Complexity and Maintenance: A Comparative Study of Object-Oriented and Structured Methodologies

A thesis presented in partial fulfilment of the requirements

for the degree of

Master of Business Studies

in Information Systems at

Massey University

Michael Andrew Bailey

1996

ABSTRACT

Maintenance has been found to be one of the most expensive phases in the life of an information system. It has been suggested that the use of object-oriented methods instead of traditional structured methods may be one way of reducing the cost of maintenance required for an information system.

This thesis is an attempt to determine whether the object-oriented approach does in fact undergo a relatively smaller increase in complexity when subjected to a change in specifications than a similar system that is developed using a "structured methodology", and is therefore easier to maintain.

The methodologies used in this study were Yourdon's (1989) Modern Structured Methodology and Booch's (1994) Object-Oriented methodology. The analysis phase of both methodologies were applied to the same case study twice in order to evaluate the effects of a change in the system's specifications.

Once the two models for each methodology were complete, various metrics were applied to the structured system and a separate set of metrics were applied to the object-oriented system. The results of the models and the metrics were then analysed and validated in order to determine which system suffered a smaller proportional increase in complexity as a result of the changes to the system.

It was found that overall, the object-oriented system proved to undergo a smaller increase in complexity, and it was therefore easier to maintain as a result of the changes than the structured system.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my Thesis Supervisor, Dr. Richard (Dick) Whiddett for providing invaluable assistance and support throughout the entire preparation of this Thesis. I am very grateful for his input into this thesis.

I would like to thank Mr. Barry Jackson for letting me borrow a guide on how to use Oracle's CASE tool. Without this I would not have been able to complete the structured models. I am also grateful for being able to use the Trusty Furniture Company case study.

I would also like to thank the students who provided the two structured systems used in the validation of this study.

A special thanks to my family for putting up with me for yet another year.

Finally, I would like to thank everyone else who has helped me during the preparation of this thesis who I haven't specifically mentioned (you know who you are).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1. Background

A common idea among many authors in the object-oriented field is the claim that an object-oriented system should be more maintainable since object-oriented systems are generally less complex than similar systems developed using a structured methodology. For example, Meyer (1981) says that "apart from its elegance, such modular object-oriented programming yields software products on which modifications and extensions are much easier to perform than with programs structured in a more conventional procedure-oriented fashion" (p.178). A further illustration is provided by Henry and Humphrey (1994) who showed that "building applications with object-oriented languages (like C++ or Objective C) results in final systems that are much more maintainable than systems constructed with procedural languages (like Pascal or C)" (p.2).

1.2. Thesis

This project is designed to find out *whether an object-oriented system does in fact undergo a relatively smaller increase in complexity when subjected to a change in specifications than a similar structured system, and is therefore easier to maintain.* However, this study will concentrate on the results of the systems analysis phase of a case study, rather than the results obtained from systems design or a programming language.

Thus, this study is more interested in the models of the system that are developed as a result of the analysis using both an object-oriented methodology and a structured methodology. The reason that this study was limited to the systems analysis phase rather than including the design phase is because the analysis phase is capable of showing a view of the system that is unaffected by physical implementation issues. Had the design phase been included, then these physical problems with the implementation of the system could colour the results. Thus, by using only the analysis phase it was possible for this study to effectively show the effects of a change on the systems complexity.

1.3. Methodology

The hypothesis was tested by applying a number of different metrics that were designed to measure the complexity of a system to an initial model and then to a model which has been modified. It was decided to use Yourdon's (1989) Modern Structured methodology to model the structured system and Booch's (1994) Object-Oriented methodology to model the object-oriented system. The reason that these methodologies were chosen was due to previous experience in both methodologies.

One set of metrics was applied to the structured models, and another set of metrics was then applied to the equivalent object-oriented models of the same system.

McCabe's (1976) *Cyclomatic Complexity* metric and a modified version of Chidamber and Kemerer's (1994/1991) *Coupling Between Objects* metric were used to measure the complexity in the structured model because both of these metrics were capable of being applied to either a dataflow diagram or an entity relationship diagram.

Similarly, Chidamber and Kemerer's (1994/1991) MOOSE metric suite was used to measure the complexity of the object-oriented models. The ability of the metric to be applied to a diagram rather than to program code was an important consideration in this study, since only the diagrams produced as a result of the analysis phase were studied.

The reason that different metrics were used is that currently there are no specific metrics that can be applied to both structured and object-oriented systems. That is, the metrics are methodology specific, so a structured metric cannot be applied to an object-oriented system for example. Thus, it is necessary to convert the change in complexity results into a percentage value to facilitate comparisons between the structured and the object-oriented models.

An attempt was made to validate the results obtained with each of the metrics where possible in order to determine whether or not the metrics are capable of giving consistent results in similar systems to the one developed in this study. This process was done by comparing the similarities between the results gained in this study to the results obtained from a system developed by extramural students for the structured model, and a model of a similar ordering system developed by Booch (1994) for the object-oriented model.

Furthermore, an attempt was made to validate the models that have been developed in this study since the actual models were not implemented. Each of the models were validated by comparing the similarities between the models developed in this study to the models developed by the extramural students and Booch (1994).

Finally, the results of the change in complexity from the structured system were compared with the object-oriented results on a percentage change basis. The results of this comparison helped to enable a conclusion to be reached as to whether the structured system or the object-oriented system underwent a relatively smaller increase in complexity when subjected to modification, and therefore which approach delivered systems that are easier to maintain on a long term basis.

Since this study followed a case study approach it is fair to say that the results obtained here are specific to the case study used to develop the models in this study. Despite this, it may be possible to draw some general conclusions which may be able to be applied to other systems.

1.4. Structure of the Thesis

This thesis will begin with a discussion of maintenance and the metrics used in this study. Following this, a brief overview will be given of the case study used. The next section of this thesis will concentrate on defining structured analysis definitions and techniques with particular emphasis on Yourdon's (1989) methodology.

The following chapter will focus on what was done in order to create the structured models in this study as well as how the metrics were applied to the structured models.

Many of the object-oriented definitions and techniques will be introduced in the next chapter along with an overview of Booch's (1994) methodology. Following this, a discussion on what was done in order to create the object-oriented models will be given, along with an overview of how the MOOSE metric set was applied to the object-oriented models.

The next part of this thesis concentrates on the results and the validation of these results. Finally, the conclusion summarises the findings of this study in order to determine whether or not the hypothesis was proven and suggests possible ideas for future work.

CHAPTER 2: OVERVIEW OF MAINTENANCE AND MEASUREMENT

2.1. Introduction

This chapter is intended to give an overview of maintenance, the methods of measuring maintenance, and some of the concepts covered by both of the methodologies which were used to create the system which was used to test maintainability. In addition, this chapter attempts to define and justify the purpose of this thesis.

2.2. Definitions of Maintenance

Software maintenance appears to have several similar meanings. The previous statement is supported by the number of similar definitions of software maintenance in the literature. One such definition is given by Schneidewind (1987), who defines maintenance as the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" (p.303).

Another definition of maintenance is given by the United States National Bureau of Standards, who define maintenance as "...the performance of those activities required to keep a software system operational and responsive after it has been accepted and placed into production.... It is the set of activities which result in changes to the originally accepted (baseline) product set" (Pickard and Carter, 1993, p.A-36).

It can be seen from the definitions given by Schneidewind (1987) and the United States National Bureau of Standards (1993) that some similarities exist in these two definitions. For instance, both Schneidewind (1987) and the United States National Bureau of Standards (1993) refer to the idea of making changes to the software after it has been accepted. Both authors also refer to the idea of modifying the software in order keep it operational and responsive to changes in the system's environment. While this is not an exhaustive list of definitions, it does demonstrate that there are a number of similar definitions that exist for software maintenance.

In comparison to the previous definitions of maintenance, Booch (1994) argues that "it is maintenance when we correct errors; it is evolution when we respond to changing requirements" (p.6). Thus, many definitions of maintenance including the above definitions, describe what should really be known as evolution as they are concerned with improving the software over time rather than fixing errors.

A similar argument is also put forward in van Genuchten et al. (1992), who state that they are "aware of the fact that maintenance is an inappropriate term for software because maintenance is in fact prolonged development" (p.507). Further to this point, van Genuchten et al. (1992) argue "that the terms software development and maintenance could be better replaced by the term software evolution" (p.507).

2.3. Definitions of Maintainability

Maintainability is another concept which needs to be defined as it differs from the concept of maintenance. Maintainability is defined by Schneidewind (1987) as being "the ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements" (p.303). Based on Schneidewind's (1987) definitions of maintenance and maintainability, it appears that the main area in which maintainability differs from maintenance is that maintenance refers to the actual process of modifying the software, whereas maintainability appears to refer to the ease with which these modifications take place. Thus, Schneidewind (1987) implies that maintainability is concerned with the level of maintenance that is needed in order to modify the software in such a way as to meet the new requirements.

Another view of maintainability is put forward by Card and Glass (1990), who define maintainability as the "effort required to locate and fix an error in an operational program" (p.8). As was seen in the previous definition, maintainability is defined as a process. However, unlike the previous definition, the definition provided by Card and Glass (1990) defines maintainability in terms of the removal of errors from an operational program rather than the modification of that program. It is this point which differs between the two definitions which have so far been put forward.

Sharble and Cohen (1993) state that "a software system is maintainable if it can be modified to adapt to external changes, such as changes in hardware or operating systems, or to correct deficiencies and improve performance" (p.61). Further to this, in order "to achieve a high degree of maintainability, a system should have weak coupling between parts and have interactions that rely only on the external interfaces of those parts" (Sharble and Cohen, 1993, p.61). This definition is similar in many ways to the previous definitions of maintenance as put forward by Schneidewind (1987) and the United States National Standards Bureau (1993) especially in terms of the ideas relating to the ability to modify the system to adapt to changes and to improve performance. Sharble and Cohen's (1993) definition of maintainability also shares some similarities with those of Schneidewind (1987) and Card and Glass (1990) in that it refers to the processes which are synonymous with maintainability. An example of one of these processes is the process of modifying the system.

Further to the above definitions, Harrison et al. (1982) state that "the degree to which characteristics that impede software maintenance are present is called software maintainability and is driven primarily by software complexity, the measure of how difficult the program is to comprehend and work with" (p.65). This definition shows the differences between maintenance and maintainability as well as making a link between maintainability and complexity. Unlike the definitions provided by the previous authors, Harrison et al. (1982) try to make a clear distinction between maintenance and maintainability.

Pickard and Carter (1993), claim that "the consensus seems to be that maintainability can be thought of as consisting of three attributes: understandability, modifiability, and testability" (p.A-36). Such a statement appears to include most of the aspects that the other definitions discussed here covered.

2.4. Overview of Maintenance

This section will give a brief overview of a number of important areas in maintenance.

2.4.1 Cost of maintenance.

Maintenance is an important part of a system's lifecycle, however it is also one the most expensive areas of a system's lifecycle. The previous statement is supported by Harrison et al. (1982) who state that "recent estimates suggest that about 40 to 70 percent of annual software expenditures involve maintenance of existing systems" (p.65). A further example of the cost of maintaining a system is given by Yau and Collofello (1985) who say that the cost of maintenance activities ranges "from 40 percent to 80 percent of the total cost during the lifecycle of a large scale software system" (p.849).

Burch (1992) says that "the cost of maintenance has increased steadily in the past 25 years.... Some organisations spend 80 percent or more of their systems budget on software maintenance" (p.786). Thus, it is generally agreed that maintenance contributes to a significant portion of the cost of a system throughout its life. This leads to the question of why systems maintenance costs so much.

Burch (1992) gives one reason as to why maintenance costs so much of an organisations system budget. "One of the major reasons why systems maintenance takes such a large bite out of the systems budget is because of the excessive effort spent on trying to maintain poorly structured and documented software" (Burch, 1992, p.789).

Such a statement suggests that if an organisation wants to reduce the amount that it spends on systems maintenance, then it would be wise to spend more time structuring and documenting the software during the development of the system. Thus, rather than viewing maintenance as an afterthought, developers need to build maintainability into the analysis and design phases of the Systems Development Life Cycle (SDLC) rather than waiting till after the system is implemented.

2.4.2 Concepts of maintenance.

Fournier (1991) argues that "the traditional concept of maintenance (a programmer fixing a bug) is not an accurate picture of the entire maintenance process.... In fact, various surveys conducted demonstrate that more time is spent enhancing the system to meet changing user needs than on any other type of maintenance activity" (p.214).

Thus, the extra time which is needed to enhance the systems results in further maintenance costs. Developers could follow an approach similar to Total Quality Management (TQM) whereby they attempt to do things "right the first time", thus lessening the need for costly maintenance at a later stage in the information system's life. In this case, it would be necessary to extensively involve the users in the development process in order to reduce the number of changes that may be needed at a later stage.

2.4.3 Obsolescence and maintenance.

Despite the previous statement, it must be noted that it is not possible to completely eliminate the need for maintenance as a result of changing user needs, as to do so would lead to an obsolete system that would be of no value to the users. While maintenance may be a burden on an information system's budget, it is a necessary burden, as without maintenance, an information system would cease to evolve.

Thus, without maintenance the useful life on an information system is significantly reduced. This statement is supported by Fournier (1991) who says that "as the years go by, the system will probably be frequently modified to meet new user requirements or to adapt to new hardware/software/networking configurations.... Therefore, symptoms of obsolescence might eventually surface, such as increasing maintenance costs and a decreasing business value" (p.230).

Further to the previous point, Fournier (1991) goes on to say that "this might be especially true if the maintenance process has not been performed in a controlled and rigorous manner" (p.230). Thus, it is important to control and monitor the maintenance process in order to limit the effects of obsolescence on the information system.

Obsolescence is inevitable for all information systems no matter how good their maintenance programme is. According to Fournier (1991) "there are two types of obsolescence: functional and technological" (p.230). In order to better understand obsolescence it is necessary to define both types of obsolescence. "Functional obsolescence arises when the system no longer adequately fulfils its mission or the business environment has evolved so drastically that the original mission no longer holds true in light of the new business functions that must now be supported" (p.230). In comparison, "technological obsolescence occurs when the technology that was used to implement the system becomes inefficient, unreliable, and outdated" (Fournier, 1991, p.230). Thus, it can be seen from these two definitions that there are a large number of factors which can contribute to the obsolescence of an information system.

2.4.4 Strategies to remedy obsolescence.

In order to reduce the effects of obsolescence, Fournier (1991) offers four different strategies that can be used to remedy the problem. These strategies are: restoration, expansion, elimination, and replacement.

2.4.4.1 Restoration.

Restoration "entails correcting technical deficiencies while maintaining the status quo on the functions supported by the system" (Fournier, 1991, p.230). Thus, the restoration process is used to optimise the system in order to help it make more efficient use of the technology that the system was implemented with.

2.4.4.2 Expansion.

According to Fournier (1991) "the expansion process entails the addition of new functions around or as a front end to the current system while maintaining the status quo on the technical architecture surrounding the system" (p.230). Thus, the expansion process is used in order help the information system evolve as a result of changing user requirements. Unlike, the restoration process which concentrates on making changes to the technology, the expansion process is more concerned with making changes to the functions in the system. Therefore, the expansion process is an attempt to reduce the problem of functional obsolescence.

2.4.4.3 Elimination.

Another strategy which was proposed by Fournier (1991) was the elimination strategy. In this strategy the organisation discontinues the system "without a replacement since it achieves a very low business value" (Fournier, 1991, p.231).

Such a strategy is of obvious use when a system is considered to be obsolete and it no longer supports the business functions of the organisation.

2.4.4.4 Replacement.

The final strategy which was put forward by Fournier (1991) is that of the replacement strategy. According to Fournier (1991), "the existing system is replaced by acquiring a commercial package or building a new application in-house" (p.231). Such a replacement strategy would be of use to an organisation where the existing system has evolved to such an extent that the entire system needs to be replaced. Thus, the use of this strategy would effectively mean the end of the current systems life.

Both the elimination and the replacement strategies appear to be designed to combat both technological and functional obsolescence as they both lead to the end of the current systems useful life.

2.4.5 Non-monetary costs of maintenance.

However, not all of the costs involved in maintenance are monetary costs. An example of one of these non-monetary cost is the opportunity cost that can arise if "the system becomes so fragile that data-processing managers are reluctant to change it" (Martin and McClure, 1983, p.7).

In a situation such as the one described in the previous sentence, "any change has unforeseen consequences which often cause problems elsewhere, annoy users, and waste precious personnel resources" (Martin and McClure, 1983, p.8). The opportunity cost in this situation is that by choosing to forego the maintenance, the organisation does not get the full benefit of its system. Thus, it can be said that, "computers offer the promise of enormous improvements in business efficiency.... However, the promise will not be fulfilled unless the best techniques are used for achieving maintainability" (Martin and McClure, 1983, p.8).

2.4.6 Types of maintenance.

A common theme in much of the literature written on software maintenance is the distinction between several specialised types of maintenance. An example of this distinction is given by Burch (1992) who categorises maintenance "into the following four types:

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventive maintenance"

(p.786).

