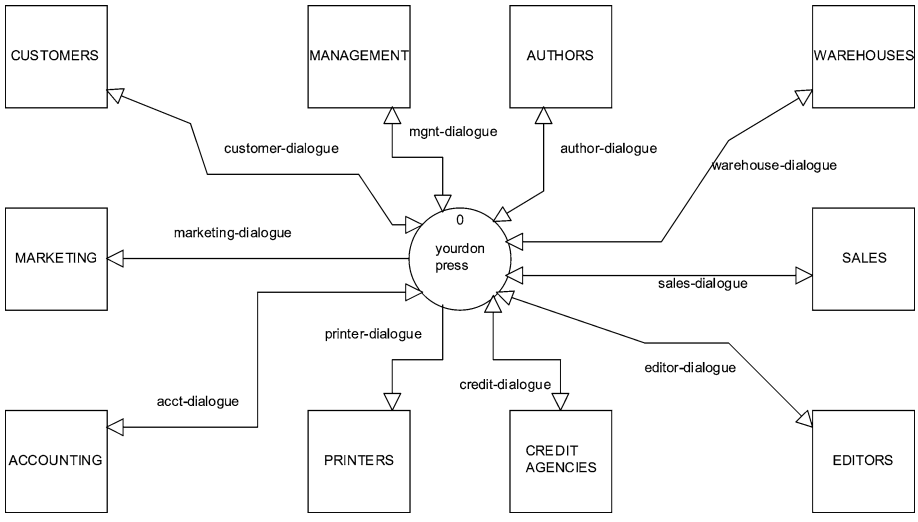


Figure A1. The Yourdon Press information system (Yourdon, 1988).

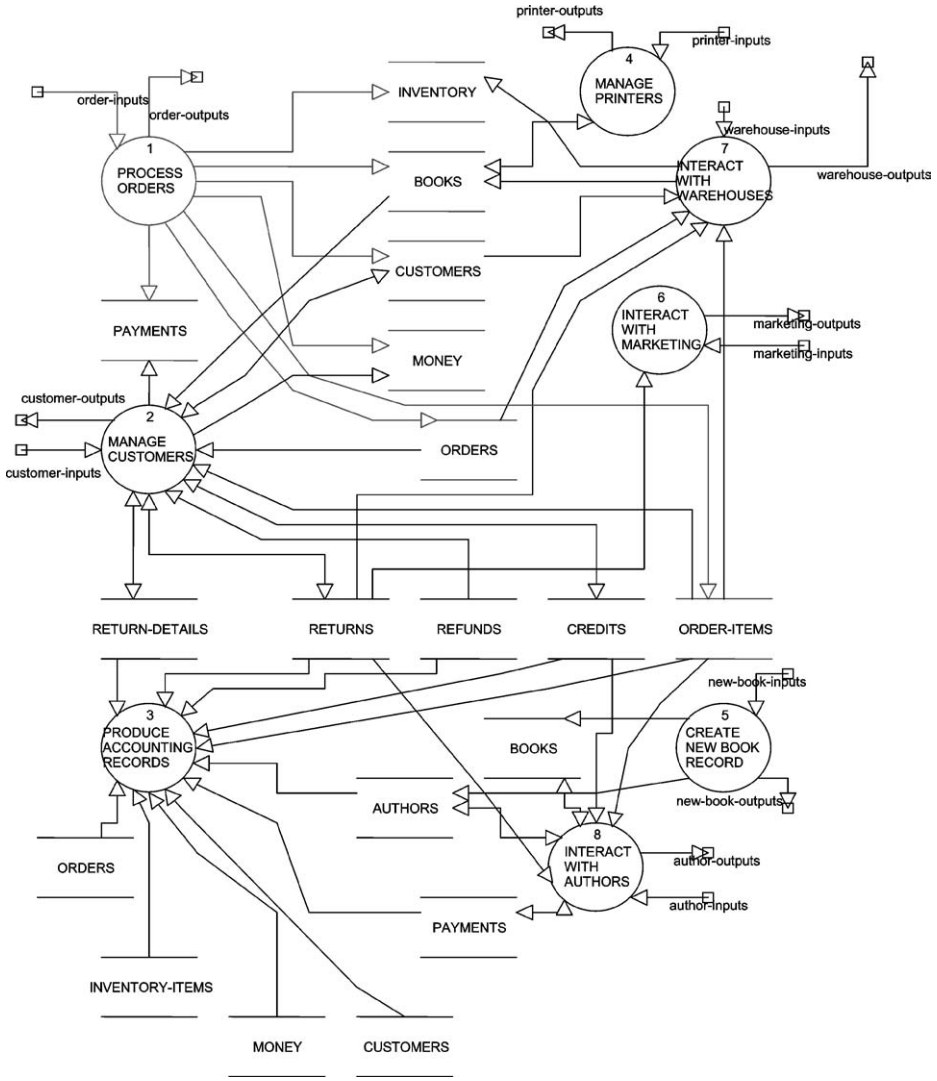


Figure A2. DFD1 for YPIS.

A.3. Re-designing

A.3.1. Selecting a use case In the Yourdon method, the functionality of the system is expressed in the statement of purpose, and the boundaries of the system are visualised in the context diagram. In UML, one can apply Use Case diagrams for this purpose, with them one can model the services that the system provides and that are visible from outside, with a "black-box view," independent of how it is done, it is to say, model the requirements of the system. Also, it is possible to define the limits of the system by drawing a line showing which actors interact with the system from outside (Allen and Frost, 1998).

There is no need to define every service in the beginning, incremental developing allows designers to choose a use case, design an initial system that provides that functionality, then choose another use case, enlarge the system and so on, until all the requirements are covered.

Use case diagrams, due to their simplicity, are very useful for eliciting requirements and verifying them with the user, but in this case, as the system is already designed, they will not be used for this purpose.

Different granularity can be applied when defining a use case, large granularity can be useful when eliciting requirements, but for design, it could turn into complicated diagrams, difficult to understand, that might lead to less reusable components that the ones coming from smaller granularity. From the SELECT perspective, a use-case is "a self-contained unit of interaction with no intervening time delays. It must be performed by a single actor, in a single place, although it might result in output flows that are sent to other passive actors. The use case must also leave the system in a stable state, this may result in some measurable value to the user" (Allen and Frost, 1998).

Following this guideline, it was decided not to use a bigger use case like "Process book order" and start with a smaller one: "Record book order" (Figure A3(a)). It was not difficult to identify the actors and the external event that triggers the use case. The event is number 1 from the event list: Customer orders book. The external actors

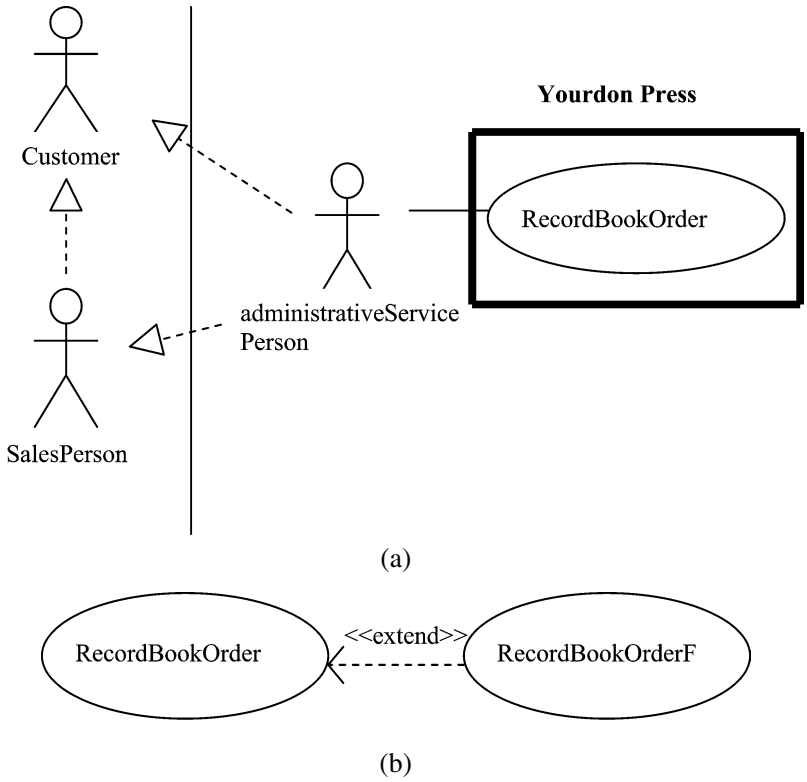


Figure A3. (a) Use case diagram: RecordBookOrder. (b) Extending use cases.

involved are customer or sales person on behalf of him, and the internal actor, the one that interacts with the system, is the administrative service person, as can be seen in the introduction of the case study.

A.3. Extends and uses "A use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor" (UML users guide). There is a variant for this use case: the order is placed by a new customer (Figure A3(b)). The variants of use cases that extend the main flow are other use cases, and are related with the extended by an "extends" relationship.

Note. In the context diagram of Yourdon Press, there is a data flow for rush orders, but it does not appear in the lower levels of Yourdon's DFD's. It could be another extend use case: RecordRushBookOrder, but the authors decided not to model it.

Use cases can be better specified using text to describe the flow of events. It is possible to know how a use case is worked out following the thread of the data flow diagrams. The starting point is the event triggering the use case. Tables 1 and 2 are the textual description for the use cases.

Table 1. Record Book Order use case.

Name	Record Book Order
Intent	To allow an administrative service person to create book orders for customer, specified quantities of different books
Description (main flow)	1. Edit order detail 1.1 Verify customer ID (book order from new customer) 1.2 For each order item Verify book code 1.3 Verify sales-tax-rate 1.4 Verify shipping charges 2. Check book in stock 3. Check payment 4. Enter order
Exceptions	1. Invalid order details Display message Exit *Order not processed* 2. Not enough books in stock Display message Exit *Order not processed* 3. Not payment OK Display message Exit *Order not processed* 1,2,3. Order cancelled Exit *Order not processed*

Table 2. Record book order from new customer use case.

Name	Record book order from new customer Extends record book order
Intent	To allow an administrative service person to place a book order for a new customer and enrol him in the agency plan
Description (main flow)	1. Enter new customer details 2. Enrol customer in the agency plan
Exceptions	1. Invalid details Display message Exit *Order not processed* 2. Invalid choice for the agency plan Display message Continue *Order not processed*

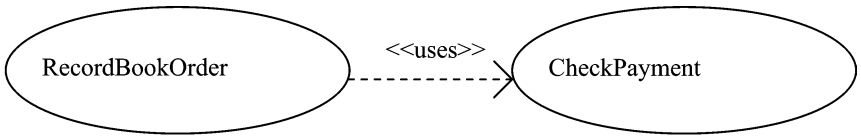


Figure A4. Extracting common behaviour.

"Check payment" in the use case "RecordBookOrder" can be a single use case. The relation between them will be «uses» relationship (Figure A4).

This refinement will allow us to reuse "Check Payment" in another use case such us "Manage Payments."

One has to be careful because each process does not correspond to a use case. The process "1. Process Orders" is not only related with these three use cases, there is also in its expansion the process "1.3 Send Invoices" that in the model would go with other use case.