2.4.6.1 Corrective maintenance.

The first type of maintenance that will be defined is that of corrective maintenance. According to Burch (1992), "corrective maintenance is the less noble and more burdensome part of systems maintenance, because it corrects design, coding, and implementation errors that should never have occurred" (p.786). The need for such maintenance can usually "be traced back to poor application of the Systems Development Life Cycle" (Burch, 1992, p.786). Swanson (1976) on the other hand, provides a simpler definition of corrective maintenance by saying that it deals "with failures in processing, performance, or implementation" (p.4).

2.4.6.2 Adaptive maintenance.

The second type of maintenance is adaptive maintenance. "Adaptive maintenance is performed to satisfy changes in the processing or data environment and meet new user requirements.... The environment in which the system operates is dynamic; therefore the system must continue to respond to changing user requirements" (Burch, 1992, p.787). Thus, adaptive maintenance is the on going maintenance that occurs as the system evolves. Further to the previous point, Burch (1992) states that "generally, adaptive maintenance is good and inevitable.... Too much of it, however, may mean that phases of the Systems Development Life Cycle were not thoroughly and properly performed" (p.787).

2.4.6.3 Perfective maintenance.

Perfective maintenance is the third type of systems maintenance that is performed. In simple terms, "perfective maintenance enhances performance or maintainability" (Burch, 1992, p.787). Perfective maintenance also allows the system to meet user requirements that were unrecognised before (Burch, 1992). Thus, this type of maintenance is an attempt to "perfect" the system, so that it is easier to maintain in the future as well as making the system perform better both in the present and in the future. An example of what activities take place during this type of maintenance is given by Burch (1992), who says that "this maintenance activity may take the form of reengineering or restructuring software, rewriting documentation, altering report formats and content, defining more efficient processing logic, and improving equipment operating efficiency" (p.787).

2.4.6.4 Preventive maintenance.

The final type of maintenance is preventive maintenance. This type of maintenance is an attempt to purge any potential problems from the system before they occur. According to Burch (1992) "preventive maintenance consists of periodic inspections and reviews of the system to uncover and anticipate problems" (p.788). Further to this, Burch (1992) states that "while not requiring immediate attention, these defects, if not corrected in their minor stages, could significantly affect either the functioning of the system or the ability to maintain it in the near future" (p.788).

2.4.7 An alternative categorisation of maintenance.

An alternative to Burch's (1992) categorisation of maintenance is given by Perry (1981). Perry's (1981) defines seven different types of maintenance. The seven types of maintenance are:

- Corrective maintenance
- Deferred maintenance
- Preventive maintenance
- Emergency maintenance
- File maintenance
- Program maintenance
- Scheduled maintenance.

2.4.7.1 Corrective maintenance.

As with Burch (1992) and Swanson (1976), Perry (1981) includes corrective maintenance in his classification of the types of maintenance. Perry (1981) defines corrective maintenance to be "maintenance that is conducted for the purpose of eliminating and existing error or problem" (p.1). This definition is similar to the definitions given by Burch (1992) and Swanson (1976), in that it deals with the concept of correcting errors that exist in the current system.

2.4.7.2 Deferred maintenance.

Deferred maintenance is "maintenance that is needed but is postponed until appropriate resources are available" (Perry, 1981, p.1). This is one of the classification which does not seem to have an equal in the types of maintenance put forward by Burch (1992) and Swanson (1976). Such a categorisation takes into account that an organisation may not be able to afford to use valuable resources to fix small problems due to various constraints at a particular point in time. Thus, the organisation's management are able to make a decision to defer the maintenance until the necessary resources are available. It is important to note that such maintenance must be done at some time in the future and not just left to be forgotten about. The reason for the previous statement is that otherwise the system will end out with of a lot of smaller problems that turn into a larger problem which may require what Perry (1981) calls emergency maintenance.

2.4.7.3 Preventive maintenance.

Preventive maintenance is another area in which there are similarities between Burch (1992) and Swanson (1976). Perry (1981) defines preventive maintenance as "maintenance that occurs in anticipation of problems.... For example, an application may install a fix to prevent a problem that has already occurred in another application" (p.1).

The example given by Perry (1981) suggests that the problem can be identified by similar problems occurring in other systems. Such problem identification can be seen as a way of learning from the mistakes of others. This definition of preventive maintenance is indeed the same as that put forward by Burch (1992) as it tries to remove the possibility of the problem occurring in the system. Thus, it appears as though there is a general agreement amongst the authors in this area of maintenance, as there is a standard definition for preventive maintenance.

2.4.7.4 Emergency maintenance.

Emergency maintenance is defined by Perry (1981) as being "unscheduled maintenance that is designed to eliminate a current problem situation which has stopped or otherwise affected successful operation of the application" (p.1). Emergency maintenance is in some ways similar to parts of Burch's (1992) and Swanson's (1976) corrective maintenance. An example of this is given by Burch (1992), who says that "corrective maintenance usually involves an urgent or emergency condition that calls for immediate attention" (p.786).

2.4.7.5 File maintenance.

Perry (1981) says that file maintenance is "the addition, modification, or deletion of systems specifications" (p.1).

This type of maintenance can be compared to perfective maintenance as described by Burch (1992) and Swanson (1976) because it is concerned with trying to "perfect" the system so that it will work without any problems at all. However, Perry (1981) considers maintenance of the system specifications as a separate form of maintenance as will be seen in the next definition.

2.4.7.6 Program maintenance.

Program maintenance is defined as "the addition, modification, or deletion of program instructions" (Perry, 1981, p.1). This form of maintenance is similar to Perry's (1981) file maintenance category except it is more concerned with "perfecting" the actual program rather than the specifications of the system. It is important to note that Perry (1981) makes a distinction between these two forms of maintenance rather than grouping them together under the title of perfective maintenance, as is the case with Burch (1992) and Swanson (1976).

2.4.7.7 Scheduled maintenance.

The final type of maintenance defined by Perry (1981) is that of scheduled maintenance. Scheduled maintenance is "maintenance that occurs at a predetermined time" (Perry, 1981, p1). This type of maintenance is designed to maintain the system at specific times in order to reduce the chances of a major problem occurring.

Scheduled maintenance appears to be similar to what Burch (1992) calls preventive maintenance as scheduled maintenance also involves the inspection of the system in order to anticipate any problems that might occur. Where these two types of maintenance differ, is that according to Burch (1992), preventive maintenance only inspects and reviews the system in order to uncover problems, whereas scheduled maintenance appears to involve both the inspection and the review as well as the correction of the problem. Thus, preventive maintenance does not appear to try to correct any of the problems that it finds. Instead, preventive maintenance is designed to identify the problems rather than fix them.

2.5. Systems Maintenance Lifecycle

The systems maintenance lifecycle is a series of five steps that are designed to "improve systems maintenance" (Perry, 1981, p.58). An example of the systems maintenance lifecycle is given below in figure 1.

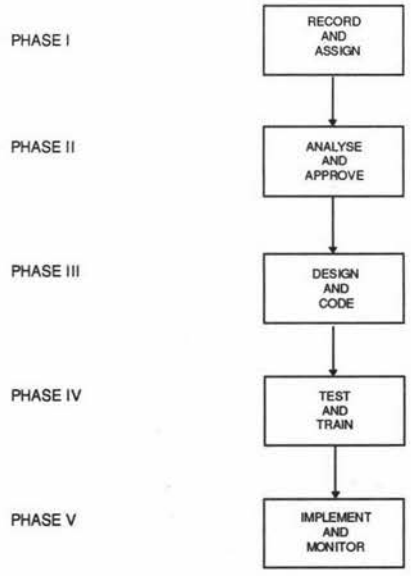


Figure 1: **The Systems Maintenance Life Cycle** (Perry, 1981, p.59)

According to Perry (1981), the "main advantages of the life cycle concept are that the process is subdivided into identifiable and measurable parts.... Prior to the development of the life cycle concept, systems maintenance frequently flowed unchecked.... There were no clear checkpoints; thus, it was difficult to measure progress" (p.58).

While the previous statement suggests that using the systems maintenance life cycle helps an organisation to monitor the maintenance which is being carried out, Perry (1981) notes that "a major argument against using the life cycle concept in systems maintenance is that the size of the change may not warrant the administrative effort" (p.58). Thus, the argument against the use of the lifecycle concept is that if only a small amount of maintenance is required, then it may not be worth the effort of the organisations management to implement the systems maintenance life cycle.

In order to determine whether it is really necessary to use the systems maintenance life cycle for such small amounts of maintenance work, Perry (1981) suggests that "organisations which think their systems maintenance efforts are too small to warrant using the systems life cycle should ask the following question: 'Should an organisation spend time and effort to administer a process which consumes 80 percent of its total resources?'.... If the question is asked this way, the answer is obvious" (p.58).

Such a statement suggests that no matter how small and insignificant the maintenance appears to be, it is always a good idea to use the systems maintenance life cycle concept which was put forward by Perry (1981) as it will help an organisation to monitor the progress of the maintenance which is being carried out. Such monitoring will also help to reduce costs, as the monitoring of the maintenance will help to ensure that resources are efficiently and effectively spent.

In order to be able to understand the systems maintenance life cycle it is necessary to define the processes which occur in each of the five phases. The first of these phases to be defined is phase one which is to record and assign changes.

2.5.1 Record and assign.

In the first phase of the systems maintenance life cycle the necessary system changes are recorded and a person or a group of persons is assigned to be responsible for analysing each change (Perry, 1981). According to Perry (1981) this phase must be completed "for all changes regardless of why they are needed and whether or not they are implemented" (p.58). The reason that these changes are documented is that it serves as a method of tracking the types of changes that need to be made in the system (Perry, 1981). An example of what details should be recorded is given by Perry (1981) who says that "in the case of an error, record the problem or the condition that the change should eliminate" (p.58).

2.5.2 Analyse and approve.

Following the first phase, the second phase of the systems maintenance life cycle consists of analysing and approving the solution to the problem previously identified in phase one. According to Perry (1981), "the analysis process identifies the solution and estimates the time and effort required to implement that solution" (p.60). Once this analysis has been conducted, then the proposed solution is presented to the organisation's management. "At this point management must decide whether or not to implement the proposed solution" (Perry, 1981, p.60).

2.5.3 Design and code.

Once management have approved the proposed solution, the third phase of the systems maintenance life cycle is to design and code the solution to the problem which must be corrected. Perry (1981) states that "this phase of the systems maintenance life cycle is equivalent to the similar phases in the systems development life cycle.... The difference is the magnitude of effort that may go into this phase" (p.60). An example which is given by Perry (1981) to demonstrate the previous point is that some needs can be satisfied by adding or changing one line of code. Such a process "may take only a few minutes, or this phase may take many months of effort" (Perry, 1981, p.60). Thus, the amount of effort needed to complete phase three of the systems maintenance life cycle is determined by the complexity of the proposed solution.

2.5.4 Test and train.

The fourth phase in the systems maintenance life cycle is to test the solution and train those end-users that will be affected by the changes to the system. Perry (1981) says that, "traditionally, management has been concerned about testing during systems maintenance.... The dilemma confronting project personnel is making a tradeoff between testing in ideal circumstances and not testing at all" (p.60). Therefore, management must make clear policies for the organisation in regard to their position on testing during systems maintenance. Such policies will empower the project personnel to make decisions on how much testing is needed for specific maintenance jobs.

However, despite the previous point, Perry (1981) states that "too frequently, individual programmers have made testing decisions without referring to department policy" (p.60). Thus, individuals have made important decisions that affect the outcome of the maintenance without consulting the other members of the project personnel. Therefore, it is important for an organisations management to actively encourage all members of the project personnel to discuss how much testing should be done in accordance with the guidelines that have been provided by management.

As well as the testing that is done in phase four, some training is also done. Perry (1981) says that training people to use the system is as important as testing the system. In addition to the previous point, "if the change will affect the reports applications personnel receive or the procedures they must follow, then they should receive appropriate training or training materials" (Perry, 1981, p.60).

Thus, if the application that is being maintained causes a change in the reports that are produced or the procedures that need to be followed, then it is desirable that those personnel who are affected by the changes receive some form of training. Such training will enable these personnel to perform their functions at an optimum level.

2.5.5 Implement and monitor.

The final phase in the systems maintenance life cycle is to implement and monitor the changes that have been made during the previous phases. "Implementation of a change subjects an organisation to some new risks.... The first risk is that the change will not be implemented correctly and thus will not produce the necessary results; the second risk is that once implemented the change will also impact the unchanged segment of this or another application system, resulting in unanticipated new problems" (Perry, 1981, p.60).

Such a statement demonstrates many of the concerns that an organisations management may have regarding the results of any maintenance to the system. Thus, the management need to find a solution in order to manage the changes effectively. One of the solutions "is to closely monitor the output immediately following a change.... The monitoring process provides management with one additional assurance that the change has been correctly implemented" (Perry, 1981, p.60).

2.6. Measuring Maintenance

There are several possible ways of measuring the level of maintenance needed by a system. The methods used to measure maintainability vary depending on the type of development method used. Thus, systems developed using a structured methodology will use a metric for measuring maintainability in a structured system, whereas a system developed with an object-oriented methodology will generally use an object-oriented maintainability measure. There are some exceptions to this rule however.

2.6.1 Methodology independent metrics.

These particular types of metrics are able to be used on a wide variety of systems without having to be concerned with the methodology used to develop these systems.

2.6.1.1 Number of changes.

One such exception is where the number of changes made to the system after implementation are counted. This method is used by Eid and Rose (1996). Such a method can obviously be used in both structured and object-oriented systems as it does not take any of the special features of each type of system into account. The reason or this is because such a measure is only interested in measuring the number of changes that have occurred, and so the development methodology does not affect the method used to measure the maintainability of the system.

The idea of measuring maintainability by making comparisons based on the number of changes is also put forward by Clapp et al. (1995). However, unlike Eid and Rose (1996), Clapp et al. (1995) expand this idea further by making a wide variety of comparisons. Clapp et al. (1995) note that "metrics that were used during development may be used again during maintenance for comparison purposes (e.g., measuring the complexity of a module before and after modification)" (p.279). By using such metrics it is possible to find where the maintenance problems are originating from. Thus, it would be possible to highlight specific areas that needed improvement during the development of the system.

2.6.1.2 Other methods.

In terms of metrics used to measure changes made during maintenance, Clapp et al. (1995) put forward a number of useful metrics. These metrics include: the number of changes made during maintenance, the cost/effort of those changes, the time required for each change, the number of lines of code added, deleted or modified, and the number of fixes, or enhancements (Clapp et al., 1995). As can be seen from this list, some of these metrics may not be required in all cases, therefore it is necessary to choose only those metrics which are appropriate to the system which is being developed. Each of the metrics put forward by Clapp et al. (1995) can be used for a specific purpose.

2.7. Measuring Object-Oriented Systems

In terms of metrics for object-oriented systems, Sharble and Cohen (1993) state that "metrics could be based on either objects or classes" (p.67). The first reason for this is that "since objects are the components of the system that exist when the system is actually running, metrics based on objects would be influenced by the conditions of execution, and therefore the measurements must be made empirically" (Sharble and Cohen, 1993, p.67). Thus, by observing and experimenting with an object-oriented system, it is possible to make a series of measurements relating to the way that the system runs under a variety of conditions. Such a method could also be applied to a wide variety of structured systems as well, thus, enabling some level of comparison between the two types of systems in terms of performance.

Sharble and Cohen (1993) also argue that metrics that are based on classes, do not measure any particular execution of the system, instead they produce a measurement that applies to all possible scenarios of execution. Further to the previous statement, Sharble and Cohen (1993) state that "since classes are formal descriptions, measurements can be produced deductively from an analysis of the design of the system.... Whereas, objects are the components that exist at run-time, classes are the components that exist during maintenance, and are the units of extension and reuse" (p.67). Thus, it can be concluded that it is better to use classes as a basis for measuring maintainability in an object-oriented system as the use of classes help to give a result which is unaffected by the execution of the system in different scenarios.

2.7.1 MOOSE.

The most common metric for object-oriented systems is the Metric for Object-Oriented Software Engineering (MOOSE) metric as proposed by Chidamber and Kemerer (1994, 1991). The MOOSE metric suite consists of six design metrics which have been designed to measure a number of common areas in the design of an object-oriented system. The six metrics proposed by Chidamber and Kemerer (1994) are as follows:

- *Weighted Methods per Class* (WMC)
- *Depth of Inheritance Tree* (DIT)
- *Number of Children* (NOC)
- *Coupling between Object Classes* (CBO)
- *Response for a Class* (RFC)
- *Lack of Cohesion in Methods* (LCOM)

In order to be able to fully understand the relevance of these metrics to maintenance it is necessary to explain each of the metrics in the MOOSE metric suite.

2.7.1.1 Weighted methods per class.

The first of the metrics described by Chidamber and Kemerer (1994) is that of the *Weighted Methods per Class*.

According to Chidamber and Kemerer (1994) "since methods are properties of object classes and complexity is determined by the cardinality of its set of properties.... The number of methods is, therefore, a measure of class definition as well as being attributes of a class, since attributes correspond to properties" (p.482).

An alternative definition of this metric is given by Sharble and Cohen (1993) who state that "the complexity of a class is given by the complexity of its attributes and its methods.... Weighted methods per class is the sum of the complexities of the methods of a class" (p.67).

There are a number of reasons why the *Weighted Methods per Class* metric is important. The first of these reasons is that "the number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class" (Chidamber and Kemerer, 1994, p.482). This is the main reason that this metric is important to maintenance. Because it is possible to predict the level of maintenance needed on a class by using the *Weighted Methods per Class* metric, it is therefore possible to modify the class in order to minimise the level of maintenance needed.

The second reason is that "the larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class" (Chidamber and Kemerer, 1994, p.482). Such a metric has an obvious impact on maintenance as changes in the parent class will be reflected in the children of that class.

Thus, the *Weighted Methods per Class* metric is useful in determining the number methods in a class and therefore, the effects that any maintenance will have on a class and its children.

Finally, the third reason is that "classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse" (Chidamber and Kemerer, 1994, p.482). In terms of maintenance, such a metric is important because it can be used to determine the level of maintenance needed by each class as well as the effect of any maintenance on other classes (children) that inherit the characteristics of that class (parent).

2.7.1.2 Depth of inheritance tree.

The second metric proposed by Chidamber and Kemerer (1994) is the *Depth of Inheritance Tree* (DIT). According to Chidamber and Kemerer (1994), "depth of inheritance of the class is the *Depth of Inheritance Tree* metric for the class....In cases involving multiple inheritance, the *Depth of Inheritance Tree* will be the maximum length from the node to the root of the tree" (p.482). Thus, "the deeper a class is in the inheritance hierarchy, the greater the number of methods it will inherit" (Sharble and Cohen, 1993, p.67). Further to the previous statement, Sharble and Cohen (1993) state that the "Depth of Inheritance Tree for a class is defined as the number of its ancestor classes" (p.67). Thus, it appears as though the *Depth of Inheritance Tree* metric is designed to provide an insight into the amount of inheritance experienced by a class.

Chidamber and Kemerer (1994) give several reasons why the *Depth of Inheritance Tree* metric is used. The first of these reasons is that "the deeper the class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour" (Chidamber and Kemerer, 1994, p.483).

Another reason that the *Depth of Inheritance Tree* metric is used is that "deeper trees constitute greater design complexity, since more methods and classes are involved" (Chidamber and Kemerer, 1994, p.483). Finally, the third reason that the *Depth of Inheritance Tree* metric is used is that "the deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

The above reasons show that the more complex the class, the harder it will be to maintain the class at a later stage in the systems life as any maintenance will have to take the effects of the inherited methods into consideration as well as the effects of the maintenance being carried out. Despite this problem, the reuse of inherited methods is a benefit that may help to reduce the amount of maintenance that is needed. The reason for this reduction in maintenance is due to the fact that the classes where the methods came from are further up the inheritance tree. Therefore, once the maintenance is carried out on the parent class, it should no longer be necessary to maintain the methods as they are inherited by subsequent classes.

2.7.1.3 Number of children.

The third metric which is described by Chidamber and Kemerer (1994) is that of the *Number of Children* (NOC). Chidamber and Kemerer (1994) state that the *Number of Children* metric "is a measure of how many subclasses are going to inherit the methods of the parent class" (p.485).

In comparison, Sharble and Cohen (1993) say that the "Number of Children for a class is the number of its immediate sub-classes" (p.67). Further to the previous point, "this is an indication of the potential influence a class can have on the system" (Sharble and Cohen, 1993, p.67). Thus, both of these definitions complement each other, as Sharble and Cohen's (1993) definition expands on Chidamber and Kemerer's (1994) definition by stating the influence that the number of children can have on the system as a whole.

Chidamber and Kemerer (1994) provide three different views on the number of children in a class. The first of these views is that the "greater the number of children, the greater the reuse, since inheritance is a form of reuse" (Chidamber and Kemerer, 1994, p.485). Secondly, the "greater the number of children the greater the likelihood of improper abstraction of the parent class.... If a class has a large number of children, it may be a case of misuse of subclassing" (Chidamber and Kemerer, 1994, p.485). Finally, "the number of children gives an idea of the potential influence a class has on the design.... If a class has a large number of children, it may require more testing of the methods in that class" (Chidamber and Kemerer, 1994, p.485).

The viewpoints discussed above help to demonstrate a number of points that will affect the eventual maintenance of the system. The main point that is mentioned by Chidamber and Kemerer is that of an increased use of reuse. More reuse will help to reduce the level of maintenance needed over time as the number of faults in a class are eventually fixed.

Further to the previous point, the *Number of Children* metric "is a measure of the breadth of the inheritance hierarchy, and the Depth of Inheritance Tree metric is a measure of its depth.... Generally, it is better to have depth than breadth, since this promotes reuse and reduces redundancy in the system" (Sharble and Cohen, 1993, p.72).

This comparison with the *Depth of Inheritance Tree* metric shows that the use of the *Number of Children* metric can help developers avoid a wide breadth in the inheritance hierarchy, thus allowing the developers to design systems that increase the level of reuse. The other major point is that the greater the number of children the greater the need for testing, which could also translate into a greater need for maintenance as an increased number of children will increase the complexity of the system.

2.7.1.4 Coupling between objects.

The fourth metric provided by Chidamber and Kemerer (1994) is that of the *Coupling Between Object classes* (CBO). "Coupling between objects for a class is a count of the number of other classes to which it is coupled" (Chidamber and Kemerer, 1994, p.486).

This definition is expanded on by Sharble and Cohen (1993) who say that "coupling can exist between classes that are not related through inheritance.... One class is coupled to another if its methods use the methods or attributes of the other class" (p.67).

According to Chidamber and Kemerer (1994) "excessive coupling between object classes is detrimental to modular design and prevents reuse.... The more independent a class is, the easier it is to reuse it in another application" (p.486). Further to the previous statement, Chidamber and Kemerer (1994) also state that "in order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum.... The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult" (p.486). The previous statement reinforces the idea that changes made in a parent class will have an effect on any of the parents children as a result of inheritance. In addition, it is obviously to the developers benefit if the system is designed to be reusable as it means that any subsequent systems can be built from components which have been thoroughly tested and are therefore less likely to require extensive maintenance.

2.7.1.5 Response for a class.

The fifth metric proposed by Chidamber and Kemerer (1994) is that of the *Response For a Class* (RFC) metric. Sharble and Cohen (1993) state that "the response for a class is the sum of the number of its methods and the total of all other methods that they directly invoke.... This measures a combination of the complexity of a class through the number of its methods, and the amount of communication with other classes" (p.67).

Chidamber and Kemerer (1994) propose several reasons why the *Response For a Class* metric is needed. The first of these reasons is that "if a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester" (Chidamber and Kemerer, 1994, p.487). The second reason is that "the larger the number of methods that can be invoked from a class, the greater the complexity of the class" (Chidamber and Kemerer, 1994, p.487).

The above reasons show that the more methods are involved, the more complicated the class gets. Therefore, as the complexity of the class increases, maintenance will become more difficult.

2.7.1.6 Lack of cohesion in methods.

The final metric proposed by Chidamber and Kemerer (1994) is that of the *Lack of Cohesion in Methods* (LCOM) metric. Sharble and Cohen (1993) say that "this metric attempts to measure the degree of similarity by counting the number of disjoint sets produced from the intersection of the sets of attributes that are used by the methods" (p.67).

Further to the previous point, "the cohesion of the methods in a class increases with the degree of their similarity.... Methods are more similar if they operate on the same attributes" (Sharble and Cohen, 1993, p.67). Chidamber and Kemerer (1994) expand this definition further by saying that the *Lack of Cohesion in Methods* metric is "tied to the instance variables and methods of a class, and therefore is a measure of the attributes of an object class" (p.489). Thus, this metric differs from the previous metrics as it concentrates on measuring the attributes rather than the class itself.

One of the main reasons that the *Lack of Cohesion in Methods* metric is important is that "low cohesion increases complexity, thereby increasing the likelihood of errors during the development process" (Chidamber and Kemerer, 1994, p.489). Thus, in terms of maintenance it is important that there be as little complexity as possible in order to make it easier to perform maintenance at a later stage in the systems life. Further to the previous point, Chidamber and Kemerer (1994) argue that "cohesiveness of methods within a class is desirable, since it promotes encapsulation" (p.489). The reason why a high level of encapsulation is good is that it means that a lot of the complexity can be hidden within the objects, thus making any maintenance easier.

2.8. Measuring Structured Systems

2.8.1 McCabe's cyclomatic complexity.

One of the methods used for measuring the maintainability in a structured system is described by McCabe (1976), who puts forward the idea of a metric based on a graph-theoretic complexity measure. According to Martin and McClure (1983), McCabe's "strategy is to measure program complexity by computing the number of linearly independent paths through a program.... McCabe refers to this number as the cyclomatic number" (p.57).

In terms of structured programs, "cyclomatic complexity can be measured simply by counting the number of compares:

cyclomatic complexity = compares + 1

" (Martin and McClure, 1983, p.57). Compares refers to statements in the program that are designed to evaluate one or more variables, such as an IF...THEN statement. Thus, it is the number of comparisons that are done within each module that determines the value of compares in the equation.

According to Martin and McClure (1983) "complexity evaluation is applied at the module level in a program.... McCabe uses the cyclomatic number to control the "size" of a program and hence its understandability by limiting the cyclomatic complexity of each module in the program to a maximum of 10.... McCabe arrived at 10 as a reasonable limit for cyclomatic complexity after examining several FORTRAN programs" (p.57).

The reason for limiting the cyclomatic number to 10 is that, McCabe "found that modules (and indeed programs containing modules) whose cyclomatic complexity was greater than 10 were generally more troublesome and less reliable.... McCabe suggests that modules with a cyclomatic number greater than 10 should be redesigned and perhaps subdivided into a group of modules" (Martin and McClure, 1983, p.57). While McCabe (1976) originally tested his metric with FORTRAN programs, it can also be used with a wide variety of other programming languages, such as COBOL.

2.9. Comparisons Between the Metrics

In comparison to Chidamber and Kemerer (1994), McCabe's (1976) metric does not cover as many areas of maintenance. The reason that McCabe (1976) only covers one area of maintenance is that the *cyclomatic number* metric is only designed to measure one particular aspect of maintenance, namely program complexity, whereas Chidamber and Kemerer (1993) have proposed an entire suite of metrics designed to cover a wide variety of maintenance issues which are relevant to an object-oriented system.

Trying to make a direct comparison between McCabe's (1976) metric and any of the metrics in Chidamber and Kemerer's (1993) suite is difficult due to the fact that both have been designed to measure maintainability in systems that were developed using different paradigms, that is the structured approach and the object-oriented approach. It is the revolutionary nature of the object-oriented approach which frustrates any possible comparisons between the two. While it may be possible to make some comparisons in terms of what each metric is attempting to measure, such an exercise is ultimately very difficult due to the inherent complexity in the object-oriented approach. One of the advantages of the object-oriented approach is due to the way that it handles this complexity. Despite the previous problems some comparison can be made between McCabe's (1976) *cyclomatic number* metric and Chidamber and Kemerer's (1994) *weighted methods per class* metric.

The similarity between these two metrics is the fact that both have been designed to measure the complexity of their respective systems. McCabe's (1976) metric is designed to measure the complexity of modules in a program, whereas, Chidamber and Kemerer's (1994) metric is designed to measure the complexity of a class.

Chidamber and Kemerer's (1994) *Depth of Inheritance Tree* metric also measures complexity to an extent as it determines the depth of inheritance in an object-oriented design. Thus, the deeper the inheritance in the design, the higher the complexity of the system. This metric is harder to compare with McCabe's (1976) metric due to the fact that McCabe's (1976) metric is trying to measure a different type of complexity.

McCabe (1976) attempts to measure the complexity of a program in terms of the number of conditional statements present in a given program, whereas Chidamber and Kemerer (1994) attempt to measure the depth of inheritance within a given object-oriented system. McCabe's (1976) metric is harder to apply to a diagram than any of the metrics in Chidamber and Kemerer's (1994) object-oriented metrics suite. In Chidamber and Kemerer's (1994) case, it is possible to check the level of inheritance in an object-oriented system just by looking at one of the design diagrams.

While the previous two examples of metrics in Chidamber and Kemerer's (1994) metric suite show that it is difficult to compare an object-oriented metric with a structured metric, they are by no means the only metrics in the MOOSE metric suite that measure complexity to some extent. Other metrics such as the *Response for a Class* and the *Lack of Cohesion in Methods* metrics also measure some complexity as well.

Overall, it can be said that it is very difficult to make direct comparisons between Chidamber and Kemerer's (1994) metric suite and McCabe's (1976) metric. This is despite the fact that several of the components of Chidamber and Kemerer's metric suite are designed to measure complexity, as is McCabe's (1976) metric. The main problem is that it is difficult to make direct comparisons between the metrics as the output obtained from each metric means different things in terms of the level of complexity present in a given system. Despite this factor, both sets of metrics have one major aspect in common.

This similarity is the fact that both sets of metrics can be used to predict the complexity of a system in the future. Thus, the results of these metrics do not show the actual complexity of a system, rather they show a prediction of the complexity at a particular point in time.

2.10. How this Overview Relates to the Thesis

As can be seen in the preceding overview there are a lot of different problems related to measuring the level of maintenance in an information system. One of the aims of this thesis is to find out if an object-oriented system is more maintainable than an equivalent system built with a structured approach.

By determining which approach leads to more a maintainable system it is hoped that this thesis will help to clarify some of the problems involved in measuring the level of complexity. One such problem which needs to be clarified is how to make meaningful measurements that can easily be compared no matter what approach to systems development is used. This is one area that needs further investigation as current maintainability metrics are predominantly designed to measure either structured or object-oriented systems but not usually both.

2.11. Summary

This chapter has covered a lot of material concerned with maintenance and the measurement of maintenance in terms of the level of complexity in a system.

The chapter started by defining and distinguishing between maintenance and maintainability. Then, an overview of maintenance and the types of maintenance that exist was given. The systems maintenance lifecycle was also defined. There was also a discussion on the ways of measuring maintenance using various metrics. In particular, there was an in depth discussion on the MOOSE metric set for object-oriented systems as well as McCabe's *Cyclomatic complexity* metric for structured systems. Finally, this chapter finished by discussing how this overview of maintenance and the methods of measuring maintenance related to the thesis.

CHAPTER 3: THE CASE STUDY

3.1. Introduction

The purpose of this chapter is to give an overview of the case study used in this study as well as the changes that were made to the case study. The case study used is taken from 57.221 Information Systems Analysis Project Guidelines and Scenario booklet (Jackson, 1996). For the purposes of this study the scope of the case study only includes the order processing system. Therefore, only an outline of the order processing system will be given.

3.2. The Trusty Furniture Company Case Study

The case study is based around a fictional furniture company in Palmerston North called the Trusty Furniture Company Limited. The Trusty Furniture Company is a small private company located in the lower part of the North Island that was founded in 1946 (Jackson, 1996). This furniture company specialises in " 'reproduction' dining and living room furniture" (Jackson, 1996, p.9).

The Trusty Furniture Company supplies approximately 100 furniture stores as well as providing a mail order service. The mail order service uses a catalogue that currently contains 70 different products.

In terms of distribution, the Trusty Furniture Company divides "New Zealand into 12 areas of roughly equal population" (Jackson, 1996). The purpose of this division is to enable the orders for each region to be delivered in a full van, so that the Trusty Furniture Company can make economical deliveries to their customers.

"When the orders are received they are first stamped with a received date.... The orders are then passed to Accounts for entry on the computer.... After being entered they are then returned to Sales were they are filed in folders corresponding to the 12 distribution areas" (Jackson, 1996, p.10).

"After the original order is fed into the computer, a printout produces a six-part document set consisting of:

- Acknowledgement of Order (sent to customer)
- Salesperson's Copy (sent to salesperson via Sales)
- Office Copy (kept in Accounts)
- Despatch Instruction (sent to the Despatch Manager Initially)
- Despatch Note (sent to the Despatch Manager Initially)
- Van Copy (sent to the Despatch Manager Initially)" (Jackson, 1996, p.10).

"At the time of despatch the items actually loaded are recorded on the despatch instruction which copies through onto the Despatch Note and Van Copy.... The Van Copy goes with the Driver to obtain the customer's acknowledgement of receipt....After an order is loaded, the Despatch Note is sent to Sales who forward it to the customer, noting any shortages from the original customer order on the original order.... Daily and weekly summaries are available of orders received, analysed by product" (Jackson, 1996, p.11).

The main activity that the Sales Director has in the order processing system is to produce a delivery plan which show the areas to be delivered to and the orders that need to be loaded on each van (Jackson, 1996). These delivery plans are "usually issued on Friday Nth for the week Monday (N + 9)th/Friday (N + 14)th" (Jackson, 1996, p.22).

The Orders Clerk is responsible for handling customer requests. "New customers generally telephone the company seeking information.... However, a substantial number do write.... A brochure an copies of the order form are then posted to the prospective customer" (Jackson, 1996, p.30).

Orders are generally received on the standard order form, however, existing customers may place an order via telephone. Currently, new customers are not allowed to place an order via telephone.

Once the order has been placed, the Orders Clerk checks the order. "If there are format problems the Orders Clerk either makes the necessary adjustment or contacts the customer by telephone and occasionally by fax.... The customer is always contacted if the furniture selected is no longer offered or if there has been a price change" (Jackson, 1996, p.30).