A.4. Creating the class diagram

Once we have the use cases, the next step is creating the *class diagram*. This is the heart of the object-oriented model. The rest of the diagrams are based on it. It is very important that it is correctly done. UML does not have a technique for identifying classes and their relationships. We used the guidelines from SELECT perspective in "Component Based Development for Enterprise Systems" (Allen and Frost, 1998). Hints and tips from UML Users Guide (Booch et al., 1999) and the strategies and patterns given by Coad et al. (1995).

A.4.1. Partitioning the system Before identifying the classes, the system was partitioned into four packages (Figure A5) (Coad et al., 1995).

The packages are:

- Problem Domain (PD) (classes related to the business at a hand).

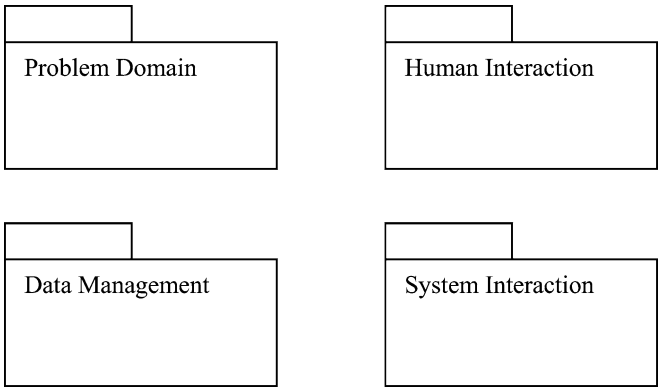


Figure A5. Partition of the system.

- Human Interaction (HI) (windows and reports).
- Data Management (DM) (database).
- System Interaction (SI) (other systems).

The purpose of these packages is to organise the model. The business classes become independent from the rest, usually less stable and more dependent with the environment. It is important to distinguish between this type of package and those in component-based development, according to Paul Allen: reusable packages holding a software component that can be used like LEGO pieces (Allen and Frost, 1998).

A.4.2. Re-designing the system without losing data or functionality Re-designing a system could cause the loss of data or functionality. One must not forget that changing the design must not change the system. One must not lose information and the new system has to offer at least the same services.

A.4.2.1. Selecting the related DFDs. The first step is to select the DFDs related with the use case "RecordBookOrder." It is done searching for the flow that activates the DFD "1. Process Orders." It is "Order." Follow the data flow in the child. This is "Order details" and goes to "1.1 Edit order detail."

A.4.2.2. Using language analysis to identify the classes. In structured modeling, the data and the operations are separate. The operations are located in the DFDs and the data in the Entity-Relationship diagram. Identify the Problem Domain classes using language analysis inside the process. Coad et al. (1995) recommend identifying this type of classes first. Human Interaction, System Interaction and Data Management classes are easily figured out afterwards. Consider nouns as potential classes, especially if they appear in the entity-relationship diagram. Join the data and the operations. Thus one has to swap from one diagram to the other.

A.4.2.3. Applying patterns. The classes are also selected if they conform to patterns. A pattern is a way of reusing good modeling ideas that apply to different types of systems and prevent one from "re-inventing the wheel." The authors used the patterns

proposed by Allen and Frost (1998). As UML is not chained to a specific methodology, one has the advantage of using the best features from its constituent methodologies.

A.5. Potential and accepted models

A.5.1. Creating the first class Two models are used: the potential model and the accepted model. In the potential model, all the features that could be in the system are added to. They are transferred to the accepted model as one goes through the design if they are related with the actual use case. The reason for this is not only to do some work in advance, but also to avoid forgetting anything along the way.

The process begins checking if a CUSTOMER exists, customer is also a data store in the original design. Customer is the participant in the transaction order. This is the first class: "Customer." It is added to both of the models, the potential and the accepted. The first attributes come from the data dictionary; add them to the potential model.

The data dictionary entries are:

CUSTOMER = {Customer}

Customer = *a Yourdon Press customer*
@customer-ID + (company-name) + customer-name
+ customer-address + current-balance + credit-limit
+ agency-plan-level

Customer-address = *"bill-to" address: where we send the customer invoices*
Street-address + city + state + postal-code + (country)

The system checks if a customer exists or not by using its ID. The authors added customer ID to the accepted model and the operation "findforID(customerID)" to both of them (Figure A6).

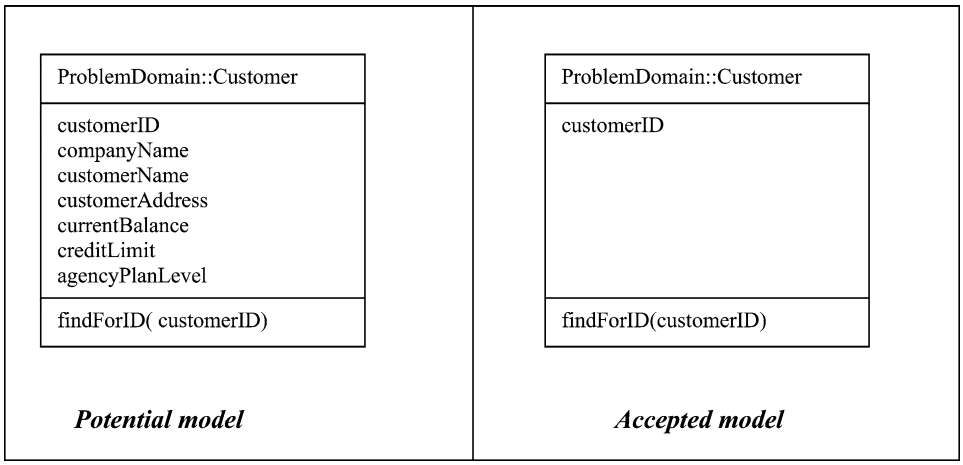


Figure A6. Class Customer.

A.5.2. Selecting more classes Order is the transaction a customer makes. This is the second class (Figure A7). The definitions of the attributes come from the data dictionary:

ORDERS = {order}
order = * and order from a Yourdon Press book*
@invoice-number + customer-ID + order-date + {order-item}
+ shipping-charges + sales-tax + shipment-date + (salesperson-ID)
+ order-total + warehouse-ID

As in customer @ points to the unique attribute used to identify the data, so the operation "findForInvoiceNumber" is added to the class (Figure A8).

A.5.3. Converting an attribute to a new class The attribute {order-item} requires a different treatment. First of all, it has its own attributes, that make it candidate for being another class, and second, an order is compound of order items. Having a look at the pattern "transaction–transaction-line-item" (Figure A9), there is no doubt that we have another class (Figure A10) and a composite aggregation relationship (Figure A11).

The data dictionary entries are:

ORDER-ITEMS = {orderitem}
order-item = @invoice-number + @book-code + quantity-ordered
+ unit-price + discount

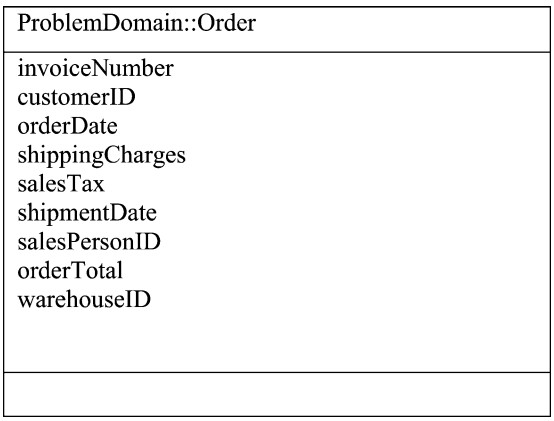


Figure A7. Class Order.

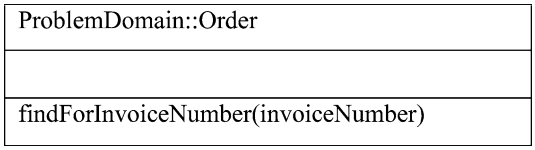


Figure A8. Class Order.

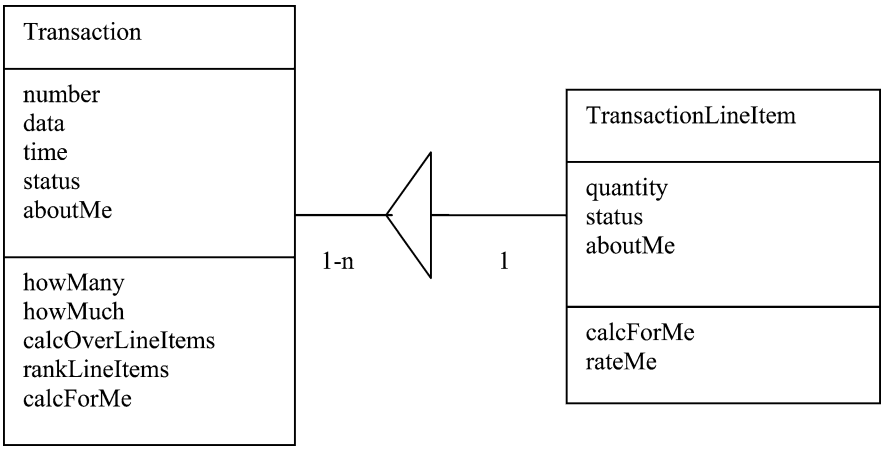


Figure A9. Pattern Transaction-TransactionLineItem.

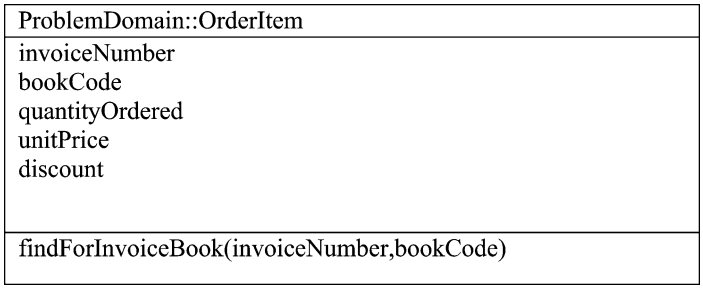


Figure A10. Class OrderItem.

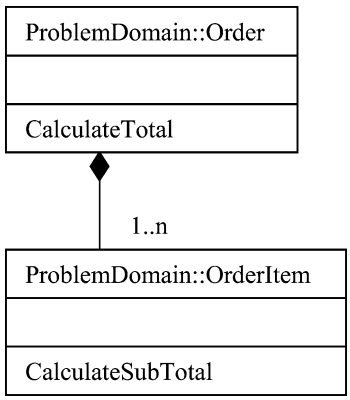


Figure A11. Composite association.

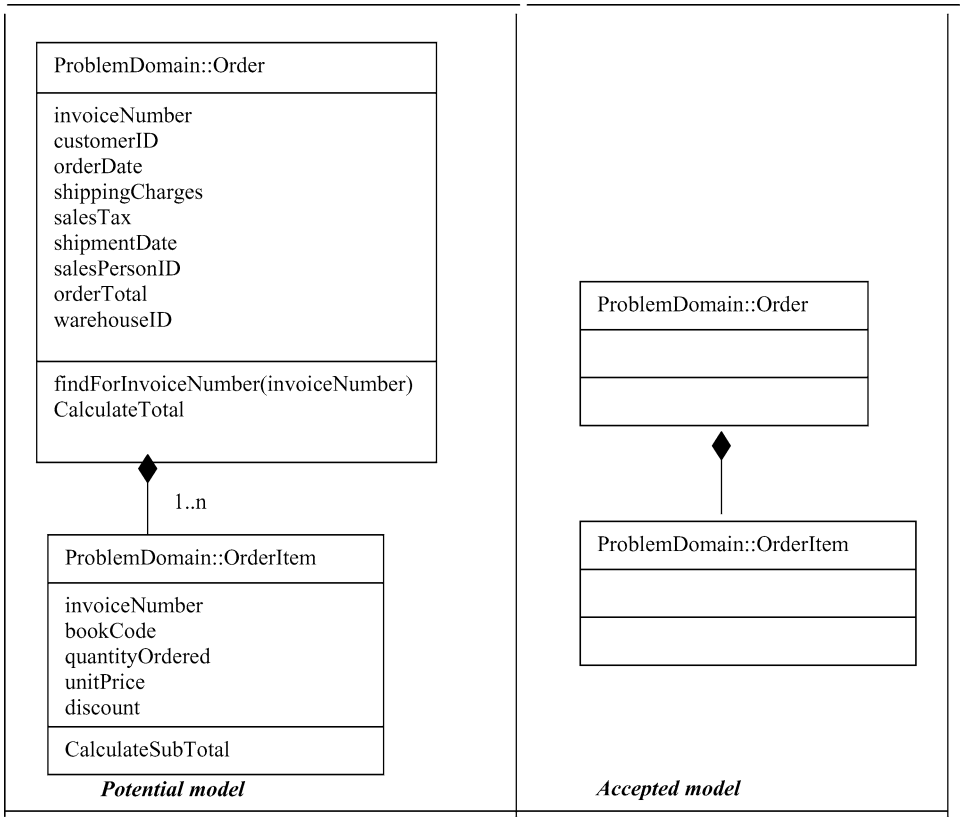


Figure A12. Classes Order-OrderItem.

By using the pattern one can do some work in advance. In order to calculate the total of an order one must calculate the subtotals for each order item. This is a service that "OrderItem" will provide. This is called "delegation": Order delegates the task of calculation to its items.

Two classes were added to both models. The attributes and the operations were only added to the potential model (Figure A12).

A customer places an order for books. BOOK is another class. The data dictionary entries are:

BOOKS = {book}

Book = *information maintained about a Yourdon Press book*
@book-code + book-title + author-ID + total-in-stock
+ or-order-quantity + on-hand date + royalty-rate
+ out-of-print-indicator + reorder-threshold

When a customer places an order, an order item does not refer to a generic book. It refers to a specific book. The customer orders an "Inventory Item."

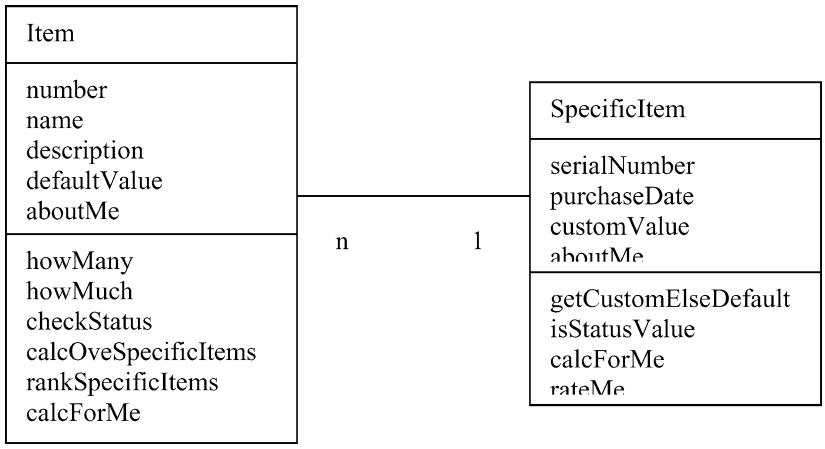


Figure A13. Pattern Item-SpecificItem.

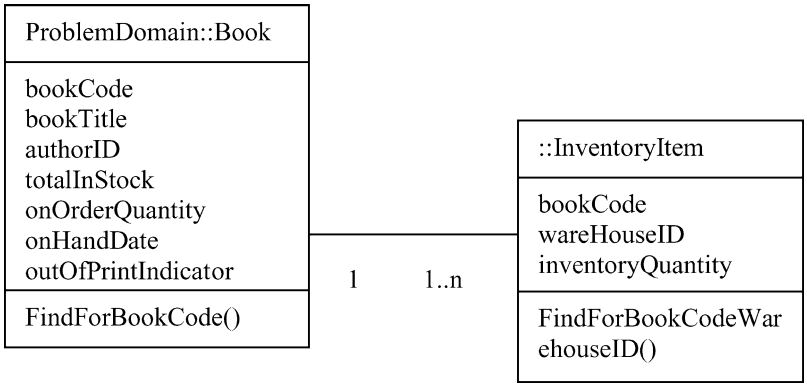


Figure A14. Classes Book-InventoryItem.

INVENTORY-ITEMS = {inventory-item}
Inventory-item = *a group of books with the same title, located in a single warehouse*
@book-code + @warehouse-ID + inventory-quantity

Apply the pattern "Item-SpecificItem" (Figures A13 and A14).

The pattern is applied for "Book-InventoryItem" and the classes are enclosed in both of the models (Figure A14).

The relationship between customer and order conforms to the pattern "Participant-Transaction" (Figure A15).

The pattern is applied and keep on tracing the DFD "Edit Order Details." The first "IF" inside the process is related to the extending use case. Do not take it into consideration now.

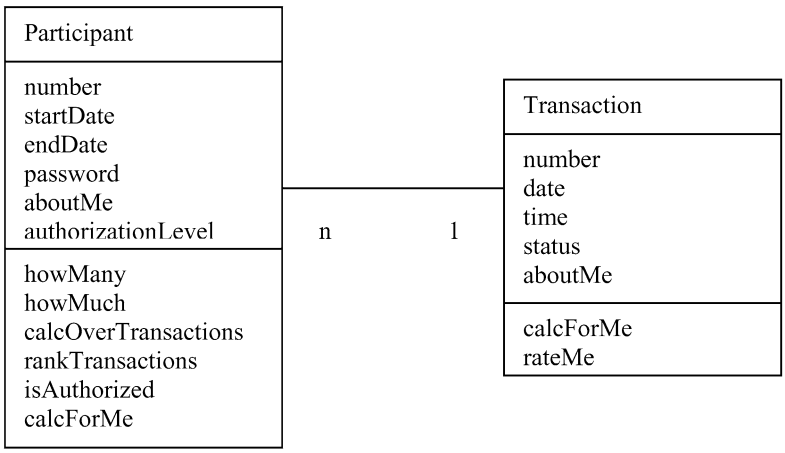


Figure A15. Pattern Participant-Transaction.

A.5.4. Identifying attributes and operations Books are found by the attribute "bookCode." The operation is transferred to the accepted model, together with the attributes "shippingCharges" and "salesTaxRate."

Checking the existence of customer and books is an operation performed by the class "Order." The class delegated the operation of checking each book to its order items. Thus the operations "checkBooks()" and "checkCustomer()" are added to both models. Also check "shippingCharges" and "salesTaxRate;" the operations "checkShippingCharges" and "checkSalesTaxRate" are needed in the models.

Another two attributes are required:

- "orderStatus": An order can be cancelled if "order-details" are not correct.

  There are two possible statuses for an order, "edited" and "cancelled."

- "orderResponse": If the order is cancelled, one needs to know the reason why:

  "No such customer;"
  "No such book;"
  "Illegal sales tax rate;"
  "Illegal shipping charges."

As an order delegates the tax of checking if a book exists to order item, The attribute "OrderItemStatus" and "OrderItemResponse" is added to the class "OrderItem."

- "orderItemStatus": "edited," "cancelled."
- "orderItemResponse": "No such book."

Tracing the DFD "Edit Order Details" for this use case is now complete and the progress at this point can be seen in Figure A16.

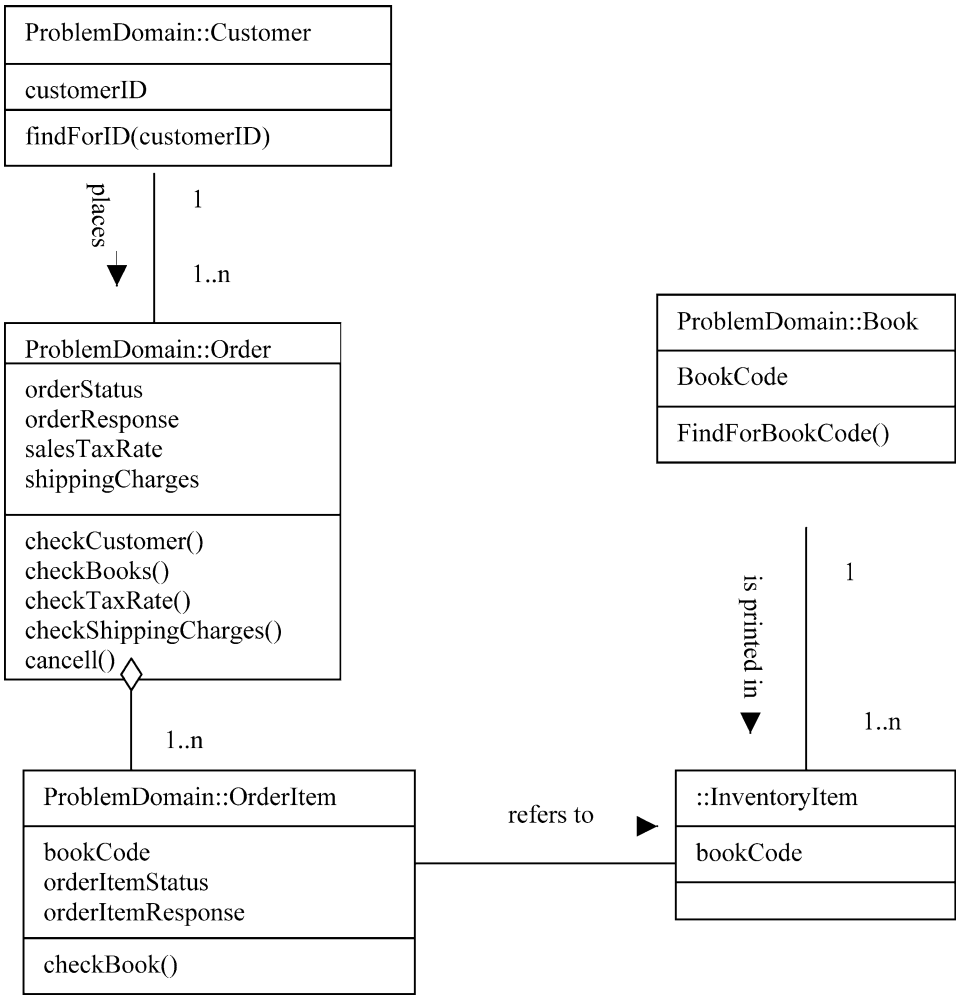


Figure A16. Progress so far: accepted model.