"At the end of the day the Orders Clerk 'posts' all the order forms to the Accounts Department.... The following day computer generated copies of the forms are returned" (Jackson, 1996, p.30).

Those customers who pay on credit and have a bad credit history must receive approval from the Finance Director. The Finance Director checks the customer's credit rating and then makes a decision as to whether the order should be accepted or rejected. Any orders which do not meet the Company's criteria are declined. Cash-On-Delivery is not accepted due the problems which it can cause.

3.3. Description of the Changes Made to the Case Study

A number of changes were made to the basic Trusty Furniture Company case study in order to facilitate the measurement of these changes. The changes to the basic case study revolved around the introduction of letting customers pay for their goods in advance, rather than paying for the goods when the invoice arrived.

Thus, any changes to the basic Trusty Furniture Company Order Processing system need to take into account that some customers may prefer to pay for their goods at the time of purchase rather than buying the goods on credit through the Trusty Furniture Company. As a result of these changes, the Trusty Furniture Company's Order Processing system needed to incorporate facilities for the payment of goods at the time of purchase. Such facilities include processing for payment by, cash, cheque, credit card, or money order.

Unlike, the basic case study, the modified version of the Trusty Furniture Company's Order Processing system needs to incorporate the payment of the order in the order processing system rather than in the accounts system. Despite this factor, the accounts system will continue to be used for billing those customers who have elected to pay for their goods after they have been received.

CHAPTER 4: STRUCTURED ANALYSIS DEFINITIONS AND TECHNIQUES

4.1. Introduction

While there are several structured analysis and design methodologies around, there are not many variations in the definitions of the main concepts. The structured development methodologies have been in existence since the 1970's when it was identified that there was a need for a more structured way of developing information systems (Fichman and Kemerer, 1992). Further to this point, "the structured methodologies were developed to promote more effective analysis and more stable and maintainable designs" (Fichman and Kemerer, 1992, p.24).

Some examples of structured development methodologies are Gane and Sarson (1977), DeMarco (1978), McMenamin and Palmer (1984), and Yourdon (1989).

This chapter will investigate the advantages and disadvantages of the structured methodologies as well as providing definitions for the essential components of the structured methodology. Finally, this chapter will conclude by giving an overview of Yourdon's (1989) modern structured methodology.

4.2. Advantages of Structured Methods

There are a number of advantages to using a structured methodology in the design of a information system. One of the big advantages of the structured method is that "much of the necessary documentation is an integral part of the systems development process" (Bingham and Davies, 1992, p.11) in comparison to earlier methods of system development. Because the documentation is an integral part of the development it means that the documentation is kept up-to-date more rigorously (Bingham and Davies, 1992). Thus, the up-to-date documentation makes it easier to maintain the system at a later stage in its life.

Another advantage of using a structured methodology is that it "provides an effective way of partitioning work during the analysis phase" (DeMarco, 1979, p.35). This means that "once the top-level decomposition of function has been worked out, the whole is divided into separable pieces.... Work can then be allocated on the same basis, one analyst or team to each piece" (DeMarco, 1979, p.35). Thus, while proportionately more work is done in the analysis phase, the total elapsed time need not be increased (DeMarco, 1979). Therefore, the partitioning of effort allows some stages of the systems development to be carried out in parallel, thus reducing the amount of time needed to complete the system.

A further advantage of using a structured methodology is that it "makes a clear distinction between logical analysis and physical design" (Mason and Willcocks, 1994, p.200). Thus, the structured approach attempts to model a system during the analysis phase without any of the physical constraints, such as the type of computer, which are used in the system. Instead, these constraints are dealt with during the design phase. This distinction is useful during maintenance as it can show whether any of the physical constraints which have been imposed on the system are limiting the efficiency and effectiveness of the system.

According to Gane and Sarson (1979), one of the main benefits of using a structured methodology is that they clearly show what the systems developers are building. Using this process the users can check that the right system is being built (Gane and Sarson, 1979). This is important as it is the users who must used the finished system. Thus, by giving the users some input into the development process, they are more likely to get the system that they asked for.

4.3. Disadvantages of Structured Methods

While there are obviously a lot of benefits to be gained from using a structured approach to information systems development, such as those examples given previously, there are some disadvantages to using a structured approach.

One such disadvantage is highlighted by Bingham and Davies (1992) who say that structured approaches tend to have problems "portraying temporal relationships" (p.13). An example of such a temporal relationship is where an exception to the normal processing in a system occurs.

Because structured methodologies were originally designed to handle large batch processing applications which do not contain many temporary relationships, they have trouble showing the temporary nature of some modern applications such as Geographical Information Systems. Thus, because some relationships only exist for a temporary amount of time, structured methodologies are unable to model these temporal relationships effectively.

The second disadvantage of using a structured approach that is put forward by Bingham and Davies (1992) is that the structured approach has problems "indicating the relative importance of major and minor processes which are often given the same apparent emphasis" (p.13). An example of this would be in an order processing system where two processes, *Process Payment* and *Print Packing List* would be given the same level of emphasis even though the *Process Payment* process may be more important than the *Print Packing List* process. Such a disadvantage becomes evident in a critical application such as a guidance system because it is very likely that some processes will be more important in the case of a failure than other processes. Thus, there is a need to show some order of precedence in order to overcome this problem.

As can be seen from the disadvantages given above, it is obvious that most of the problems that have occurred with the structured approach are a result of the structured methodology being applied to applications that it was not originally designed to handle. Thus, modern applications have outgrown the structured approach and are being hindered rather than helped by the methodology. Further to the previous point, as the applications get more complex, the problems with the structured approach will only intensify.

4.4. Structured Analysis Definitions

The purpose of this section is to give an overview of some of the key concepts in structured analysis.

4.4.1 Processes.

The first concept to be defined will be that of the process. The process is one of the components of a common structured development diagram known as a dataflow diagram (DFD). Processes "represent the various individual functions that the system carries out.... Functions transform inputs into outputs" (Yourdon, 1989, p.67). Yourdon (1989) also calls processes "bubbles" (p.67).

Further to this, some of the other structured development methodologies that exist use different names to describe the process. One such example of this is Teague and Pidgeon (1985) who call the process a transform. A transform is defined by Teague and Pidgeon (1985) as "a process that changes incoming dataflows into outgoing dataflows" (p.77). Despite the differences in name, the definitions for processes and transforms are essentially the same. This similarity suggests that while the authors may not agree on the name of the process, transform or bubble, they do agree on the general function of this particular component of the dataflow diagram.

4.4 Dataflows.

The next concept to be defined will be that of the dataflow. A dataflow is also part of the dataflow diagram. According to Yourdon (1989) dataflows "are the connections between the processes (system functions), and they represent information that the processes require as input and/or the information they generate as output" (p.68). In comparison, DeMarco (1979) defines a dataflow as a "pipeline through which packets of information of known composition flow" (p.52). DeMarco's (1979) definition is an attempt "to define dataflow in a way that distinguishes between the interface and the information that passes over the interface" (p.52) whereas Yourdon's (1989) definition does not make this distinction, rather Yourdon (1989) views the dataflow as a means of connecting processes together.

4.4.3 Datastores.

Datastores are the next concept which will be defined. Datastores are also a component of the dataflow diagram. Datastores "show collections (aggregates) of data that the system must remember for a period of time.... When the systems designers and programmers finish building the system, the stores will typically exist as files or databases" (Yourdon, 1989, p.68). DeMarco (1979) calls this component of the dataflow diagram a file. According to DeMarco (1979), "a file is a temporary repository of data.... It may be a tape, or an area of a disk, or an index file in someone's drawer.... It might even be a wastebasket" (p.57).

4.4.4 Terminators.

The final concept that relates to dataflow diagrams is the terminator or external entity. Yourdon (1989) states that the "terminators show external entities with which the system communicates.... Terminators are typically individuals, groups of people (e.g. another department or division within the organisation), external computer systems, and external organisations" (p.68). The terminator is sometimes called a source or a sink by some authors such as DeMarco (1979). DeMarco (1979) states that "a source or sink is a person or organisation lying outside the context of a system, that is a net originator or receiver of system data" (p.59). Thus, it appears as though there is very little difference between the definitions put forward by Yourdon (1989) and DeMarco (1979) as both definitions emphasise the idea of an entity outside the system.

Teague and Pidgeon (1985) separate external entities into origins and destinations. An origin is "a person or organisation or system outside the system that provides information to the system in the form of an incoming dataflow" (Teague and Pidgeon, 1985, p.77). A destination is defined as "a person or organisation or system outside the system that receives a system output" (Teague and Pidgeon, 1985, p.77). Further to the previous definitions, Teague and Pidgeon (1985) state that "origins and destinations are sometimes collectively called external entities" (p.77). Teague and Pidgeon's (1985) view is unique compared to the definitions provided by Yourdon (1989) and DeMarco (1979) as it makes a distinction between the different types of entities that exist outside the system.

4.4.5 Entities.

The next set of definitions relates to components of the entity-relationship diagram. The first of these components is the object type or entity. Yourdon (1989) says that "an object type represents a collection, or set, or objects (things) in the real world whose members play a role in the system being developed, can be identified uniquely, and can be described by one or more facts (attributes)" (p.70). Mason and Willcocks (1994) on the other hand define an entity as "something which exists independently and has a number of features called attributes, which can singly or in combination uniquely identify one particular instance of that entity" (p.258). Both of these definitions have a lot of similarities. These similarities are highlighted by the inclusion of key ideas such as uniquely identifying each entity through the use of particular attributes. However, these two definitions also differ in some respects.

One such area in which there is a difference is that Yourdon's (1989) definition mentions that an entity is a representation of something in the real world, whereas Mason and Willcocks (1994) completely omit this factor from their definition, preferring instead to state that an entity is just something that "exists independently" (p.258). This omission from Mason and Willcocks (1994) definition may cause problems in determining the entities in a system as their definition suggests that as long as the proposed entity exists independently it does not matter if the entity exists in the real world. Such a definition could lead to a situation where some entities in the system did not exist in the real world, thus making it harder to determine which of the proposed entities were valid.

4.4.6 Relationships.

A relationship is the second basic component in an entity-relationship diagram. "A relationship represents a set of connections, or associations, between the object types" (Yourdon, 1989, p.70). In comparison, Mason and Willcocks (1994) provide a much more simplistic definition of a relationship by saying that it is "a link between two or more of the entities" (p.259). The main similarity between these definitions is the emphasis on the idea of a link or a connection between the entities.

A further definition of a relationship is given by Fertuck (1995) who says that a relationship is "a representation of some interaction in the real world between corresponding entity types" (p.641).

The main difference between Fertuck's (1995) definition and the previous definitions given here is the idea of "interaction in the real world" (p.641). The previous definitions of a relationship given by Yourdon (1989) and Mason and Willcocks (1994) do not specifically mention where the interaction between the entities occur. Instead, Yourdon (1989) and Mason and Willcocks (1994) concentrate on the idea of a connection or a link between the respective entities.

4.5. Overview of Yourdon's (1989) Modern Structured Methodology

One of the first steps in Yourdon's (1989) methodology, after the initial investigation into the feasibility of the proposed system, is the construction of an essential model. "The essential model is a model of what the system must do in order to satisfy the user's requirements, with as little as possible (and ideally nothing) said about how the system will be implemented" (Yourdon ,1989, p.323). Thus, "our system model assumes that we have perfect technology available and that it can be obtained at zero cost" (Yourdon, 1989, p.323).

The essential model itself consists of two components, the environmental model and the behavioural model. Each of these models are used to define and describe different aspects of the system which is to be developed.

According to Yourdon (1989) the environmental model is used to "define the boundary between the system and the rest of the world" (p.326). The environmental model consists of a "context diagram, an event list, and a short description of the purpose of the system" (Yourdon, 1989, p.326). In comparison, the behavioural model "describes the required behaviour of the insides of the system necessary to interact successfully with the environment" (Yourdon, 1989, p.326). The behavioural model consists of a number of different components. Among these components are "dataflow diagrams, entity-relationship diagrams, state-transition diagrams, data dictionary entries, and process specifications" (Yourdon, 1989, p.326). Only those diagrams which will be used in this study will be described. The first of those diagrams is the dataflow diagram.

4.5.1 The dataflow diagram.

The dataflow diagram is used to show a "function oriented view of the system" (Yourdon, 1989, p.140). This means that the dataflow diagram shows how the different processes in the system are accomplished. In Yourdon's (1989) Modern Structured Analysis there are a number of different components which make up the dataflow diagram. The definitions for these components were given earlier. The purpose of this section is to illustrate the notation used by Yourdon (1989) in the construction of the dataflow diagram.

4.5.1.1 Processes.

The first component is that of the process. An example of a process is given below in figure 2. It must be noted that both Yourdon's (1989) and DeMarco's (1979) notations are given here due to the limitations of the CASE tool used in this study.

Yourdon *DeMarco*

Figure 2: Example of a Process

In order to be able to first construct the dataflow diagram it is necessary to identify the processes contained within the dataflow diagram. Yourdon (1989) identifies processes by looking for a function that is being carried out by a person, a group, or a mechanism which is involved in the transformation of data flows. In the case of the function being carried out by a person, Yourdon (1989) suggests that the role of the person be identified "rather than the person's name or identity" (p.157). Thus, in general the name of the process describes "what the process does" (Yourdon, 1989, p.143). A good name will usually "consist of a verb-object phrase such as *Validate Input* or *Compute Tax Rate*" (Yourdon, 1989, p.143).

4.5.1.2 Terminators.

The next component of the dataflow diagram is that of the terminator. As can be seen below in the example, the terminator or external entity is a square with the name of the terminator contained within the square.

Figure 3: Example of an External Entity

Finding and naming terminators is generally considered to be a simple activity. In some cases "the terminator is the user" (Yourdon, 1989, p.155). An example of such a terminator would be *Customer*. As with the naming of a process, if the name of a terminator is that of a user or some other person, then it is best to name the terminator with that persons role rather than their name. In other cases, the terminator may be a department or some other external organisation that the system must communicate with in order to function properly. An example of this point would be a terminator such as *Accounts*.

4.5.1.3 Datastores.

The datastore is yet another component which is essential in the construction of a dataflow diagram. Without this component it would not be possible to retrieve or store data that the system needs to remember. An example of how a datastore is drawn in Yourdon's (1989) methodology is given below:

Figure 4: Example of a Datastore

Identifying the datastores in a system is one of the easier tasks in the initial construction of a dataflow diagram. "Typically, the name chosen to identify the store is the plural of the name of the packets that are carried by dataflows into and out of the store" (Yourdon, 1989, p.149). Thus, the dataflow O*rder* is likely to carry packets of data to and from a datastore called O*rders*.

4.5.1.4 Dataflows.

The final component of the dataflow diagram which will be presented here is that of the dataflow. The example given below shows that the dataflow is basically a line which is used to connect the other components of the dataflow diagram to one another.

Figure 5: Example of a Dataflow

Since a dataflow carries specific packets of data, this is usually a good indication of the name of a dataflow. However, the naming process prescribed by Yourdon (1989) is considerably more comprehensive, and thus Yourdon's (1989) method of finding and naming dataflows will be used throughout the development of the dataflow diagram. According to Yourdon (1989), "the name represents the meaning of the packet that moves along the flow.... A corollary of this is that the flow carries only one type of packet, as indicated by the flow name" (p.144). An exception to the rule of a dataflow carrying only one type of packet is the composite dataflow. A composite dataflow is where a group of different dataflows which have something in common are consolidated into a single flow (Yourdon, 1989). An example of such a composite dataflow would be a dataflow called *Customer Details* which may consist of *Customer Name*, *Customer Address*, and *Customer Phone Number*. It must also be noted that the notation for a composite dataflow differs slightly from that of a normal dataflow. The difference is that the composite dataflow has a hollow arrowhead rather than the filled-in arrowhead that is normally used to show a dataflow. An example of the notation used for a composite dataflow is shown below in figure 6:

Figure 6: Example of a Composite Dataflow

4.5.2 The entity relationship diagram.

The second major part of the behavioural model that will be used in this study is that of the entity-relationship diagram. In order to be able to construct this diagram, it is necessary to identify the entities and the relationships between these entities within the system under construction.

4.5.2.1 Entities.

There are several important points which must be taken into consideration when identifying and naming entities. The first such point is that each entity "plays a necessary role in the system we are building" (Yourdon, 1989, p.236). Thus, for the entity to be legitimate, the system would not be able to operate without access to the entity's members (Yourdon, 1989). An example of this point would be in an order processing system where the system needs customer information in order to operate.

In terms of identifying entities, many of the entities will be a "representation of a material thing in the world" (Yourdon, 1989, p.236) such as *Customer*, *Product*, and *Order Form*. This does not mean that all entities are material things, in fact some entities "may also be something nonmaterial" (Yourdon, 1989, p.236) such as a *Delivery Plan*, or an *Order*.

Finally, the last point about entities is concerned with how an entity should be named. While there is no general rule, Yourdon (1989) suggests that a useful convention in naming entities is to use a "singular form of noun" (p.237) such as, employee, customer, or order. An example of the notation used to symbolise an entity is shown below in figure 7.