A.5.5. Rejecting a diagram Figure A16 shows the First-Cut class diagram. It will not be the definite diagram. It is very difficult to draw the correct diagram the first time. It requires experience and knowledge of the system. It is good practice to include these diagrams in the model and the reason why they were rejected. It will prevent us from coming back to them. In large projects many people are involved. Another person might change the diagram, without concern about the reasons why it was rejected.

In this diagram, the class "Order" has the responsibility of checking "Order-details" and canceling the transaction if anything is wrong. The business class becomes a controller. It increases coupling with the user interface. It is possible to delegate this responsibility to the class, but if there are several interactions it is better to introduce another class. "OrderControl" will now carry out this task.

The operations that it will perform will be:

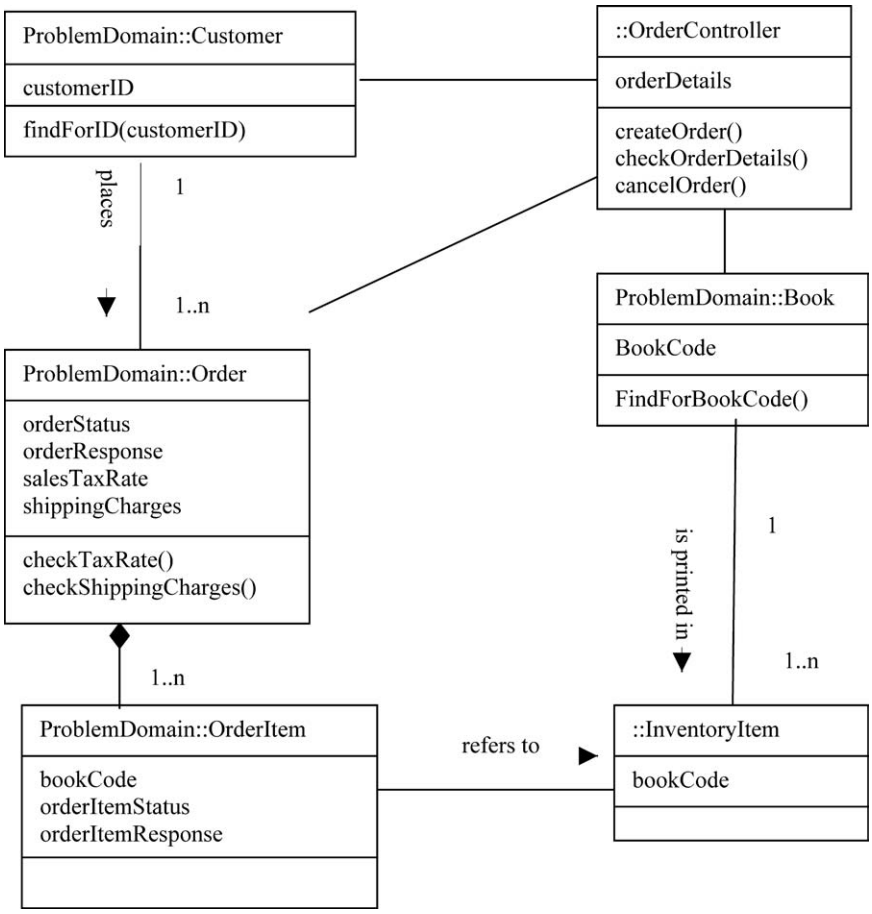


Figure A17. Accepted model: new class diagram.

- createOrder().
- checkOrderDetails().
- cancelOrder().

The new class diagram (Figure A17) is similar to the former with the exception that there is a new class. This diagram shows the static behaviour. The interaction diagram will show who carries out these operations.

A.5.6. Tracing another DFD When the DFD "Edit Order Details" finishes it displays its results to the DFD "1.1.2 Check Book in Stock." New classes can be found within the domain vocabulary. There is the class "Warehouse." The data dictionary entry is missing, but there is a data store in the Entity-Relationship diagram.

There is an entry related with it:

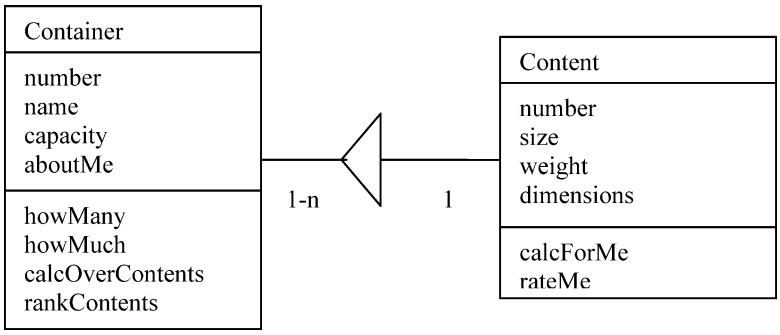


Figure A18. Pattern Container-Content.

warehouse-ID = *identification of the various warehouses where
Yourdon Press books are stored*
Apply the pattern "Container-Content" (Coad et al., 1995, p. 435)
to the classes "Warehause" and "InventoryItem" (Figure A18).

The class "Warehouse" with the attribute "warehouse-ID" was added to both models, and an association relationship between "Warehouse" and "InventoryItem" was created. The reason for creating a class with only one attribute is that one can get some services from it. It will also make the system easy to enlarge or modify.

A.5.7. Converting branches into specialised classes Now trace the DFD. This process behaves in three different ways depending on the type of order:

- "Back order" is an order that will be processed even if there is not enough stock to satisfy the order.
- "Walk-in order" is an order placed by a customer in a specific warehouse and can only be satisfied with stock inventory items from that warehouse.
- "Mail order" or "Phone order" can be filled by any of the warehouses with enough stock.

If there is not enough inventory items in any of them it will not be processed. This means that the general class order has three children.

Specialisation/generalisation relationship must be used carefully. It is a powerful mechanism but we can mistake it for "Implementation inheritance." One cannot use it only to provide a common interface. It has to reflect the business structure and have a semantic meaning. This case is suitable for using inheritance. The different orders have not only different behaviour, but also different attributes (Figure A19).

In the potential model we have the attribute "warehouseID." The class "Order" is in charge of setting it. The way it is achieved depends on the type of order. If an order could be from different types at the same time, this task would have to be delegated to order item. Here an order is only of one kind. "Phone order" and "Mail order" are grouped into "PhoneMail order" because they behave in the same way and have the same attributes. The attribute "type" will be used to differentiate them. An "OrderItem" is responsible of checking if there is enough stock of an "InventoryItem"

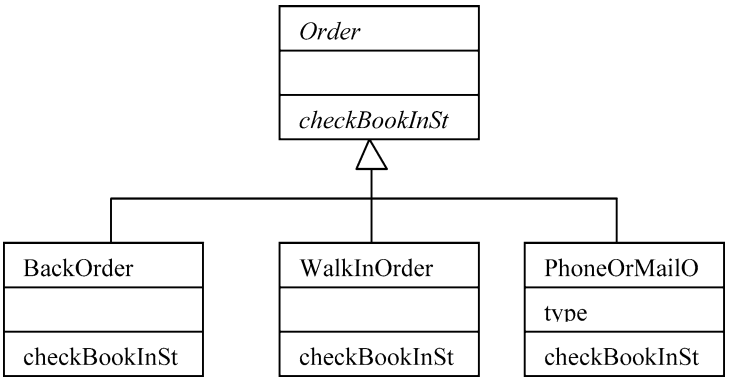


Figure A19. Generalisation-specialisation for the class "Order".

in a warehouse. The warehouse is set by "Order." "OrderItem" does not need to be specialised.

"Order" class becomes abstract, it can not be instantiated. The services provided by the class will have a common interface but they will be carried out in a different way depending on the type of order. The operation "checkBookInStock()" has the same signature in the father than in the child. The operation in the child overrides the operation in the father, due to Polymorphism.

There is another possible value for the attributes "orderResponse" and "orderItem-Response." It is "Not enough books in stock to fill the order." An "OrderItem" will ask an "InventoryItem" if there are enough books. The operation "checkBookInStock()" is added to the class "InventoryItem."

These classes are added to both of the models and the following attributes are transferred from the potential model to the accepted model:

- "warehouseID" to the classes "Order" and "InventoryItem."
- "bookCode" to the classes "OrderItem" and "InventoryItem."
- "quantityOrdered" to "OrderItem."
- "inventoryQuantity" to "InventoryItem."

Tracing the process "1.12 Check Book in Stock" is now completed.
The progress at this point is shown in Figures A20 and A21.

A.5.8. The use case "Check Payment" The next DFD is "1.1.3 Check Credit Authorisation."

This process belongs to the use case "Check Payment." Even though the actual use case is "Record Book Order," the modeling of the use case "Check Payment" is not delayed because the actual use case is related to it by means of a «uses» relationship. It means that "CheckPayment" will be included in "RecordBookOrder."

The new class we find is "Payment," the data dictionary entries are:

PAYMENT = {payment}

payment = *payment made for a book order or to pay an invoice*
(customerID) + payment-date + payment-details

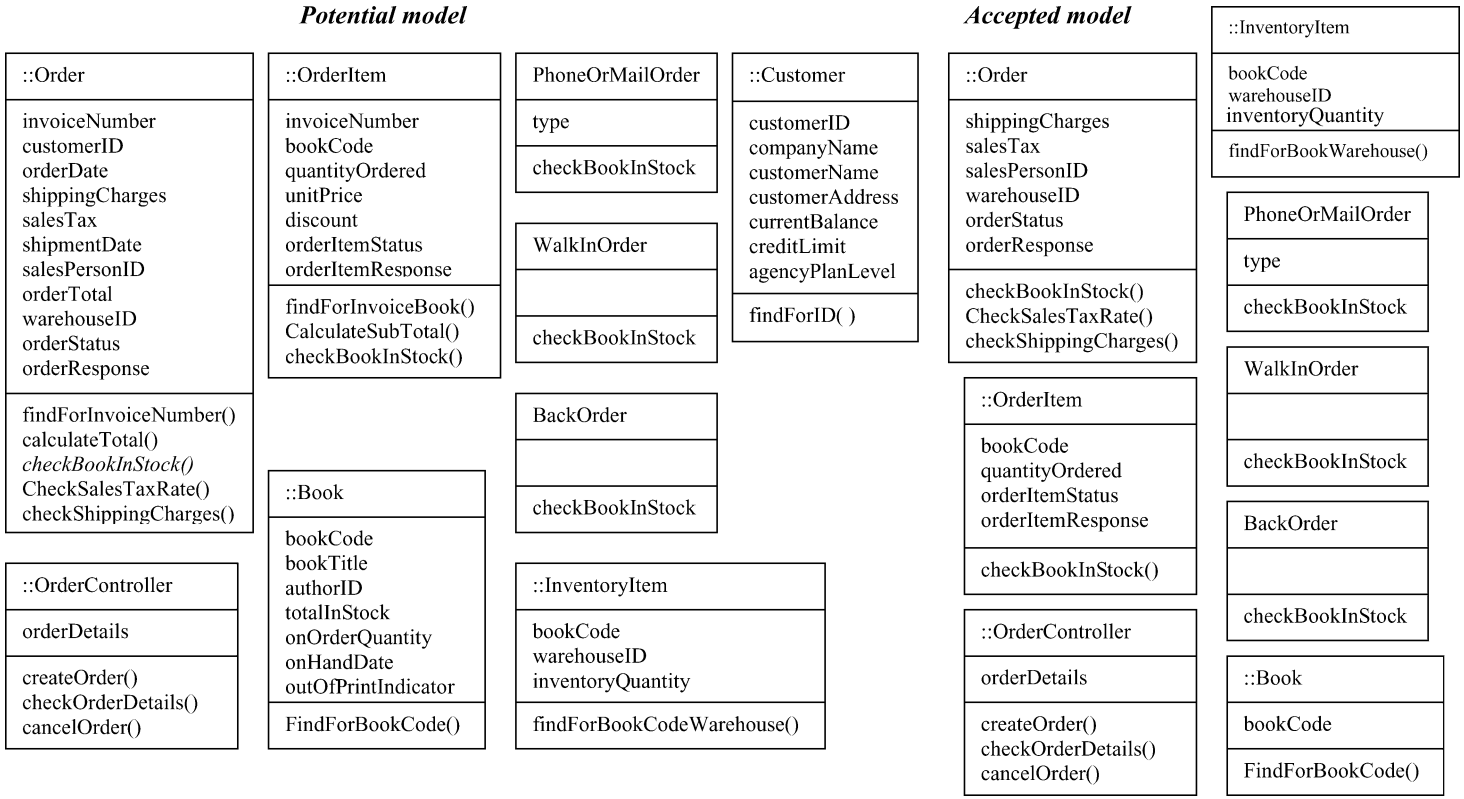


Figure A20. The classes in the problem domain.

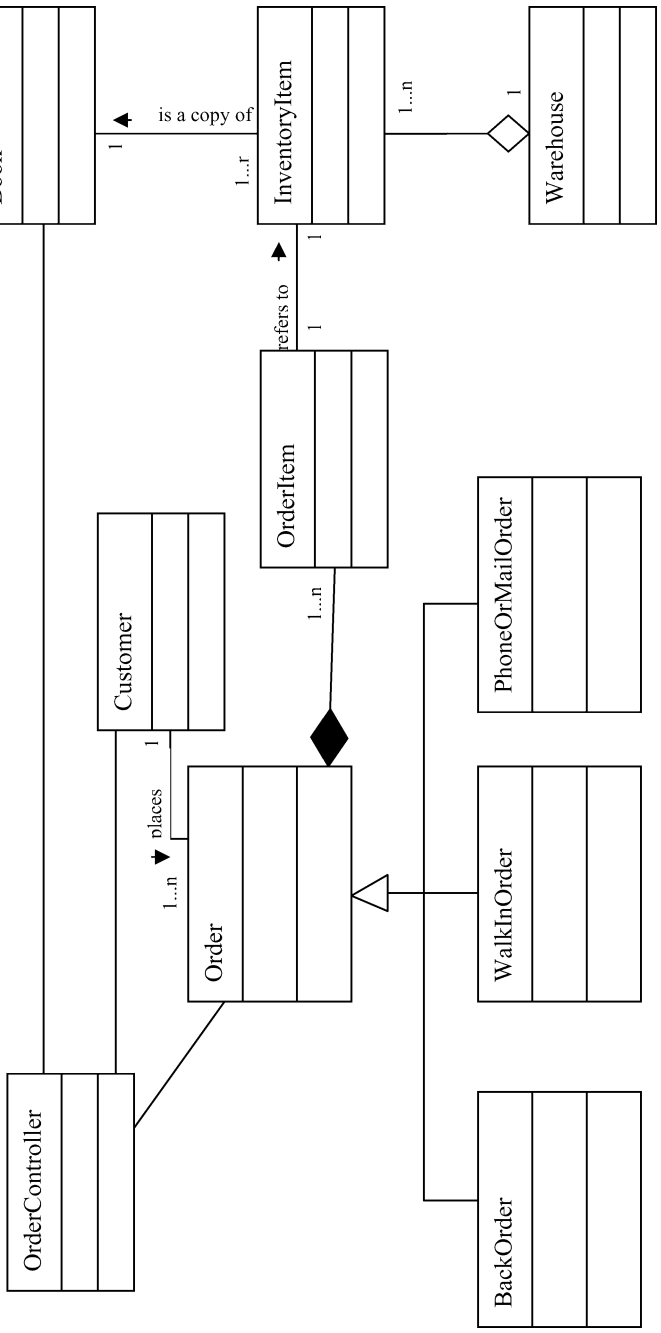


Figure A21. The class diagram, with the attributes and the operations elided is the same in both of the models.

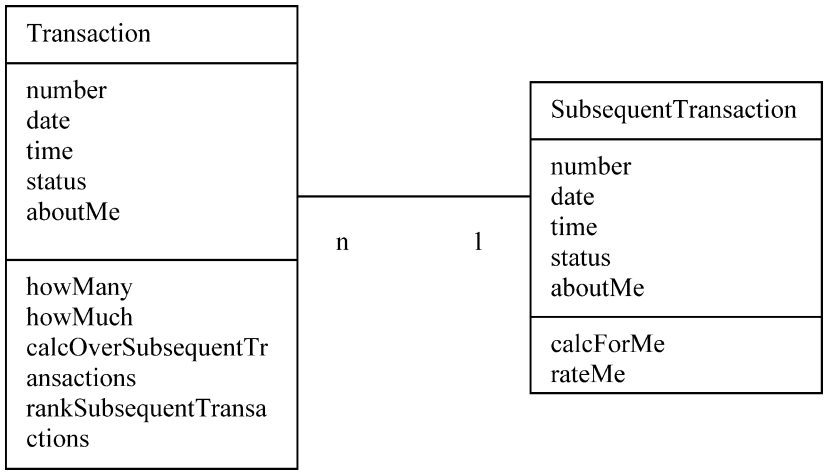


Figure A22. Pattern Transaction-Subsequent transaction.

payment-details = *detailed information about an item or invoice being paid for*
{invoice-number} + total-amount

The class is added to the models, the attributes to the potential model.

In Yourdon Press, a payment can refer to several invoices. It was decided to add the operation "findForInvoiceNumber()" to the potential model. It is its most distinctive attribute and it could be useful, in case we wanted some information about an invoice.

Apply the pattern "Transaction-Subsequent Transaction" (Figure A22).

The DFD begins by calculating the total price of the order. The attributes "orderTotal," "unitPrice," and "discount" are transferred. It calculates the price of the order. In our model each orderItem will calculate its subtotal and order will use it to calculate the total-price. These operations are already in the potential model, due to applying the pattern. They are transferred to the accepted model.

A.5.9. Generalisation/specialisation Once the price is calculated, the payment is processed. There is a "CASE" structure. As in the previous DFD, branches might be signs of a generalisation/specialisation relationship. If the thread to follow is ruled by the different values of an attribute from a class, the probability of sub-typing is high.

As with the class "Order," one has to be sure of not using this relationship to simplify implementation. In this case, specialising the class "Payment" reflects the business structure. The children have different behaviour and attributes. Thus it is possible to apply it (Figure A23). "Payment" is an abstract class. The operation "makePayment()" is over-ridden in the children.

Before making the payment it is necessary to check if it can be done. "checkPayment()" is another deferred function in the abstract class. It is placed in the parent and in the children. One needs to know if the operations were successful. Two attributes are used for this purpose:

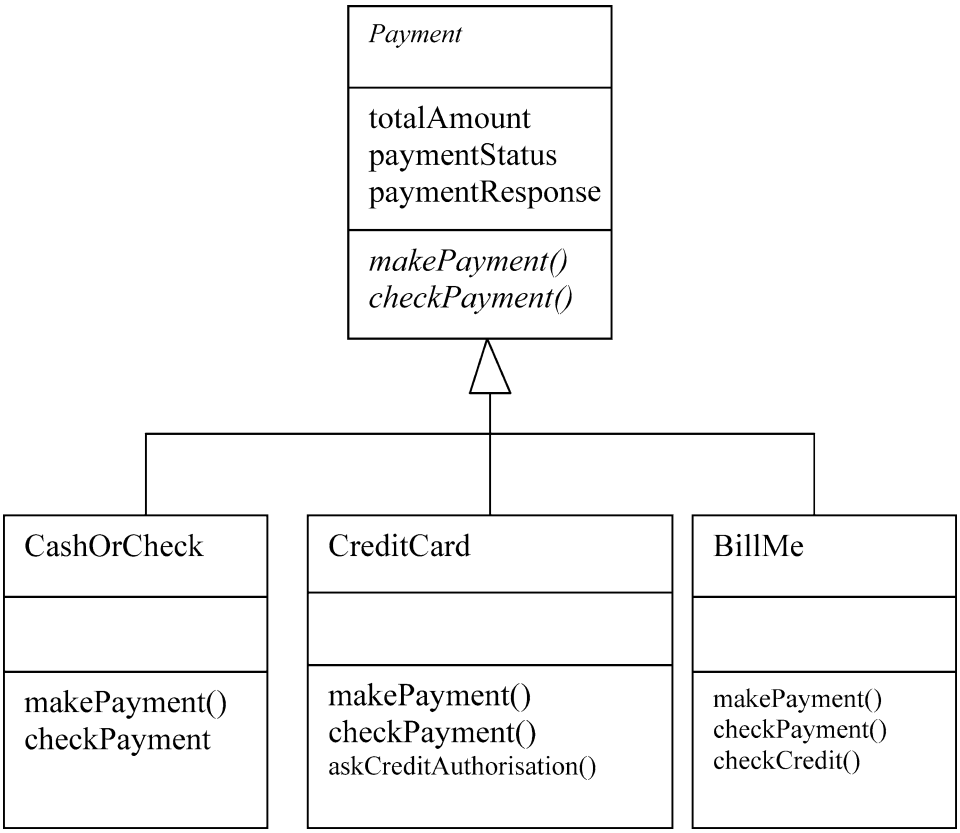


Figure A23. Generalisation-specialisation for the class "Payment".

"paymentStatus" = "OK"
"cancelled"
"paymentResponse" = "Purchase price exceeds amount paid"
"Credit request denied"
"Order exceeds your credit limit"

Tracing the DFD, the following attributes are transferred from the potential to the accepted model:

- "totalPayment" to the class "Payment."
- "currentBalance" and "creditLimit" to the class "Customer."

Class "BillMe" has to check the credit of the customer, it delegates this operation to the class "Customer." "checkCredit" is added to the classes "Customer" and "BillMe."

Class "CreditCard" has to ask for credit authorisation. The operation "askCreditAuthorisation" is enclosed.

Class "Payment" in the accepted model can be seen in Figure A23.

When the DFD finishes, the class "Order" could be "Billed," "Paid" or "Cancelled." There are two more possible values for the attribute "orderStatus."

A.6. Tracing the last DFD's

If the order is not cancelled, two processes will be running concurrently: "1.1.4 Enter Order" and "1.1.5 Check Inventory for Reprinting". Concurrency can be shown in the activity diagram.