Figure 7: Example of an Entity

4.5.2.2 Relationships.

The entity-relationship diagram also consists of another major component, the relationship. The relationship is used to show the link between the entities, and should therefore be named in such a way as to represent this link. Thus, the relationship between a customer and a product could be *purchases*, that is, a customer purchases a product. It is important to note that "the relationship can be described from the perspective of *any* of the participating object types, and *all* such perspectives are valid.... Indeed, it is the set of all such viewpoints that completely describes the relationship" (Yourdon, 1989, p.239). Thus, the relationship, product *is purchased by* customer, is also valid. An example of the notations used to symbolise various types of relationships is shown below in figure 8.

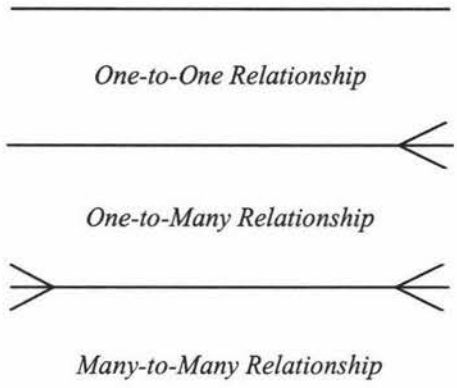


Figure 8: Example of the Notations Used to Represent a Relationship in this Study[1]

4.6. Summary

This chapter has presented an overview of the structured approach to systems development. One of the discussions in this chapter looked at the advantages and disadvantages of the structured approach. Following this discussion, some of the essential concepts such as dataflows and entities were defined. Finally, an overview was given of Yourdon's (1989) Modern Structured Analysis methodology. The overview of Yourdon's methodology discussed how the different components of the dataflow diagram and the entity relationship diagram were found and what each of the components looked like.

[1] Due to limitations of Oracle CASE it is not possible to use Yourdon's (1989) notation for entity relationship diagrams, thus the default notation provided by the CASE tool is used instead.

CHAPTER 5: THE CREATION OF A STRUCTURED MODEL

5.1. Introduction

The structured version of the Trusty Furniture Company's order processing system was created using Yourdon's (1989) modern structured analysis methodology. Originally, it was decided to start on the structured version of the order processing system before the object-oriented version. This decision was carried out to a certain extent as the analysis and design of the system were done on paper originally. However, due to several external factors the system was not able to be completely entered into the CASE tool, Oracle CASE, until after the object-oriented model had been completed.

5.2. Statement of Purpose

The first step in the development of the structured model was to create a statement of purpose (see appendix A). The reason that this document was done first was in order to determine what parts of the Trusty Furniture Company would be modelled. It was decided that for the purposes of this study only the order processing system should be modelled. The main reason for limiting the scope of the system to one particular area was to enable the construction of a system that would not take a large amount of time to complete and would also be suitable for making modifications to at a later stage.

5.3. Event List

After the statement of purpose was completed an event list was created for the Trusty Furniture Company's Order Processing System (See appendix A). The event list is a document which is used to show a list "of the events in the environment to which the system must respond" (Yourdon, 1989, p.351). An example of one such event would be *Customer places an order*. Once a customer places an order the system must respond by processing that order. In order to find the events for the event list it was necessary to find what events in the order processing system caused the system to respond to these events. The process of finding these events was done by answering a series of questions such as *Is this event occurring outside the system?*, *How does the system respond to this external event?*, and *Does this event occur within the defined specifications of this Order Processing System?*.

5.4. Development of an Entity Relationship Diagram

At this point, the foundation for the rest of the structured development process had been created. The next step was to begin constructing an Entity Relationship Diagram (ERD). In order to construct the Entity Relationship Diagram it was necessary to identify the entities that were involved in the Trusty Furniture Company's order processing system.

5.4.1 Identifying the entities.

The process of identifying the entities was accomplished by looking for nouns in the case study. The previous step gave a list of possible entities. The list of entities was reduced through a process of elimination by looking for those entities which the Trusty Furniture Company's order processing system had to remember certain types of information about. Once the list of possible entities had been reduced it was necessary to check each of the remaining entities to ensure that they were needed in the order processing system.

5.4.2 Identifying the relationships.

The process of determining the relationships between each of the entities was the next step to be done in the development of an entity relationship diagram for the Trusty Furniture Company's order processing system. It was necessary to go through each entity and find out which of the other entities it interacted with. As each relationship was found the relationships were named in such a way as to indicate how the entities were related to each other. An example would be the relationship between the entities *Customer* and *Order*. In this case many *Customers* place one *Order*, and one *Order* is placed by many *Customers*.

5.4.3 Identifying the attributes.

At this point the entity relationship diagram was ready to be entered into the CASE tool, Oracle CASE. This stage also involved the addition of attributes for each of the entities in the entity relationship diagram. The attributes for each entity contain information that the system needs to remember, such as *Customer Name*, *Customer Address*, and *Product Number*. The attributes were found by looking at the case study and working out what information needs to be stored by each entity in the order processing system.

5.5. Development of a Context Diagram

The next step is to produce the context diagram which is a dataflow diagram (DFD) that is used to give an overview of the system and its interactions with entities that exist outside of the system. The construction of the context diagram was begun at the conclusion of the entity relationship diagram.

5.5.1 Identifying the external entities.

The first step to be undertaken was to identify the external entities that the Trusty Furniture Company's order processing interacts with. The most obvious of these external entities is that of *Customer*. It was found that the rest of the external entities in this particular case were mainly other departments in the Trusty Furniture Company such as *Despatch* and *Accounts*.

5.5.2 Identifying the dataflows.

At the conclusion of the previous step, work was begun on finding the movement of data within the order processing system. The dataflows within this particular system were found by looking through the case study and finding what inputs and outputs were needed in order for the system to operate successfully. An illustration of this point is shown between the entity *Finance Director* and the *X.Trusty Order Processing System* process on the context diagram. In this illustration, the *Finance Director* receives input in the form of the dataflow *credit status*, and returns the output dataflow *order status details* which is used to show if the Finance Director has/has not approved an order that is being paid on credit. A process similar to the above, was repeated until all of the dataflows were found.

5.6. Development of the Event Response Diagrams

Now that the context diagram was complete, it was tested to ensure that it was consistent with the case study and the statement of purpose. Once this step had been completed the event response diagrams were created using the event list as a guide to each of the events that had to be created. An event response diagram is nothing more than a dataflow diagram that shows the specific processes that occur within each event identified on the event list. At this stage, all of the external entities had been identified as well as some of the dataflows. Thus, it was only necessary to identify which events used which external entities instead of having to go through the process of identifying the entities again.

5.6.1 Identifying datastores.

The only major task that had to be completed with the event response diagrams was to identify the datastores that were to be used in the Trusty Furniture Company's order processing system. This particular task was made easier by the fact that the entities on the entity relationship diagram usually correspond with some datastore on the relevant dataflow diagram. Thus, the entity *Customer* on the entity relationship diagram becomes a datastore on the dataflow diagram called *Customers*. All that was necessary at this point was to identify which events used each of the different datastores. This identification process was done by looking at what data needed to be retrieved and stored in each case. Thus, in the event *Customer Places Order*, it is obvious that the order processing system needs to store data about both the customer and their order. As a result of the previous conclusion, the datastores *Customers* and *Orders* are used in this particular event.

5.6.2 Creation of an unlevelled dataflow diagram.

Once the necessary dataflows were added, the event response diagrams were complete. The completed event response diagrams were then combined together in order to create an unlevelled dataflow diagram. In order to continue it is necessary to level the dataflow diagram. What this means is that the dataflow diagram is decomposed in order to show greater and greater levels of detail. For example, the level 0 dataflow diagram decomposes the context diagram into the main processes that occur within the system, in this case *Validate Order* and *Process Order*.

5.7. Levelling the Dataflow Diagram

The process of levelling the dataflow diagram in this case was done by grouping the processes on the unlevelled dataflow diagram into sets that contained similar processes. Thus, those events that dealt with validating the customer's order were placed in the *Validate Order* group and the events that dealt with the processing of an order were placed in the *Process Order* group. Each of these groups then becomes the bottom level of the dataflow diagram for each of the respective processes. The bottom level of the dataflow diagram is used to show the most detail of all the levels in the dataflow diagram. Thus, the bottom level of the dataflow diagram is the point at which the processes can not be meaningfully decomposed any further.

5.7.1 Level 0.

Once the bottom levels of the dataflow diagram were complete it was necessary to create a level to act as an interface between the context diagram and the bottom level of the dataflow diagram. This level is called level 0 and it represents "the highest-level view of the major functions within the system, as well as the major interfaces between those functions" (Yourdon, 1989, p.166). As was mentioned earlier, level 0 in the Trusty Furniture Company's order processing systems comprises of two processes, *Validate Orders* and *Process Orders*.

On the level 0 dataflow diagram it was decided to only include those datastores that were essential to the operation of the system. What this means is that some of the less critical datastores are hidden in the lower levels of the dataflow diagram. In this particular case the critical datastores are *Orders* and *Customers*, therefore they were included on the level 0 diagram.

5.7.2 Checking the dataflow diagrams.

Once all of the necessary dataflows had been added to level 0, all of the dataflow diagrams were checked to ensure that they were consistent with each other. This consistency is especially important in terms of the same input and output dataflows occurring on each level in the system. Thus, if the dataflow *Order* appears as an input to the system from the external entity *Customer* on the context diagram, then it must continue to appear as an input on both the level 0 diagram and on the bottom level of *Process Orders*.

Apart from the above consistency check, several other miscellaneous checks were done on the dataflow diagram to ensure that it was an accurate representation of the Trusty Furniture Company's order processing system. These checks included checking the dataflow diagram against the statement of purpose and the event list, as well as going through the case study again to make sure that no important details had been left out.

5.8. Implementing Changes to the System

In order to make the changes to the Trusty Furniture Company's order processing system it was necessary to amend the statement of purpose and the event list in order to reflect the changes that needed to be made.

5.8.1 Creating a new event.

In the case of the event list this involved the inclusion of a new event, *Customer Pays for Order at Time of Purchase*. Once these initial changes had been made, work was begun on creating a new event response diagram to reflect the new event that had been added in the previous step.

5.8.2 Modifying the entity relationship diagram.

The creation of the new event *Process Payment* followed the same steps described earlier in this chapter in relation to the creation of the event response diagrams. It was found that a new datastore *payments* was needed. This discovery lead to the inclusion of a new entity on the entity relationship diagram called *Payment*. After studying the entity relationship diagram it was decided that the new entity had relationships with the entities *Customer* and *Order*. Thus, the necessary relationships were placed on the diagram in order to show this link.

5.8.3 Modifying the dataflow diagram.

The next stage in the completion of the changes was to make the necessary changes to the dataflow diagram. The first thing that was done was to include the event response diagram of *Process Payment* on the bottom level of the *Process Orders* dataflow diagram. After this step was done the necessary dataflows from *Process Payment* were levelled upwards to the level 0 dataflow diagram and the context diagram. Thus, the flow of data from a customer who decides to pay at the time of purchase is now documented on all levels of the dataflow diagram. It was decided to bury the datastores that were involved in the processing of payments at the bottom level of the dataflow diagram. As a result of this decision the only dataflows that are concerned with the processing of payments on the higher levels of the dataflow diagram are *Order Payment* and *Payment Receipt.*

5.8.4 Completing the changes.

Once both sets of structured diagrams had been completed, they were then entered into the CASE tool Oracle CASE. As they diagrams were being entered descriptions of important parts of the order processing system were entered into the data dictionary. Amongst these descriptions were descriptions of the attributes that were present on the entity relationship diagram. Selected output from the data dictionary is included in appendix A.

5.9. Application of Metrics to the System

At this point both the original structured model and the changed structured model were complete. It was now time to apply several different metrics to both models in order to measure the changes in complexity that had or had not occurred as a result of the changes to the Trusty Furniture Company's order processing system.

5.9.1 Applying McCabe's cyclomatic complexity metric.

The first metric that was applied was that of McCabe's complexity measure. While this particular metric is primarily designed to be used in the measurement of the complexity of program code, it can also be applied to a dataflow diagram as outlined in McCabe and Schulmeyer (1983). The results for this metric are shown in appendix C.

In order to use McCabe's complexity measure it is first necessary to convert the processes on the bottom level of the dataflow diagram into nodes that can be used on a flowgraph. This process involves listing the processes that occur on the bottom level of the dataflow diagram and labelling them according to which sub-system they belong to. For example, the bottom level of the *Validate Customer Order* sub-system consists of the processes, *Validate Order* and *Determine Order Status*, thus *Validate Order* is labelled V1 and *Determine Order Status* is labelled V2. This process is then repeated with the *Process Customer Order* subsystem, however instead of labelling the processes V1 and V2, the processes are labelled P1, P2, P3, and so on until all the processes are labelled.

The next step in the conversion of the processes into nodes is to place the labelled processes in the order that they logically occur within the system. That is, the nodes are placed in the order that they would be executed if they were modules in a program. Next, each node is linked to the next node in line. For those nodes where a decision in the system occurs, a link is made to reflect the outcomes of such a decision. An example of a decision in the Trusty Furniture Company's order processing system is in the process *Validate Order* (also known as node V1) where the output of this particular process is either that of a Valid Order or an Invalid Order that needs modifications made to it. Thus, this situation is shown on the flowgraph by splitting the nodes output into two separate flows. These nodes with two separate outflows are called decision nodes. It is these decision nodes which are used to calculate McCabe's complexity metric. Thus, once the original and the changed versions of the dataflow diagram had been converted into flowgraphs, the number of decision nodes were counted.

All that was now required was to add 1 (one) to the total number of decision nodes from each flowgraph in order to yield the complexity of each system. In McCabe's (1976) original equation the decision nodes are referred to as the Number of Compares[1]. Hence, *Number of Compares + 1.*

5.9.2 Applying the coupling between objects metric.

The next type of metric to be applied to the structured system was that of the *Coupling Between Objects* (CBO) metric from the MOOSE set of metrics which was used in the measurement of the object-oriented versions of the Trusty Furniture Company's order processing system. *Coupling Between Objects* is the only one of the six MOOSE metrics that can effectively be applied to a structured model.

[1] It must be noted that *Number of Compares + 1* is not the only way of calculating McCabe's Cyclomatic Complexity as there are two other methods described by Sallis et al. (1995, p. 194). Each of these methods yields the same result, hence the simplest equation was chosen for this study.

The reason for this is that the *Coupling Between Objects* metric is concerned with measuring the number of links between objects other than those linked by inheritance. It is because of the fact that this metric does not measure inheritance that it can be used in a structured model. In order to enable some comparison between the object-oriented models and the structured models, it was decided to apply the *Coupling Between Objects* metric to the entity relationship diagram since the entities present had comparable objects on the object-oriented model. Further, the process of creating an entity relationship diagram is similar to that involved in the development of the object-oriented class diagram, thus the entity relationship diagram was a natural choice to be measured using an object-oriented metric.

The *Coupling Between Objects* on the entity relationship diagram was measured by totalling the number of relationships that each entity had. This process was repeated for each entity on both the original and the changed versions of the Trusty Furniture Company's order processing system. The results for each set of measurements were then entered into a spreadsheet in order to enable further calculations. One such calculation was to total the *Coupling Between Object* figures from each entity in order to determine the overall value for each version of the structured model.

5.10. Summary

This chapter has discussed a variety of issues in relation to how the structured models of the Trusty Furniture Company's order processing system were developed and then measured in order to find out the complexity of both versions.

The process of developing the models included developing a statement of purpose and an event list. This was followed by the creation of an entity relationship diagram and a context dataflow diagram which was used to give an overview of the order processing system. The final steps in the development included developing event response diagrams using the event list as a guide, these event response diagrams were then used in the creation of the lower levels of the dataflow diagram. Once the above steps had been completed and the system had been checked for consistency changes were made to the order processing system in order to reflect the addition of a payment facility. In order to complete these changes it was necessary to repeat the previous steps involved in the development of the original system.

The completion of both versions of the order processing system meant that it was possible to check both versions to see if the changes had affected the complexity of the order processing system. These changes were measured using McCabe's *cyclomatic complexity* metric for the dataflow diagram, and a modified version of the *Coupling Between Objects* metric from the MOOSE metrics set was used to measure the complexity of the entity relationship diagram.

CHAPTER 6: OBJECT-ORIENTED DEFINITIONS AND TECHNIQUES

6.1. Introduction

There is a lot of literature written in the object-oriented field which is designed to give the reader a basic overview of many of the basic concepts which are needed in order to be able to understand what is written in the literature. The following section is an overview of the basic object-oriented concepts that are discussed in the literature as well as an attempt to clarify some of the confusion which exists in the literature over the definitions of a lot of the basic object-oriented concepts. This section also tries to answer the common question of *what does object-oriented mean?* Further to the previous point, the reason for using an object-oriented approach to systems development is also discussed as well as the advantages and disadvantages of using the object-oriented approach. Finally, this chapter will give an overview of Booch's (1994) object-oriented methodology.

6.2. What Does Object-Oriented Mean?

This appears to be a common question that many authors have attempted to answer. Because of this there are a lot of different definitions of just what the term object-oriented means. The term object-oriented can mean different things to different people, depending on their background.

An illustration of this point is that a programmer will define the term object-oriented in one way, whereas a database specialist may define object-oriented in a completely different manner. The reason for these differences is attributable to the various applications which object-oriented techniques have been applied to.