"1.1.4 Enter Order" is traced in first place. There are two new classes, "Invoice" and "Money." They are related to the use case "CheckPayment." The data dictionary entries for them are:

INVOICES = {invoice}

invoice = *information contained in a Yourdon Press Invoice*
@invoice-number + customer-name + customer-address + order

MONEY = {money}

money = *information about checks, cash or other moneys*
@money-date + @customer-ID + {invoice-number} + money-amount

These classes are added to both models. The attribute order in the class "Invoice" is in fact a composite aggregation between the classes "Invoice" and "Order." The pattern "transaction-subsequent transaction" can be applied between

- "CashOrCheck," "CreditCard" and "Money."
- "Order" and "Invoice."

A.6.1. Selecting data management classes In this process the data are stored, in our model this is a task for the data management classes. The classes "Order," "OrderItem," "Invoice," "Payment" and "Money" are added to the package "Data Management." The operations they will perform will be "store" and "search." These classes make the Problem Domain classes independent from the data base details, making them more reliant.

The following attributes are transferred from the potential model to the accepted model: "customerID," "orderDate" to the class "Order":

- "invoiceNumber," "customerName" and "customerAddress" to the class "Invoice."
- "moneyDate," "invoiceNumbers," "customerID," "moneyAmount" to the class "Money."
- "invoiceNumbers," "customerID," "paymentDate," "paymentDetails" to the class "Payment."

Once the data are stored, the status of order changes to "accepted." It is another possible value of the attribute "orderStatus."

This completes the use case "Check Payment." Return to "RecordBookOrder."

The DFD "1.1.5 Check Inventory for Reprinting" is traced. There are no new classes. The following attributes are transferred:

- "totalInStock" and "reorderThreshold" to the class "Book."

The class "OrderItem" will ask the class "InventoryItem" to subtract the quantity ordered from the inventory quantity, the class "InventoryItem" will then ask the class book to subtract the quantity from the total in stock. The class "Book" will check the reorder threshold and send a message of low inventory if needed. Again there is delegation of the operations between the classes. This reduces the coupling, the class that delegates the service does not have to know how it is achieved.

The following operations are enclosed:

- "orderBooks" to the class "Order."
- "orderBook" to the class "OrderItem."
- "subtractQuantity" to the class "InventoryItem."
- "subtractQuantity," "checkThreshold" and "sendLowInventoryMess" to the class "Book."

A.6.2. Identifying common behaviour The DFD "1.2 Process salesperson Order" is exactly the same as "1.1 Process Order," except for that the order is placed by a salesperson in behalf of the customer.

The class "Salesperson" is added and the pattern "Participant-Transaction" is applied between it and "Order."

The data dictionary entries are:

$$\text{SALESPERSON} = \{\text{salesperson}\}$$
$$\text{salesperson} = @\text{salespersonID} + \text{salespersonName}.$$

An order can be placed by a customer or by a salesperson but not by both at the same time. This can be indicated with a constraint {XOR}.

A.6.3. Doing work in advance The DFD "1.4 Send Invoices" is in charge of sending the invoices. This DFD is not related to the actual use case but is the only remaining inside the DFD "Process Orders" and it could be forgotten. Adding the operation "send()" to the class "Invoice" only in the potential model will be helpful when modeling the use case "Manage Payments." Some work is done in advance and the possibility of losing functionality is reduced.

A.6.4. Modeling the extending use case "Record Book Order from new customer" To finish with the use case "Record Book Order" the extending use case "Record Book Order from New Customer" is modelled. It forces one to trace the DFD's once again, but most of them are covered.

There is only one "IF" left in the DFD "1.1.1 Check Order Details" and the DFD "1.3 Enrol Customer in Agency Plan."

In the DFD "1.1," if the customer is new, it is added to the system. The operation "createCustomer()" is added to the classes "OrderController," and "Customer."

The 4 following attributes are transferred from the potential to the accepted model: "customerName," "customerAddress," "companyName" to the class "Customer."

The class "Customer" in the DataManagement package will store the new customer.

The attribute "agencyPlanLevel" is a composite aggregation. The class "AgencyPlan" is created together with the relationship between it and "Customer." With this class one can finish tracing this DFD and "1.3 Enrol customer in agency plan."

The following attributes and operations are defined in the class "AgencyPlan":

- "planLevel."
- "planChoice."
- "changePlanLevel()."
- "getplanChoice()."
- "enrolCustomer()."

A.6.5. Selecting human interaction and system interaction classes The use cases are completed by adding the human and system interaction classes.

The human interaction classes are windows and reports: one needs "RecordBook OrderWindow," "NewCustomerWindow," and "PaymentWindow."

They will perform operations like "display()" or "print()."

The system interacts with the credit agencies, and with management, the classes one needs are: "CreditAgenciesSI" and "ManagementSI," the operations will be "connect()," "sendMessage()," "recieveMessage()."

At this point, the system is able to provide the service "Record Book Order." One can show how the system works with the interaction diagrams.

The class diagram for the accepted model can be seen in Figure A24.

A.7. Showing the dynamics

A.7.1. From use-case to interaction diagrams Until now one has worked with classes and their relationships, from this point one will have objects, links and messages.

Sequence diagrams are used to show the main flows of the use case and collaboration diagrams to show the exceptions. The notation is kept as simple as possible. It is better not to nest optional flows in the same diagram because it makes them difficult to read. It is possible to have a set of diagrams for a use case and to keep everything in order we can join them in a package. It is easier to follow a thread with a sequence diagram than with a collaboration diagram, that is why it was decided to use them to model main flows.

Beside the interaction diagram one puts the DFD's related with it and highlights in red the parts that can be traced. Once the whole system has been remodelled, every line should be highlighted. This will show that functionality has not been lost and that the system achieves the services following the same business steps.

A.7.2. Diagrams for the use case "RecordBookOrder" The first two sequence diagrams (Figures A25 and A26) show the main flow of the use case. Polymorphic behaviour has been elided to simplify. It will be shown via other sequence diagrams.

Exceptions in the main flow can be modelled applying collaboration diagrams, Figure A27 is the collaboration diagram that shows what happens when the customer's

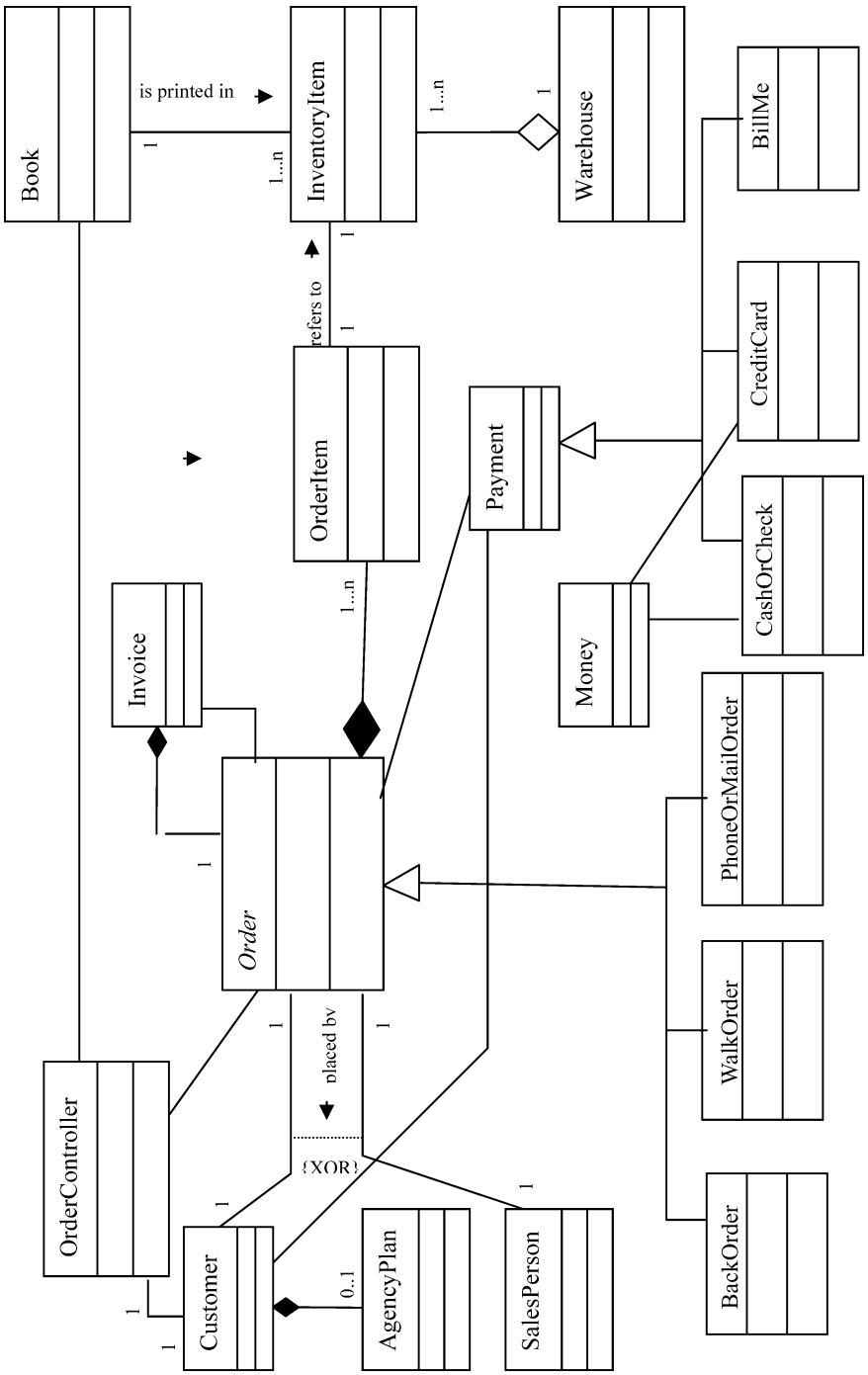


Figure A24. The class diagram in the potential and the accepted model.

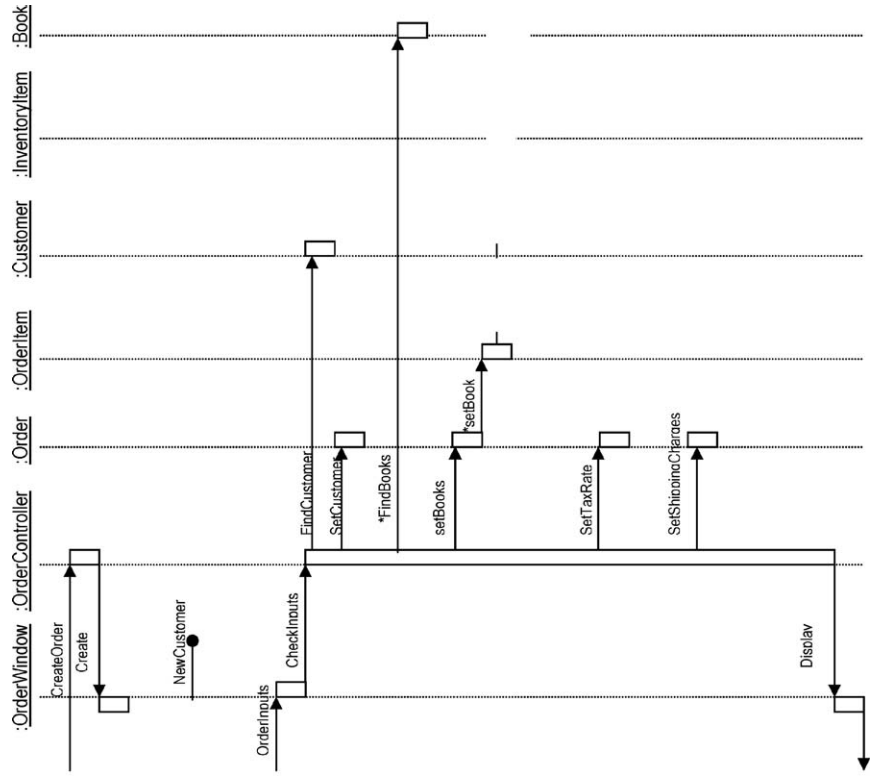

:OrderWindow
:OrderController
:Order
:OrderItem
:Customer
:InventoryItem
:Book
CreateOrder
Create
NewCustomer
OrderInputs
CheckInputs
FindCustomer
SetCustomer
*FindBooks
setBooks
setBook
SetTaxRate
SetShippingCharges
Display
PROCESS 1.1 EDIT ORDER DETAILS
BEGIN
IF Tentative-customer-ID="new"
customer-ID=next available customer-ID
credit-limit=standard credit limit
current-balance=0
agency-plan-level=0
customer=customer-ID+customer-name+(company-name)+
customer-address+current-balance+credit-limit+
agency-plan-level
APPEND new customer record to CUSTOMERS
ELSE
FIND customer in CUSTOMERS with customer-ID =customer-ID
in order-details
IF record cannot be found
order-response="no such customer"
DISPLAY order-response
EXIT
ENDIF
DO WHILE there are more order-items in order-details
FIND book in BOOKS with book-code = book-code in order-item
IF record cannot be found
order-response="No such book"
DISPLAY order-response
EXIT
ENDIF
ENDDO
IF sales-tax-rate is out of range
order-response = "illegal shipping charges"
DISPLAY order-response
EXIT
ENDIF
IF shipping-charges is out of range
order-response = "illegal shipping charges"
DISPLAY order-response
EXIT
ENDIF
valid-order-details=customer-ID+{order-item}+sales-tax-rate+
shipping-charges+payment-type+{order-payment}+
(credit-card-number)+back-order-OK+order-type
DISPLAY (to process 1.1.2) valid-order-details

Figure A25.

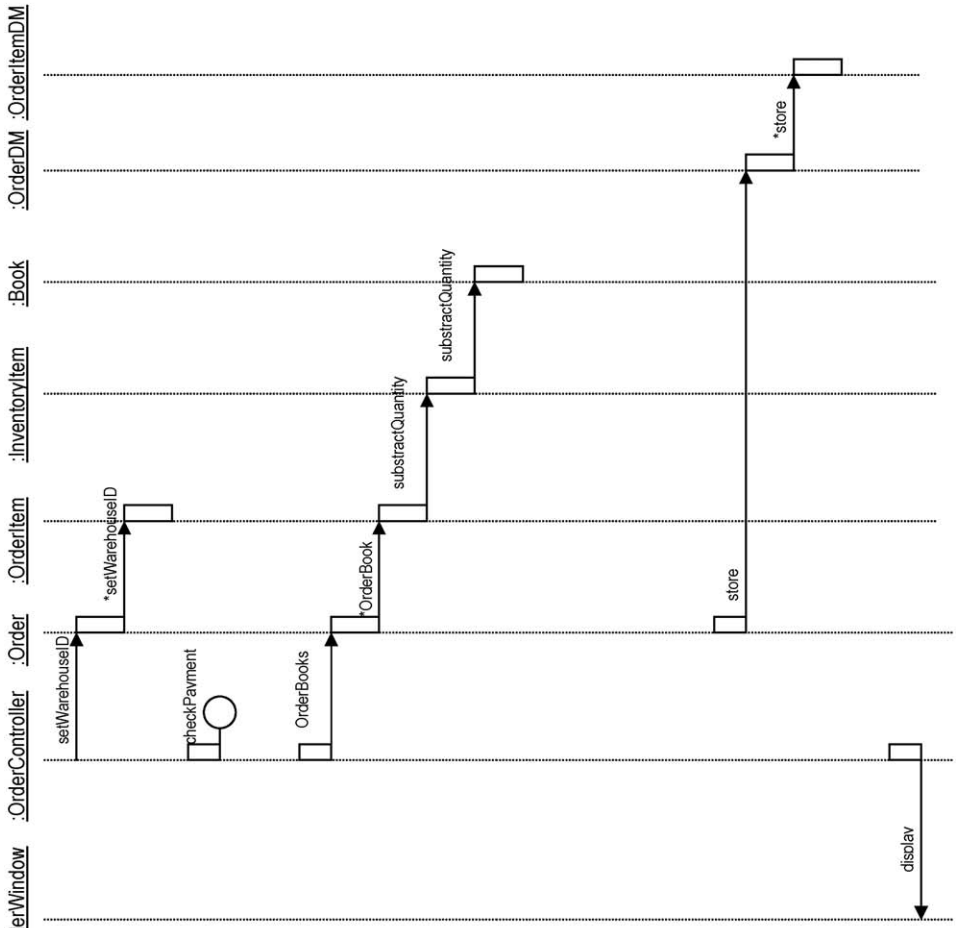

:OrderWindow
:OrderController
:Order
:OrderItem
:InventoryItem
:Book
:OrderDM
:OrderItemDM
setWarehouseID
*setWarehouseID
checkPayment
OrderBooks
*OrderBook
substractQuantity
substractQuantity
store
*store
display
PROCESS 1.1.2 CHECK BOOK IN STOCK
(NOTE:Polymorphic behaviour elided)
PROCESS 1.1.2 CHECK CREDIT AUTHORISATION
PROCESS 1.1.5 CHECK INVENTORY FOR REPRINTING
BEGIN
DOWHILE there are more order-items in valid-order-details
FIND inventory-item in inventory-items with book-code =book-code
in order-item and warehouse-ID matching warehouse-ID provided
as input to this process
READ inventory-item record
SUBSTRACT quantity-ordered from total-in-stock
WRITE inventory-item record
FIND book in books with book-code =book-code in order-item
READ book record
SUBSTRACT quantity-ordered from total-in-stock
WRITE book record
IF total-in-stock<reorder-threshold
low-inventory-message=book-code+total-in-stock
+ "time to reprint"
DISPLAY low-inventory-message
ENDIF
ENDDO
END
PROCESS 1.1.4 ENTER ORDER
BEGIN
...
DO WHILE there are more order-items in valid-order-details
CREATE order-item record from next order-item in valid-order-details
APPEND order-item record to ORDER-ITEMS
END DO
CREATE order record from valid-order-details and warehouseID
APPEND order record to ORDERS
...
order-response="order-accepted"
DISPLAY order-response

Figure A26.

```
PROCESS 1.1.1 EDIT ORDER DETAILS
...
    FIND customer in CUSTOMERS with customer-
ID =customer-ID
          in order-details
    IF record cannot be found
          order-response="no such customer"
    DISPLAY order-response
    EXIT
    ENDIF
...
END
```

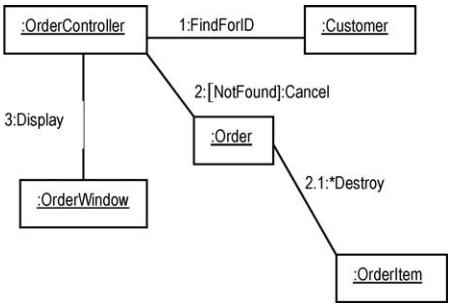


Figure A27. Collaboration diagram for incorrect customer details.

```
PROCESS 1.1.2 CHECK BOOK IN
STOCK
BEGIN
...
ELSE
    IF order-type="walk-In"...
       IF inventory-quantity < quantity-
   ordered
          order-response="Not enough
books to fill your              order

          EXIT
          ENDIF
       ELSE*Type Order is PhoneOr Mail"...
          IF  inventory-quantity < quantity-ordered
           enough-books="No"
          ENDIF
          ...
          IF enough-books="No"
           order-response = "Not  enough books to fill your order"
DISPLAY order-response
END
```

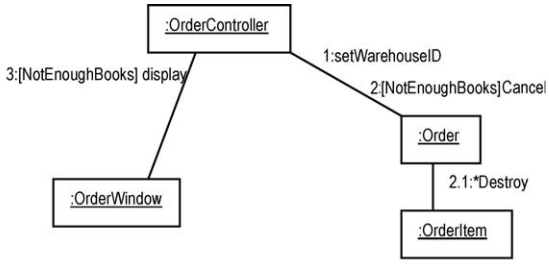


Figure A28. Collaboration diagram for insufficient stock to fill in order.

data are not correct. One would have a similar problem if the data of the books ordered were incorrect. It is not included, but the behaviour is covered.

There is another optional flow when there are not enough books in stock to fill in the order. The collaboration diagram (Figure A28) is very simple because the polymorphic behaviour has been elided.

The polymorphic behaviour in the class order can be seen in the following diagrams (Figures A29–A31).

At this point the use case "Record Book Order" is covered. Diagrams for the use case "Record Book Order" from New Customer (Figure 32).

Note. The use case "Check Payment" is not included. It would be modelled in the same fashion as the part in "Record Book Order" where Stock is checked.

```
PROCESS 1.1.2 CHECK BOOK IN STOCK
BEGIN
IF back-orderOK = "YES"
    DISPLAY (to bubble 1.1.3) valid-order-details +
...
END
```

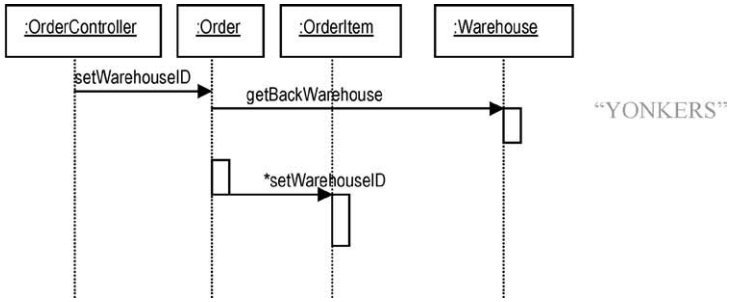


Figure A29. Check book in stock.

```
PROCESS 1.1.2 CHECK BOOK IN STOCK
BEGIN
ELSE
    IF order-type="walk-In"
        DOWHILE there are more order-items in valid-order-
            FIND inventory-item in INVENTORY-ITEM
            =book-code in valid-order-details and
            warehouseID=the location where this was walk
            READ inventoryitem record
            IF inventory-quantity < quantity-ordered
                order-response="Not enough books to
                EXIT
            ENDIF
        ENDDO
        DISPLAY (to bubble 1.1.3) valid-order-details + warehouseID
        ...

END
```