Coad & Yourdon (1991) propose an equation to help in the recognition of object-oriented approaches. This equation is as follows:

"Object-Oriented = Classes and Objects

+ Inheritance

+ Communication with messages" (Coad & Yourdon, 1991, p.30).

Communication with messages is a method that Coad & Yourdon (1991) say "is a principle for managing complexity" (p.30).

The main reason for such an equation appears to come from the fact that Coad & Yourdon (1991) have recognised the problem that object-oriented is hard to define because there have been so many different uses of the term 'object'.

The previous statement is supported by Booch (1994), who states that "because the object model derives from so many disparate sources, it has unfortunately be accompanied by a muddle of terminology" (p.35). However, Booch (1994) continues by saying that "what we can agree upon is that the concept of an object is central to anything object-oriented" (p.35).

6.3. Advantages of Object-Oriented Methods

One of the most commonly touted advantages of the object-oriented approach is its "extensive reuse of software objects" (Wilde and Huitt, 1992, p.1038). By reusing objects that have been used before, and therefore thoroughly tested in a real application, the amount of maintenance needed to correct problems in individual objects will diminish. This in turn will result in a decrease in the amount of total maintenance needed by a system during its life.

The previous statement is supported by Booch (1986), who says that one of the major goals in object-oriented software development is to "reduce the total life-cycle software cost by increasing programmer productivity and reducing maintenance costs" (p.244). Despite this reduction in maintenance costs, Booch (1994) says that in order for reuse to be useful, there needs to be a long-term commitment by an organisation if the object-oriented approach is going to be successful. If there is no long-term commitment to the object-oriented approach then the real advantages of this approach will not be apparent.

In addition to the previous point, one of the reasons that the object-oriented approach is supposed to improve the maintenance of object-oriented systems is that "apart from it's elegance, such modular, object-oriented programming yields software products on which modifications and extensions are much easier to perform than with programs structured in a more conventional, procedure-oriented fashion" (Meyer, 1981, p.178).

Thus, because each object only represents one process or behaviour within the system it is easier to modify than a procedure in a traditional system which may consist of one or more functions. Therefore, maintenance is made easier because of "better data encapsulation" (Wilde and Huitt, 1992, p.1038).

However, despite the above advantages of the object-oriented approach, Wilde and Huitt (1992) say that "although maintenance may turn out to be easier for programs written in object-oriented languages, it is unlikely that the maintenance burden will completely disappear" (p.1038). Further to the previous statement, Wilde and Huitt (1992) claim that "maintenance in its widest sense of 'post deployment software support,' is likely to continue to represent a very large fraction of total system costs" (p.1038) this is despite the previous evidence provided by Booch (1986) that the reuse of objects will help to reduce the amount of maintenance needed.

In addition, Rose (1995) notes that "designing for reuse adds to the cost of the particular project where the component is created" (p.18). Thus, the initial cost of designing a system using an object-oriented approach can be seen as being prohibitive due to the initial cost of creating objects that can be reused in later projects. Despite this initial cost, the benefits obtained from reusing an existing object will result in lower costs for projects that are developed at a later stage. Thus, the cost of developing reusable objects will diminish over time as less reusable objects will need to be developed.

6.4. Disadvantages of Object-Oriented Methods

As can be seen above, one of the most obvious disadvantages of the object-oriented approach is its initial cost. The high initial costs of object-oriented development are a result of the need to build classes that can be effectively reused in later development projects.

Related to the high initial cost is the length of time it takes to develop an object-oriented system. This statement is supported by Whiddett et al. (1995) who say that "the object-oriented analysis took considerably longer than the structured analysis, 80 hours compared to 48" (p.113). Thus, it can generally be said that an object-oriented system will initially take longer to develop than an equivalent structured system.

Further to the previous disadvantage, a situation which leads to high costs in an object-oriented system, and therefore a disadvantage, is where an object-oriented approach is being used for the first time. According to Booch (1994), such a situation means that the developers "must start from scratch or at least figure out how to interface their object-oriented applications with existing non-object-oriented ones" (p.289). As well as the previous problem, the developers also need to receive training in order to be able to use the chosen object-oriented approach correctly. Booch (1994) states that "it takes time to develop the proper mindset for object-oriented design" (p.289).

Many of the other major disadvantages with the object-oriented approach are generally concerned with the performance of the system once it has been implemented using an object-oriented language. One such problems is the "performance cost for sending a message from one object to another in an object-oriented programming language" (Booch, 1994, p.288). According to Booch (1994), "in the worst case, a method invocation may take from 1.75 to 2.5 times as long as a simple subprogram call" (p.288) in a procedural language.

6.5. Basic Object-Oriented Concepts

One of the main observations, after studying a lot of literature which covers the basic concepts, is that there appears to be a lot of confusion over the meaning of a lot of the basic concepts which are essential to understanding the object-oriented field.

The main reason for this confusion appears to come from the fact that a lot of the authors have different backgrounds and thus, they have different understandings of what the object-oriented field is about. The previous statement is supported by Korson & McGregor (1990), who recognise this problem and state that "object-oriented remains a term which is interpreted differently by different people" (p.41).

Further to the previous point, Kim (1991), argues that "despite the surging interest in object-oriented programming and object-oriented databases, the programming language, knowledge representation, and database disciplines, and even any one of them by itself, presently do not agree on a single standard for object-oriented concepts" (p.10). He goes on to say that "this may lead one to question the merit of building object-oriented database systems, especially for commercial purposes" (Kim, 1991, p.10). Thus, it is necessary to clarify the meanings of many of the object-oriented concepts.

6.5.1 Objects.

One of the first questions which most of the literature attempts to answer is that of what is an object? Because there has been a great deal of writing on defining what an object is, it comes as no surprise that there is conflict over just what an object really is. According to Shlaer & Mellor (1989), an object is described as "an abstraction of a set of real-world things such that all of the real-world things in the set —the instances—have the same characteristics; and all instances are subject to and conform to the same rules" (p.67).

Zdonik & Maier (1990), provide another view of what an object is. They state that an object is "an abstract machine that defines a protocol through which users of the object may interact" (Zdonik & Maier, 1990, p.7). It must be noted that in this case the definition of an object is in relation to its use in an object-oriented database.

A further definition of an object is given by Coad & Yourdon (1991), who say that an object is "an abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of attribute values and their exclusive services (synonym: an instance)" (p.53). Such a definition gives an extremely broad view of what an object is. However this is probably the most thorough definition of an object.

Another definition is provided by Korson & McGregor (1990) who define an object from two different perspectives. The first perspective is that of an object in a programming language for which they say that "objects are the basic run-time entities in an object-oriented system" (Korson & McGregor, 1990, p.42). The second perspective is from a design viewpoint where the authors state that the "objects model the entities in the application domain" (Korson & McGregor, 1990, p.42).

These different definitions are illustrative of many of the problems that occur in the definition of what an object is. In fact, much of the literature which has been written with the purpose of trying to define the basic concepts of object-orientation, is written from a programming language perspective. Examples of this point include Korson & McGregor (1990), and Nierstrasz (1989).

The definitions given above are just samples of some of the many different definitions that are given to describe what an object is. One of the main features that appears to be contained in the majority of the readings is that an object is used to model real-world things. This is supported by Kim (1991) who says that "any real-world entity is an object" (p.13). In addition to objects modelling real-world entities, they can also be used to model abstract entities such as a chemical formulae (Cattell, 1991).

6.5.2 Classes.

The next concept is that of a class. This is another concept where a wide variety of definitions exist, however, unlike the definitions for what an object is, there appears to be a consistent definition of what a class is. For instance, Zdonik & Maier (1990) state that "every object is an instance of some class" (p.7). They go on to say that "this class defines all the messages to which the object will respond, as well as the way that objects of this class are implemented" (Zdonik & Maier, 1990, p.7).

Another definition of class which can be used as a comparison is provided by Kim (1991), where "all objects which share the same set of attributes and methods may be grouped into a class" (p.13). Further to this, "an object belongs to only one class as an instance of that class" (Kim, 1991, p.13).

A further definition of class is provided by Coad & Yourdon (1991), who define a class as "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" (p.53). This definition of class has the addition of a description which defines how new objects can be created in that class. This feature is not totally unique to Coad & Yourdon (1991) as Atkinson et al. (1992) also define how a class can create new objects, however they give a much more detailed description of how this is done.

Atkinson et al. (1992), define how a class can create new objects in a discussion on the differences between types and classes. They argue that a class "contains two aspects: an object factory and an object warehouse" (p.9). An object factory is the part of the class that "can be used to create new objects" (Atkinson et al., 1992, p.9). An object warehouse on the other hand "allows the class to be attached to its extension, that is, the set of objects that are instances of the class" (Atkinson et al., 1992, p.9).

The previous definitions of a class show that there is indeed an element of similarity in the definitions of what a class is in object-oriented terms. Following on from these definitions, it is therefore possible to provide a standard definition of a class. Thus, a general description of a class could be defined as being a group of similar objects in which each object can only belong to one class.

6.5.3 Types.

Types are another concept which is part of the object-oriented terminology. Zdonik & Maier (1990), say that it is common for people to use the terms type and class interchangeably. However, they state that "when the two terms are used in the same system, type usually refers to specifications, whereas class refers to the extension (i.e. all current instances) of the corresponding type" (Zdonik & Maier, 1990, p.7). This statement tries to differentiate the differences between type and class. While such a statement may be helpful in clarifying the differences, other authors may disagree with the statement made by Zdonik & Maier (1990) that the two terms, class and type, are interchangeable with each other.

One such group of authors is Atkinson et al. (1992), who state that "the notion of class is different from that of type" (p.9). Atkinson et al. (1992) define type as a summarisation of "the common features of a set of objects" (p.9). These authors claim that classes are used for "creating and manipulating objects" (Atkinson et al., 1992, p.9) whereas types are generally used to "check the correctness of programs" (Atkinson et al., 1992, p.9). This comparison of classes and types demonstrates that the two concepts are indeed different even though the specification of a class "is the same as that of a type" (Atkinson et al., 1992, p.9). The previous statement could be a probable source of any confusion that may exist between the class and type terms.

Another definition of type is from Cattell (1994), who states that "types are themselves objects, and may therefore have properties themselves" (p.12). Some of these properties include things such as, behaviour or state.

Because of the above confusion, some authors, such as Cattell (1991), deliberately avoid discussing the term class in favour of discussing types. This is because "the term class has been used with more than one meaning" (Cattell, 1991, p.110).

6.5.4 Abstraction.

Abstraction can be defined as "the principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate more fully on those that are" (Coad & Yourdon, 1991, p.13).

Another perspective on abstraction comes from Booch (1994), who defines an abstraction in the following way: "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer" (Booch, 1994, p.41). The difference between these two definitions of abstraction is that Booch (1994) introduces the aspect of the viewers perspective into the definition. In terms of a system under development, this factor shows that different people will view the system in different ways, depending on their background.

6.5.5 Encapsulation.

According to Atkinson et al. (1992), "the idea of encapsulation comes from (1) the need for a clear distinction between the specification and the implementation of an operation, and (2) the need for modularity" (p.7).

Encapsulation is an object-oriented concept that originated in programming languages and has since found itself adapted for use in object-oriented databases. Hence, two different views of encapsulation exist, the programming language view, and the database view (Atkinson et al., 1992). These differing views could lead to some confusion over what encapsulation is, depending on the viewpoint of the person defining encapsulation.

From a database viewpoint, Cattell (1991), states that "encapsulation provides data independence through the implementation of methods, allowing the private portion of an object to be changed without affecting programs that use that object type" (p.127).

Encapsulation, or information hiding, can be defined from a programming language viewpoint to be "a principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision.... The interface to each module is defined in such a way as to reveal as little as possible about its inner workings" (Coad & Yourdon, 1991, p.14).

As can be seen, both the database and the programming language viewpoints do have some similarities. An example of one of these similarities is that both viewpoints emphasise the idea of hiding data through the use of a private portion.

Both the database viewpoint and the programming languages viewpoint share common attributes in their definitions of encapsulation, such as the idea of hiding information in a private area of the object. However, they also appear to have differences such as the fact that the programming languages viewpoint is more concerned with the structure involved in encapsulation, whereas, the database viewpoint is more concerned about the hiding of the data.

Coad & Yourdon (1991) suggest that the main reason for using "encapsulation is that it helps to minimise rework when developing a new system" (p.14). This rework is reduced by using encapsulation to hide those parts of the analysis that are very volatile (Coad & Yourdon, 1991). The ability to "minimise rework" (Coad & Yourdon, 1991, p.14) would also help to reduce the cost of using an object-oriented methodology in the development of a new system. A reduction in the cost of using object-oriented techniques could help to increase their usage in the development of new systems.

Further to the previous statement, "encapsulation keeps related content together, it minimises traffic between different parts of the work, and it separates certain specified requirements from other parts of the specification which may use those requirements" (Coad & Yourdon, 1991, p.15). What this statement means is that, encapsulation is designed to organise the contents of an object in order to minimise the work required to find out about certain parts of the object.

Encapsulation is a very significant concept in the object-oriented field, so much so that some authors like Korson & McGregor (1990), argue that "encapsulation is such an important part of the object-oriented paradigm that, according to some definitions, a language is not object-oriented unless it provides for encapsulation" (p.59). Although this particular statement is obviously from a programming language viewpoint, it can be said that it applies equally to the database viewpoint.

6.5.6 Inheritance.

Next, the concept of inheritance is discussed. Inheritance "helps to support reuse across systems" (Korson & McGregor, 1990, p.43). Thus, inheritance is another mechanism which can be used to reduce both the amount of time and money spent in the development of new systems.

Cardelli (1990), gives an explanation of what inheritance is by saying that "data is organised in a hierarchy of classes and sub-classes, and data at any level of the hierarchy inherits all the attributes of data higher up in the hierarchy" (p.59). Cardelli (1990) continues this explanation by stating that "the top level of this hierarchy is usually called the class of all objects; every datum is an object and every datum inherits the basic properties of objects, e.g. the ability to tell whether two objects are the same or not" (p.59). This explanation helps to give an overview of what inheritance is.

Another view of inheritance is that provided by Korson & McGregor (1990), who argue that inheritance is "a relation between classes that allows for the definition and implementation of one class to be based on that of existing classes" (p.43). Thus, there is a certain amount of consistency in the definitions of inheritance since both explanations deal with the ability of a class to gain the attributes of another class.

6.5.7 Polymorphism.

Polymorphism is another object-oriented concept which also needs to be discussed. Generally, this is where a piece of code "will work on objects of different forms" (Zdonik & Maier, 1990, p.12).

In comparison to the last definition, Korson & McGregor (1990) say that generally "polymorphism means the ability to take more than one form" (p.45). A more detailed definition is that, "in an object-oriented language, a polymorphic reference is one that can, over time, refer to instances of more than one class" (Korson & McGregor, 1990, p.45).

The previous statements suggest that this concept is primarily a programming language concept. However, this concept has also been applied to object-oriented databases as well.

Zdonik & Maier (1990) call the use of polymorphism in an object-oriented database extension polymorphism. This is where code that is designed to work on objects of type A will also work on objects of type B by "ignoring the fields (e.g. c and d) that are in the extension that the subtype provides" (Zdonik & Maier, 1990, p.12).

6.6. Overview of Booch's Methodology

The object-oriented methodology that was used is that of Booch (1994). Booch (1994) proposes a methodology that appears to be aimed at object-oriented software engineering which seems to concentrate more on object-oriented design than on the object-oriented analysis aspect. Despite this, Booch (1994), does cover some object-oriented analysis.

This section will cover specific aspects of Booch's methodology that are relevant to this study. Thus, the descriptions of several diagrams such as the process diagrams will be omitted since they have not been used in this case, and are therefore irrelevant.

6.6.1 Dimensions of analysis and design.

According to Booch (1994), "we may capture our analysis and design decisions regarding classes and objects and their collaborations according to two dimensions: their logical/physical view, and their static/dynamic view" (p.174). Further to this point, Booch (1994) claims that "both dimensions are necessary to specify the structure and behaviour of an object-oriented system" (p.174). Booch (1994) shows both of these dimensions by using a diagram called *The Models of Object-Oriented Development* (see figure 9 below).

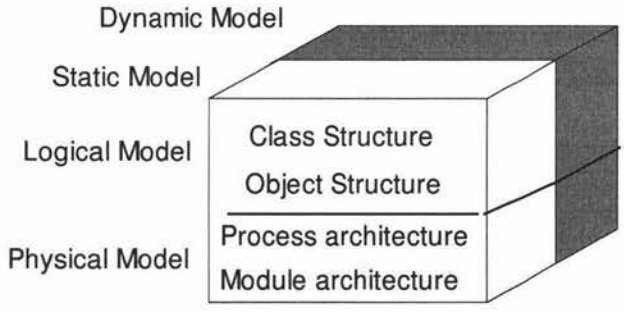


Figure 9: The Models of Object-Oriented Development

(Booch, 1994, p.172).

Booch (1994), states that "for each dimension, we define a number of diagrams that denote a system's models" (p.174). These diagrams help to ensure a level of consistency between the models which are used to model the structure and behaviour of the system (Booch, 1994). Only the basics of the diagrams that Booch (1994) presents will be covered here. The steps that are involved in this methodology will be covered later using a brief overview of the modified version of Booch's (1994) methodology. Before continuing, it is necessary to define what Booch (1994) means by logical and physical models and static and dynamic semantics.

Booch (1994), says that "the logical view of a system serves to describe the existence and meaning of the key abstractions and mechanisms that form the problem space or that define the system's architecture" (p.175). Whereas, "the physical model of a system describes the concrete software and hardware composition of the system's context or implementation" (Booch, 1994, p.175). These definitions lead in to what Booch (1994) considers to be central questions which must be addressed during the analysis of the system. The questions are as follows: "

- What is the desired behaviour of the system?
- What are the roles and responsibilities of the objects that carry out this behaviour?"

(Booch, 1994, p.175).

Further to the above questions, Booch (1994) also argues that "we must address the following central questions relative to the system's architecture:

- What classes exist, and how are those classes related?
- What mechanisms are used to regulate how objects collaborate?
- Where should each class and object be declared?
- To what processor should a process be allocated, and for a given processor, how should its multiple processes be scheduled?"

(p.175).

As can be seen from the above questions, there is indeed a great emphasis on design in Booch's (1994) methodology.

In order to answer the analysis and design questions above, Booch (1994) proposes that the following diagrams are used: "

- Class diagrams
- Object diagrams
- Module diagrams
- Process diagrams"

(p.175).

It must be noted that for this study only the Class diagram and the Object diagrams have been used. The reason that only two of the four possible diagrams have been used is that both the Module diagrams and the Process diagrams are irrelevant in this study as they are specifically concerned with the design process, rather than the overall analysis and design of the system. Further to this point, the process diagram is irrelevant as it is specifically concerned with the hardware implementation of the system, since this study is more concerned with software, the process diagram was left out.

As was mentioned earlier, it is necessary to understand what Booch (1994) means by static and dynamic semantics. According to Booch (1994), the four diagrams which were presented above "are largely static" (p.175). That is, the models do not show the dynamic behaviour of the system, rather, they show the static structures which exist in the system being studied. Thus, these static models do not show the events which "happen dynamically in all software-intensive systems: objects are created and destroyed, objects send messages to one another in an orderly fashion, and in some systems, external events trigger operations upon certain objects" (Booch, 1994, p.175). Booch (1994) goes on to say that "in object-oriented development, we express the dynamic semantics or its implementation through two additional diagrams:

- State transition diagrams
- Interaction diagram"

(p.176). However, it must be noted that in this case only the interaction diagram has been used.

An important note is made in regard to the use of the previous two diagrams by Booch (1994). Booch (1994) states that "each class may have an associated state transition diagram that indicates the event-ordered behaviour of the class's instances.... Similarly, in conjunction with an object diagram representing a scenario, we may provide a script or interaction diagram to show the time or event-ordering of messages as they are evaluated" (p.176).

6.6.2 Class diagrams.

As was mentioned previously, in order to use Booch's (1994) methodology, it is necessary to use several different types of diagrams which are used to help model the system(s) under study. The main diagram which is used in this study is Booch's (1994) class diagram. The class diagram is used to "show the existence of classes and their relationships in the logical view of a system" (Booch, 1994, p.176).

Booch (1994) also uses class diagrams in two different roles, one during analysis, and the other during design. During analysis, the class diagrams are used to "indicate the common roles and responsibilities of the entities that provide the system's behaviour.... During design, we use class diagrams to capture the structure of the classes that form the system's architecture" (Booch, 1994, p.177). However, in this study only the analysis role was covered.

In the class diagram a cloud icon is used to represent a class. Each class contains a name. Further to this "for certain class diagrams, it is useful to expose some of the attributes and operations associated with a class.... We say 'some' because for all but the most trivial class, it is clumsy and indeed unnecessary to show all such members in a diagram" (Booch, 1994, p.178). An operation is defined by Booch (1994) as being "some work that one object performs upon another in order to elicit a reaction" (p.517).

One important rule relating to classes is that "a name is required for each class" (Booch, 1994, p.177). Further to this rule, "every class name must be unique to its enclosing class category" (Booch, 1994, p.177).

A final point which is considered to be important by Booch (1994) is that an abstract class can be shown by placing a letter *A* in a triangle anywhere inside the class icon. An abstract class "is one for which no instances may be created" (Booch, 1994, p.179). An example of what a class looks like is shown in figure 10.

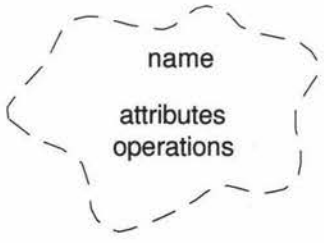


Figure 10: Class Icon

(Booch, 1994, p.177).

In order to link the classes together on the class diagrams, Booch (1994) uses class relationships to model the relationship of one class to another. According to Booch (1994) "the essential connections among classes include association, inheritance, 'has', and 'using' relationships" (p.179). Further to this, "the inheritance icon denotes a generalisation/specialisation relationship" (Booch, 1994, p.180) and the 'has' relationship is used to show aggregation (Booch, 1994).

Although class diagrams have several more advanced features, such as parameterized classes, metaclasses, class utilities, nesting and other features, these are not covered here as this section is only designed to give an overview of the basic features.

6.6.3 Object diagrams.

Object diagrams are the next type of diagram to be discussed. Object diagrams are "used to show the existence of objects and their relationships in the logical design of a system.... Stated another way, an object diagram represents a snapshot in time of an otherwise transitory stream of events over a certain configuration of objects" (Booch, 1994, p.208).

Object diagrams can be used in two different roles in Booch's (1994) methodology. The first role is during analysis when the object diagram is used "to indicate the semantics of primary and secondary scenarios that provide a trace of the system's behaviour" (Booch, 1994, p.208). The second role is during design where the object diagram is used "to illustrate the semantics of mechanisms in the logical design of a system" (Booch, 1994, p.208).

The two main elements which are shown in "an object diagram are objects and their relationships" (Booch, 1994, p.208). Like the classes in a class diagram, an object is shown on an object diagram as a cloud icon. Each object contains both the name of the object and its attributes. Further to this, the name of an object "may be written in any of the three following forms:

- A Object name only
- : C Object class only
- A : C Object name and class"

(Booch, 1994, p.209).

A relationship is shown on an object diagram by a line that shows the messages between objects. According to Booch (1994), "a link may exist between two objects (including class utilities and metaclasses) if and only if there is an association between their corresponding classes" (p.210).

Further to this point, "the existence of an association between two classes therefore denotes a path of communication (that is, a link) between instances of the classes, whereby one object may send messages to another.... All classes implicitly have an association to themselves, and hence it is possible for an object to send a message to itself" (Booch, 1994, p.210).

Another important point about the links is that they are able to show messages. "Each message consists of the following three elements:

- D A synchronisation symbol denoting the direction of the invocation
- M An operation invocation or event dispatch
- S Optionally, a sequence number"

(Booch, 1994, p.211). Continuing on from the previous point, the direction of a message is shown on the diagram by using a directed line (Booch, 1994). Further to these basic features, object diagrams also have the ability to show other features like: roles, keys and constraints, the direction of the flow of data, visibility, active objects and synchronisation, and finally, time budgets.

6.6.4 Interaction diagrams.

Interaction diagrams are the final dynamic diagram as well as the final diagram that is presented by Booch (1994). The interaction diagram is used to "trace the execution of a scenario in the same context as an object diagram.... Indeed, to a large degree, an interaction diagram is simply another way of representing an object diagram" (Booch, 1994, p.217).

Interaction diagrams do not "introduce any new concepts or icons; rather they take the most essential elements of object diagrams and restructure them" (Booch, 1994, p.217).

An example of an interaction diagram is shown below in figure 11. The sequence of events is from the top to the bottom.

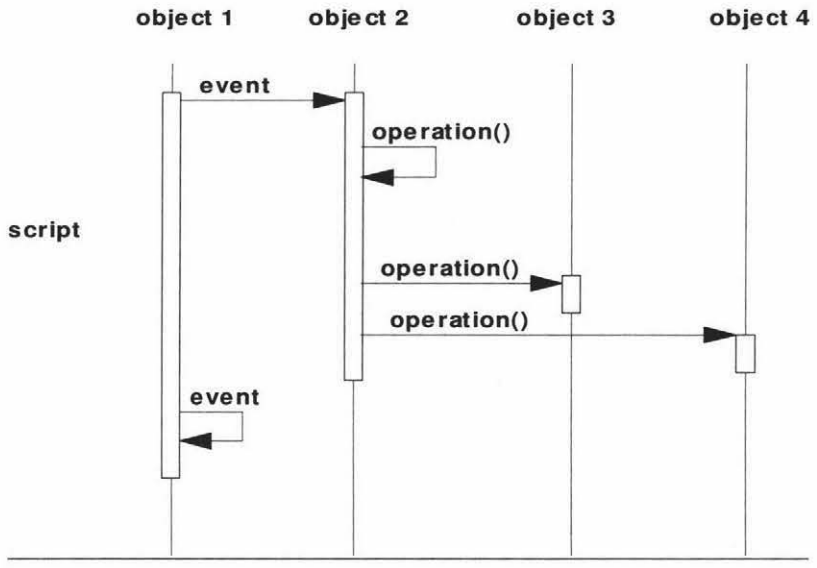


Figure 11: Example of an Interaction Diagram

(Booch, 1994, p.592).

Due to constraints imposed on Booch's (1994) methodology by the CASE tool, Rational Rose, it is necessary to give a brief overview of the steps involved in the modified Booch methodology.

6.7. Booch: The Rational Approach

The modified version of Booch's methodology consists of three major steps. These steps are requirements analysis, domain analysis, and system design. Only the first two steps were performed in this case as system design involves the creation of the module diagrams and the process diagrams.

6.7.1 Requirements analysis.

The requirements analysis "is a high-level stage that identifies the key functions the system is to perform, defines the scope of the domain that the system will support, and documents the key practices and policies of the enterprise that the system must support" (White, 1994, p.6). The requirements analysis stage uses use case analysis as is used in Jacobson's methodology in order to help "describe system functions" (White, 1994, p.7). The deliverables from this stage include the following outputs:

- a system charter, "which outlines the responsibilities of the system" (White, 1994, p.8).
- a system function statement, "which outlines the key use cases of the system" (White, 1994, p.8).

6.7.2 Domain analysis.

The domain analysis on the other hand is more complex than the requirements analysis as it is "the process of defining a precise, concise, and object-oriented model of that part of the real-world enterprise that is relevant to the system (the problem domain)" (White, 1994, p.9). The objective of this part of Booch's methodology is to identify "all major objects in the domain, including all data and major operations that will be needed to carry out the system's function" (White, 1994, p.9).

The deliverables from the domain analysis include: "

- Class diagrams, which identify the key classes, or types, of the domain
- Class specifications, which contain all semantic definitions of the classes, their relationships, their attributes, and their key operations
- Object-scenario diagrams, which illustrate how the objects will interact to carry out key system functions
- Data dictionary, which lists all the domain entities including classes, relationships, and attributes"

(White, 1994, p.10).

In order to produce the above deliverable it is necessary to go through a series of steps. The "steps that are performed during domain analysis include:

- Define classes, which includes identifying the major types of domain objects and defining them
- Define relationships, which includes describing major associations between those objects
- Define operations, which includes identifying the major operations required to support class structure and system functions
- Find attributes, which includes determining the properties that describe the classes
- Define inheritance, which includes finding generalisations and specialisations within similar domain types
- Validate and iterate, which includes the review, testing, and repair of the model and actually occurs throughout the process"

(White, 1994, p.10).

6.8. Summary

This chapter started by discussing what the term object-oriented meant as there is a lot of confusion over its real meaning. It was found that the concept of an object is central to any definition of object-oriented (Booch, 1994).

A discussion on the advantages and disadvantages of object-oriented methods found that reuse was one of the major benefits of the object-oriented methods.

The next section of this chapter defined some of the basic concepts which are used in the object-oriented method. Finally, an overview was given of Booch's (1994) object-oriented methodology and its modified version called the Rational approach. The overview of the methodologies discussed some of the diagrams that are used as well as the basic steps that need to be carried out in the design of an object-oriented system.

CHAPTER 7: THE PROCESS OF CREATING AN OBJECT-ORIENTED MODEL

7.1. Introduction.

The first thing that was done to create an object-oriented model of the Trusty Furniture Company was to create a system function statement that identified the key use-cases in the scenario. A copy of the original system function statement is included in appendix E. As was outlined in the structured model, it was decided to limit the system to only include the order processing system of the Trusty Furniture Company. Once an initial system function statement was completed, the statement was checked against the case-study to ensure that no major use-cases had been omitted.

7.2. Development of the Initial Class Diagram

Throughout the rest of the development process the CASE tool Rational Rose\C++ was used for modelling and documentation purposes. The completed system function statement was then used to develop an initial class diagram.

7.2.1 Identifying the key classes.

The first step in this process is to identify the key classes in each of the use-cases outlined in the system function statement.

This identification process consisted of searching through the system function statement and the case study for nouns such as *Customer* or *Order*. Such a process was effective in locating the majority of key classes in this study.

7.2.2 Identifying the relationships between classes.

Once the key classes have been found, it is necessary to find the relationships between these classes. This step in the development of an initial object-oriented model involves finding out which classes are linked to each other.

7.2.2.1 Identifying inheritance relationships.

The first type of relationships that were found were the inheritance relationships. It is a relatively easy process to find those classes that are linked through an inheritance relationship. In general, those classes that tend to be subtypes such as *Previous Customer* would inherit features from their supertype, *Customer*. That is, *Previous Customer* is a type of *Customer*. While this simple rule works in a large number of cases, there were a few classes which were not so straight forward. The six classes that inherit from the *Order Document Set* class are examples of this point.

Originally, it was thought that the six classes that were generated from the *Order Document Set* were related through a has relationship. An illustration of this point would be an *Order Document Set* has a *Sales Copy*. However, further investigation revealed that the relationship between the six copies of the order and the *Order Document Set* was indeed that of inheritance. The reason for this decision was that each of the six classes inherited the features of the *Order Document Set* as a result of these classes being copies of a valid order. An illustration of the symbol used to show inheritance on the class diagram is shown below.

Figure 12: The Inheritance Relationship

7.2.2.2 Identifying uses relationships.

Those classes that were linked by a uses relationship were found by searching through the classes and looking for possible client/server relationships. That is, a uses relationship involves one class acting as a server which responds to the requests of another class, or client. Thus, "this relationship indicates that the client in some manner depends on the server to provide certain services.... It is typically used to indicate the decision that operations of the client class invoke operations of the server class, or have signatures whose return class or arguments are instances of the server class" (Booch, 1994, p.180).

An example of such a uses relationship is shown between *Order* and *Credit Limit*. In this relationship, the class *Order* is the client as it asks the *Credit Limit* class to check if a specific customer has a good credit rating. Once the *Credit Limit* class has obtained this information, it sends the answer back to the *Order* class for further processing. An illustration of the symbol used to represent a uses relationship is shown below.

Figure 13: The Uses Relationship

7.2.2.3 Identifying has relationships.

The next step which was done in this study was to find those classes that were linked through a has relationship. In order to find these relationships it was necessary to look for those classes that appeared to be related as a result of a whole/part relationship. According to Booch (1994) such a relationship is "also known as aggregation" (p.180). Another way of describing this is to say that, one of the classes contains instances or parts of the other class or aggregate.

An illustration of this point is the relationship between *Order Information Pack*, *Product Brochure*, and *Order Form*. In this relationship *Order Information Pack* is the aggregate or whole, while *Product Brochure* and *Order Form* are the parts or instances. What this relationship is trying to show is that one *Order Information Pack* is composed of one *Product Brochure* and one *Order Form*. That is, the *Product Brochure* and the *Order Form* are parts of the *Order Information Pack*. An illustration of the symbol used to show a has relationship is shown below.

Figure 14: The Has Relationship

7.2.2.4 Identifying association relationships.

Finally, those classes that were left after the previous steps were checked to see if the link between them was an association relationship. An association is a relationship between two classes that does not fall into the previous categories of the inheritance, uses, and has relationships. A more formal definition of an association is provided by Booch (1994) who states that an association "connects two classes and denotes a semantic connection" (p.179). Thus, it is possible to provide names for these relationships. An illustration of one of these names would be the *requests* relationship between the *Customer* class and the *Order Information Pack* class. An illustration of the symbol used to show an association relationship is shown below.

Figure 15: The Association Relationship

7.2.3 Defining classes and relationships.

The next step in the development of an initial class diagram is to provide definitions for the key classes and the relationships which have been identified in the previous steps. This stage of the process required the entry of the appropriate definitions for each class or relationship into Rational Rose's data dictionary.

7.3. Development of the Interaction Diagrams

The construction of the interaction diagrams involves creating a separate diagram for each of the functions in the system function statement.

7.3.1 Identifying classes and objects.

In order to create each of these diagrams it is necessary to determine which classes/objects are used by the system in order to successfully complete a particular function.

7.3.2 Identifying methods.

Once the classes/objects have been determined, the methods for each of the functions are designed. A set of individual methods (or operations) are used by each function in order to complete their intended function. Thus, each method is responsible for carrying out specific tasks within a function.