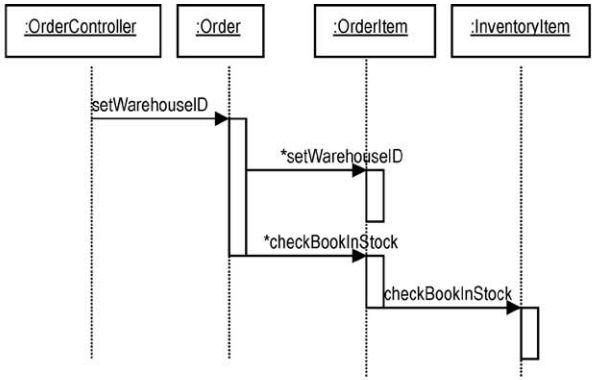


Figure A30. Check book in stock.

```
PROCESS 1.1.2 CHECK BOOK IN STOCK
BEGIN
...
ELSE *Type Order is PhoneOr Mail"
        enough-books="Yes"
    REPEATUNTIL there are no more warehouses in
        or enough-books="Yes"
        DO-WHILE there are more order-items in valid-
        FIND inventory-item in INVENTORY-ITEMS
            =book-code in valid-order-details and
            warehouseID = warehouseID of current
            warehouse record
        READ inventory-item record
        IF  inventory-quantity < quantity-ordered
            enough-books="No"
        ENDIF
        ENDDO
    ENDREPEAT
    IF enough-books="No"
        order-response = "Not  enough books to fill your
        DISPLAY order-response
    ELSE
        DISPLAY (to bubble 1.1.3) valid order details +
            warehouseID of current  warehouse record
        ENDIF
    ENDIF
        order-response = "Not  enough books to fill your order"
        DISPLAY order-response
END
```

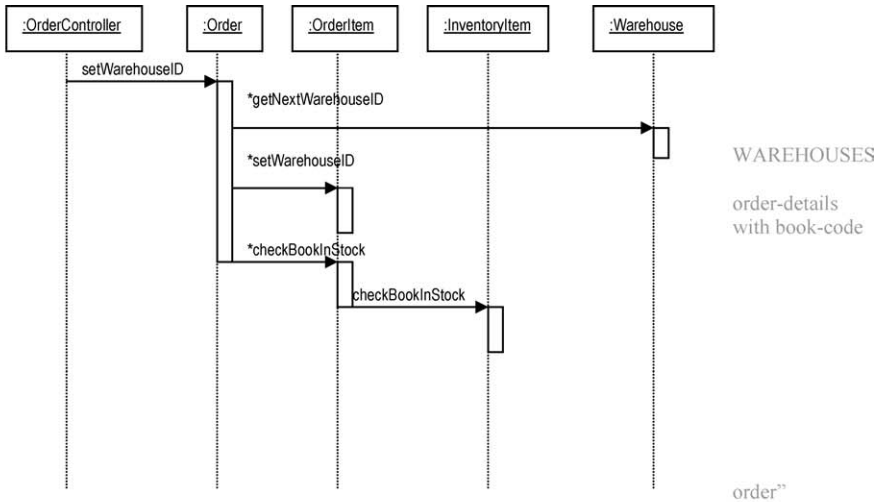
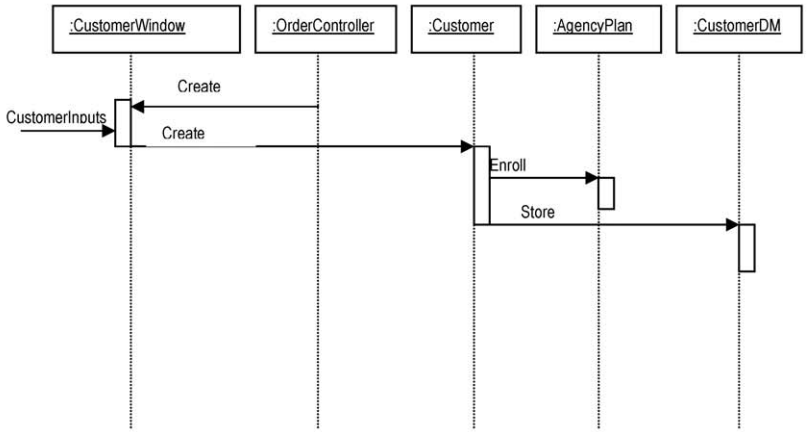


Figure A31. Check book in stock.

PROCESS 1.1 EDIT ORDER DETAILS
BEGIN
IF Tentative-customer-ID=”new”
customer-ID=next available customer-ID
credit-limit=standard credit limit
current-balance=0
agency-plan-level=0
customer=customer-ID+customer-name+(company-
customer-address+current-balance+credit-limit+
agency-plan-level
APPEND new customer record to CUSTOMERS
...
END

PROCESS 1.3 ENROLL CUSTOMER IN AGENCY
Precondition-1
There is a customer in CUSTOMERS which matches
with agency-plan-level=0 and with agency-plan-
and agency-plan-choice less than or equal maximun agency level
Postcondition-1
agency-plan-level in customer is set to agency-plan-choice



name)+

PLAN

customer-ID
choice >o

Figure A32. Sequence diagram for a new customer.

References

Allen, P. and Frost, S. 1998. *Component Based Development for Enterprise Systems. Applying the SELECT Perspective*, SIGS Books. Cambridge University Press.

Avison, D. and Fitzgerald, G. 1995. *Information Systems Development: Methodologies, Techniques and Tools*. McGraw-Hill.

Berki, E. 2001. Establishing a scientific discipline for capturing the entropy of systems process models, CDM-FILTERS: A computational and dynamic metamodel as a flexible and integrating language for the testing, expression and re-engineering of systems, Ph.D. Thesis, University of North London.

Berki, E. and Georgiadou, E. 1996. Resolving Data Flow Diagramming deficiencies by using Finite State Machines, *Proceedings of 5th Software Quality Conference*, University of Abertay, Dundee, Scotland.

Berki, E., Georgiadou, E., Sadler, C., and Siakas, K.V. 1997. A methodology is as strong as the user participation. *Proceedings of International Symposium on Software Engineering in Universities—ISSEU'97*, Rovaniemi, Finland.

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide*. Addison-Wesley.

Coad, P., North, D., and Mayfield, M. 1995. *Object Models. Strategies, Patterns and Applications*. Prentice Hall.

del Brezo Cordero, M. 1998. Software design re-engineering—migrating from structure design to UML, B.Sc. Erasmus Exchange Project.

Ezran, M., Morisio, M., and Tully, C. 2001. *Practical Software Reuse*. London, Springer.

Georgiadou, E. and Sadler, C. 1995. Achieving quality improvement through understanding and evaluating Information Systems Development Methodologies, *Proceedings of the 3rd International Conference on Software Quality Management, SQM'95*, Seville, Spain, April 1995.

Henderson-Sellers, B., Graham, I.M., Swatman, P., Winder, R.L., and Reenskaug, T. 1996. Using object-oriented techniques to model the lifecycle for OO software development, *Proceedings of OOIS*, 1996, p. 211.

Jackson, M. 1994. Problems, methods and specialisation, *Software Engineering Journal* 11(6): 57–62.

Jayaratna, N. 1994. *Understanding and Evaluating Methodologies, NIMSAD: A Systemic Approach*. McGraw-Hill.

Law, D. 1998. Methods for comparing methods: Techniques in software development, NCC Publications.

Law, D. and Naeem, T. 1992. DESMET: Determining and evaluation methodology for software methods and tools, *Proceedings of BCS Conference on CASE—Current Practice, Future Prospects*, Cambridge, England.

Manninen, A. and Berki, E. 2004. An evaluation framework for requirements management tools, *Proceedings of the 13th SQM Conference*, BCS, Canterbury.

Mohamed-Bakry, W. 1999. Specifications reuse via homogeneous interpretation of concepts, *Proceedings of BITWorld 99, Conference on Business Information Management*, South Africa, ed. A. Bytheway.

Presuman, R. 2004. *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

Ross, M., Staples, G., and Hawkins, C. 1997. From temporary awareness to crisis management—the Year 2000 problem, *Software Quality Journal*, December.

Sutcliffe, A.G. and Carroll, J.M. 1999. Designing claims for reuse in interactive systems design, *International Journal of Human–Computer Studies* 50(3): 213–242.

Wilkie, G. 1993. *Object-Oriented Software Engineering. The Professional's Developers Guide*. Addison-Wesley.

Yourdon, E. 1988. *Modern Structured Analysis*, Yourdon Press Computing Series. Pearson Education.

Elli Georgiadou is a Principal Lecturer in Software Engineering at Middlesex University, London. Her teaching includes Software Metrics, Methodologies, CASE and Project Management. She is engaged in research in Software Measurement for Product and Process Improvement, Methodologies, Classifications, Metamodelling, Cultural Issues and Software Quality Management. She is a member of the University's Global Campus project (developing and offering ODL). She has extensive experience in academia and industry, and has been active in organising/chairing conferences and workshops under the auspices of the British Computer Society, the ACM British Chapter and various European programmes for Technology Transfer and development of joint curricula. She has engaged in developing a pedagogic framework as well as the development of materials. She designed and carried out evaluations of various ODL initiatives in the UK, Greece, Spain, Finland, Hong Kong and Cyprus.

Dr Eleni Berki is a researcher/project specialist at the Information Technology Research Institute in Jyväskylä University, Finland. She completed her PhD in Process Metamodelling and Information Systems Method Engineering in 2001, in United Kingdom. Her teaching and research interests include Metamodelling and Method Engineering, Virtual Communities, Information and Communication Technologies, Multidisciplinary Approaches for Software Engineering, Knowledge Representation Frameworks and Requirements Engineering. She has worked as systems analyst and designer in industry, and has a number of academic and industrial project partners in many countries. She has been active in the development, delivery and coordination of virtual and distance learning initiatives in collaboration projects in European and Asian countries. She has co-authored and published about her research and teaching projects in world congresses, international forums and journals and has given a number of talks in international conferences. She has been a professional member of the Institute of Electrical and Electronic Engineers (IEEE), the British Computer Society (BCS) and the United Kingdom Higher Education Academy; and a Visiting Lecturer and Researcher in the University of Sorbonne, Paris-1, France and University of Crete, Greece.

Maria del Brezo Cordero obtained her B.Sc. in computer engineering from the University of Valladolid in1999 completing her final year project under the Erasmus European Programme at North London University, UK. She obtained her M.Sc. in computer engineering from the University of Valladolid in 2001, again completing her Masters dissertation under the Erasmus Programme at Lund University, Sweden. She has been working as a Software Development Engineer in Paris since 2002. Her research interests are in the comparison of OO methods and in migration of systems to the OO paradigm.

Margaret Ross is Professor of Software Quality and is a Principal Lecturer at Southampton Institute.

Margaret's area of interest is quality within a computing context. She has also been Conference Director since 1992 of the annual series of Software Quality Management international conferences, aimed at benefits to industry, and since 1995 of the annual series of international educational INSPIRE conferences. She has edited twenty books, Margaret Ross is involved with the British Computer Society (BCS), currently holding various positions including that of nationally elected member of the BCS Council, Chair of the Hampshire Branch, Vice Chair and Secretary of the BCS national Quality Specialist Group, member of the BCS Ethics Panel and the TickIT Committee.

Geoff Staples was Head of Computing in the Faculty of Technology at Southampton Institute until his retirement. He has chaired the annual series of SQM and INSPIRE annual international conferences since their inception. He is the Chairman of the British Computer Society's Quality Specialist Group, and is a member of the TickIT Committee.