The different methods in each function are found by looking at each function and then deciding on the steps required to complete a particular function. Thus, it is a matter of going through each function in a procedural manner in order to work out what information and services are needed by the function so that it can operate properly.

7.3.3 Adding scripts to the diagram.

At this point, the basic interaction diagram is nearly complete. The last step in the construction of the interaction diagrams is to add scripts to each of the diagrams. A script is the text which is shown to the left of an interaction diagram. The purpose of the script is to show the logic of what is happening in the function. Scripts are also useful for showing iteration and conditions within the function (Booch, 1994). In general, a script is written in structured english, however Booch (1994) notes that freeform english or the syntax of the chosen implementation language is also acceptable.

7.4. Development of the Object Diagrams

Once the interaction diagrams were completed, work was begun on creating the object diagrams. As a result of using Rational Rose this step in the development of the Trusty Furniture Company's Order Processing System was greatly simplified. the reason for this simplification was due to the fact that Rational Rose is able to use an interaction diagram to automatically create the respective object diagram and vice versa. Thus, once each interaction diagram was completed, it was just a matter of selecting *create object diagram* and letting Rational Rose create the object diagram.

While Rational Rose creates the object diagram, it does not generate the definitions for the objects and their respective relationships. Thus, it is necessary to define each of the objects and their relationships in turn following a procedure similar to that used in defining the classes and their relationships in the class diagram. As each object diagram was generated by Rational Rose they were checked for accuracy and consistency in order to reduce the likelihood of any errors being introduced into the model.

7.5. Why the Module and Process Diagrams Were Not Done

It was decided at this point not to construct the module diagrams or the process diagrams as they were not really needed in this particular system at present. The main reason behind this decision was that neither diagram would significantly enhance the final system.

7.6. Checking the Model for Consistency

Once the above steps were completed, the initial object-oriented model of the Trusty Furniture Company's order processing system was checked to ensure that it was consistent with the system function statement and the original scenario. After the model had been checked the initial object-oriented model was complete.

7.7. Making Changes to the Object-Oriented System

The next stage in the development of an object-oriented model for the Trusty Furniture Company's order processing system was to make a number of changes to the initial model. As in the structured model of the Trusty Furniture Company's order processing system, the changes involved the addition of a payment facility so that customers could pay for their goods at the time of purchase if they chose to do so.

7.7.1 Modifying the system function statement.

To complete the changes it was necessary to modify the system function statement in order to reflect the above changes to the order processing system. This process involved determining what the function of the changes were. In this case it was decided that the new function which had to be added was that of *Customer pays for goods at time of purchase*.

7.7.2 Identifying the new classes and relationships.

The next step involved identifying the classes which were involved in this new function. After an initial investigation it was found that two existing classes were involved in this function. These classes were *Customer* and *Order*. Next, four new classes were added to the order processing system to enable the processing of payments at the time of purchase. The four new classes mentioned above were composed of three sub-classes, *Cash*, *Cheque*, and *Credit Card* which all inherit the characteristics of the abstract class *Payment Method*. The decision to use inheritance in this function was made after it was found that the relationship between *Customer* and the three sub-classes were not suited to a uses or has relationship. A further result of this decision was the inclusion of the abstract class *Payment Method* which is used to clarify the relationship between *Order* and the three sub-classes.

After the classes and their relationships were completed, the process of defining the new classes was begun. As in the initial object-oriented model of the Trusty Furniture Company's order processing system, Rational Rose was also used to define the classes and their relationships in the modified version of the system. This step in the modification of the system involved finding out the roles and functions of each class and relationship. Once the above step was completed, this information was then documented in the system's data dictionary.

7.7.3 Creating a new interaction diagram.

At the conclusion of the previous step, the interaction diagram for the new function was begun. This interaction diagram proved to be more difficult to construct than was first anticipated. The reason for this difficulty was caused by the three sub-classes *Cash*, *Cheque*, and *Credit Card*. Originally the aforementioned sub-classes were included in the interaction diagram, however, after some thought it was determined that it did not matter what type of payment was used by a customer, rather what was important was the fact that the customer wanted to pay at the time of purchase. Hence, the three sub-classes were taken out of the interaction diagram and replaced by *Payment Method* instead. The results of this decision are reflected in the complex decision structure shown in the interaction diagram's script.

7.7.4 Generating a new object diagram.

As in the design of the initial object-oriented model of the Trusty Furniture Company's order processing system, the last step to be done in the modification of the system was to generate and check the object diagram for the new function *Customer pays for goods at time of purchase*. After the object diagram was checked for consistency it was necessary to define the objects in the diagram using Rational Rose.

7.8. Applying the MOOSE Metric Set

At this stage, both of the object-oriented models of the Trusty Furniture Company's order processing system were complete. The next step is to apply Chidamber and Kemerer's (1994/1991) MOOSE set of metrics for measuring complexity in object-oriented models. Before it was possible to start this step it was necessary to create a form for recording the data obtained from the MOOSE metric using a spreadsheet. The spreadsheet lists the name of the class and the names of the different metrics used to measure different areas of complexity. Another spreadsheet was also setup to record the names of the operations contained within each class. The reason for listing all the operations on one page is to make the process of calculating some of the metrics which use the operations easier.

The metrics were first applied to the original object-oriented version of the Trusty Furniture Company's order processing system, then to the changed version of the system, and finally to the example system given by Booch (1994, p.390). The purpose of this example system is to act as a control for the metrics.

7.8.1 Depth of inheritance tree.

It was decided to start measuring the complexity by using some of the metrics that were easier to use first. Thus, the first metric to be calculated was that of *Depth of Inheritance Tree*. In order to calculate the *Depth of Inheritance Tree* the class diagram for the order processing system was used.

The actual process of measuring the *Depth of Inheritance Tree* involved counting the number of ancestors that a particular class inherits. That is, it is necessary to count the number of levels of inheritance above a specified class. For example, in the original object-oriented model, the class *Order Document Set* inherits the characteristics of the class *Valid Order* which in turn inherits the characteristics of the class *Order* (see figure 16 below). Thus, the *Depth of Inheritance Tree* value for *Order Document Set* is 2.

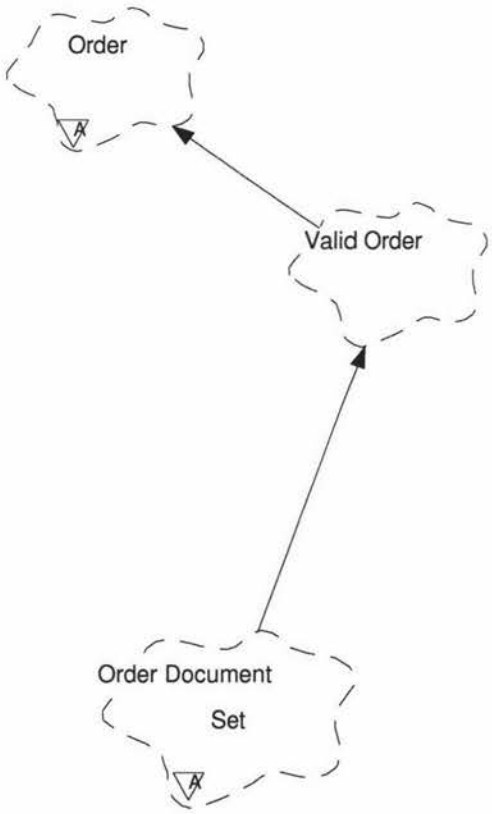


Figure 16: Depth of Inheritance Tree for the class Order Document Set.

7.8.2 Number of children.

Once the *Depth of Inheritance Tree* metric had been calculated, the next metric that was done was that of the *Number Of Children*.

Like the *Depth of Inheritance Tree* metric, the *Number Of Children* is used to measure a particular aspect of complexity in terms of inheritance within an object-oriented system. However, unlike the *Depth of Inheritance Tree*, the *Number Of Children* metric is measured by counting the number of sub-classes that directly inherit from the super-class. An illustration of this point would be the class *Order* in the original object-oriented model. In this particular case, the *Number Of Children* is 2 because *Order* has two sub-classes: *Invalid Order* and *Valid Order*.

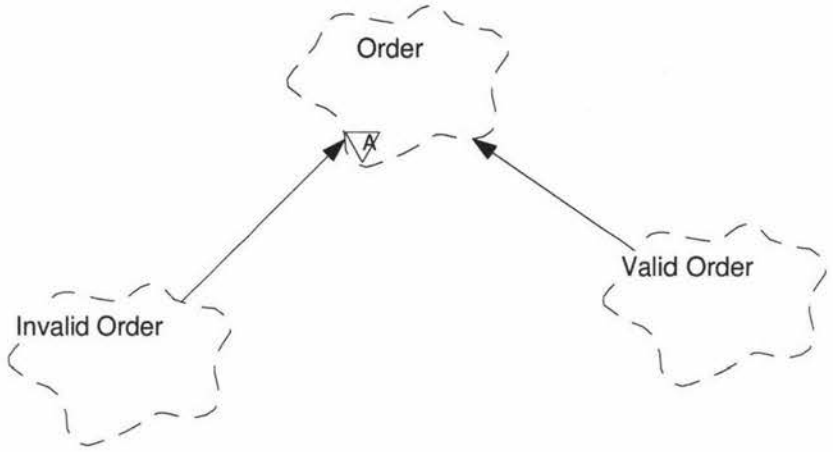


Figure 17: Number Of Children for the class Order

7.8.3 Weighted methods per class.

In order to measure the *Weighted Methods per Class*, it was necessary to make an assumption. This assumption is based on a description by Sharble and Cohen (1993) of how to use this particular metric. Sharble and Cohen (1993) state that the complexity of individual methods "can be considered unity" (p.67).

What this means is that it is assumed that the complexity of each individual method in a class is equal to one, thus making the process of working out this metric simpler. One of the main reasons that this assumption is needed is that the alternative methods of calculating this method require the complexity to measured on the basis of the number of lines of code. However, since this study contains no code, such a process was not viable. As a result of the previous assumption, it was necessary to count the number of methods present in each class and then record the values obtained from this exercise. For instance, in the class *Valid Order* was found to have five methods in the original object-oriented model of the Trusty Furniture Company's order processing system. Therefore, as a result of the assumption, the value of the *Weighted Methods per Class* metric for *Valid Order* is 5.

7.8.4 Coupling between objects.

The process of calculating the *Coupling Between Object* metric proved to be significantly easier than was first anticipated. This was a result of the description provided by Sharble and Cohen (1993) on how to apply this metric. Unlike the definition provided by Chidamber and Kemerer (1994/1991), Sharble and Cohen's (1993) definition of this metric is a lot easier to understand. Thus, the application of this metric to the original version of the object-oriented model involved counting the number of relationships between the classes other than the inheritance relationships. An example of this point would be the class *Customer*. As can be seen in figure 18 below, *Customer* is coupled to seven other classes through methods other than inheritance. Thus, the *Coupling Between Objects* value for the class *Customer is 7.*

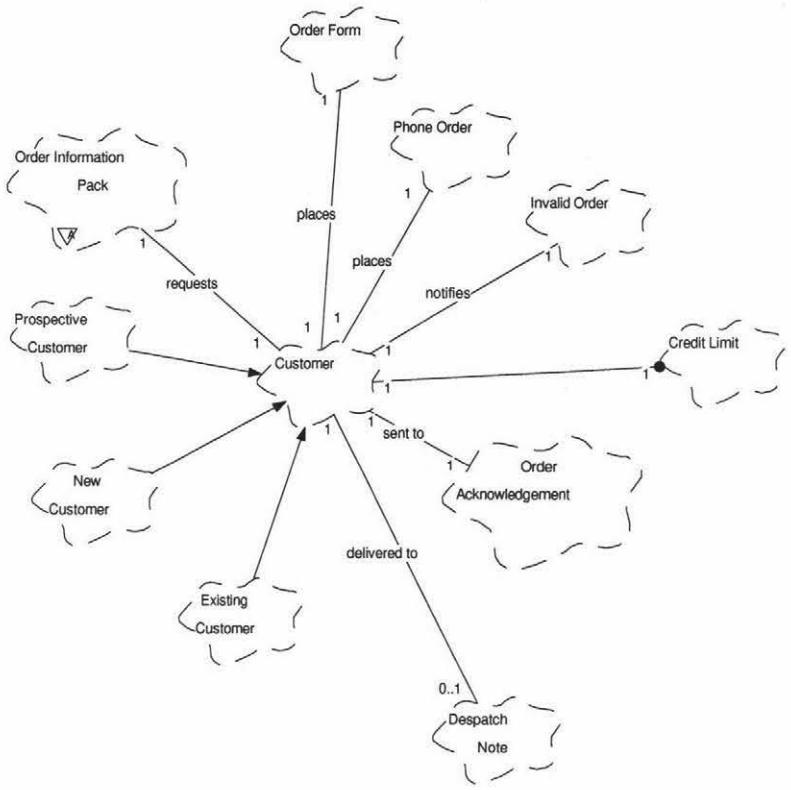


Figure 18: Coupling Between Objects for the class Customer

7.8.5 Response for a class.

The next metric from the MOOSE metric set to be used was that of *Response For a Class*. At this point in using the MOOSE metrics it was found that it was getting progressively harder to apply the metrics to an object-oriented model as it appears as though the metrics were originally intended to be applied to object-oriented code. Despite this problem it was found that the *Response For a Class* metric could indeed be applied to an object-oriented model by using the methods which had been defined earlier.

This particular metric involved totalling the number of methods in each class and then adding all of the other methods that they directly invoked (Sharble and Cohen, 1993). In order to carry out this procedure it was necessary to obtain a report from Rational Rose which showed all of the methods which had been defined. This report on the methods was then used with the interaction diagrams to identify which methods directly invoked other methods. A good example of this process is shown in the class *Credit Limit* which consists of three methods: *getCreditLimit*, *OrderNotValid*, *OrderValid*. In addition to these three methods, the method *InvalidOrder* from the class *Invalid Order* is invoked by the arrival of *OrderNotValid*. Thus, the *Response For Class* value for *Credit Limit* in the original version of the object-oriented model is 4.

7.8.6 Lack of cohesion in methods.

The final MOOSE metric to be applied to the original object-oriented version of the Trusty Furniture Company's order processing system was that of *Lack of Cohesion in Methods*. As with the previous metric, *Response For Class*, the *Lack of Cohesion in Methods* metric involved using the methods that were defined during the development of the object-oriented model.

The *Lack of Cohesion in Methods* metric is specifically interested in measuring the similarity of the attributes in each method contained within a class. An important consideration in calculating the *Lack of Cohesion in Methods* value is that the formula which is used must give an absolute value.

This means that the value obtained by the formula must always be positive. Thus, if a result is negative, then it automatically looses the negative sign in order to make it positive.

The first step in this process is to identify those methods in a class which contain a null set. A null set is a method which contains no attributes that are present in other methods within that class. An illustration of a null set is given by the method *getCustomerType* in the class *Customer*. In the case of *getCustomerType*, the only attribute present is that of *Customer Type*. Further to this, the attribute *Customer Type*, is not present in any of the other methods contained in the class *Customer*. Therefore, *getCustomerType* is considered to be a null set.

The next step in calculating the *Lack of Cohesion in Methods* metric is to find all of the methods in a class that are non-empty sets. Thus, it is necessary to find all of the methods in a class which have one or more attributes present in another method within the same class. An example of this point in the *Customer* class is given by the following methods: *CorrectedCustomerOrder* and *CustomerOrder*. Examination of these methods shows that both of them share common attributes, in this case *Order Details* and *Order Quantity*. Therefore, both of these methods are considered to be non-empty sets. In order to be considered a non-empty set, a method must contain at least one attribute in common with another method in the same class.

Finally, the last step in calculating the *Lack of Cohesion in Methods* for a class involves subtracting the number of non-empty sets from the number of null sets. For example, in the case of the class *Customer* it was found that there were five non-empty sets and one null set, thus the *Lack of Cohesion in Methods* value for the class *Customer* is 4 (that is, one null set minus five non-empty sets equals -4 which becomes 4 since the formula must return an absolute value).

7.9. Applying MOOSE to the Modified System

Once all six MOOSE metrics had been applied to the original object-oriented model of the Trusty Furniture Company's order processing system they were then applied to the object-oriented model which had been changed. The MOOSE metrics were applied in the same manner to the new version of the object-oriented model as they were to the original model as described above. This process was also applied to the example system provided by Booch (1994).

7.10. Summary

The process of modelling the Trusty Furniture Company's order processing system using Booch's object-oriented methodology involved several stages designed to identify and model the classes and their relationships correctly. In this case it was decided only to model the system using class diagrams, interaction diagrams, and object diagrams because this study is more concerned with the software side of maintenance rather than the hardware side.

Once the initial model was completed, the model was changed to reflect the addition of a payment facility for those customers who want to pay for their goods at the time of purchase. Chidamber and Kemerer's (1994/1991) MOOSE metric set was then applied to the before and after versions of the object-oriented model of the order processing system in order to measure the change in complexity. The above process was also repeated on an example system provided by Booch (1994), which acted as a control for this part of the study.

CHAPTER 8: RESULTS

8.1. Introduction

This chapter is intended to discuss the results obtained from this study both in terms of the changes in complexity within each model and in terms of the changes in complexity between both models.

8.2. How Comparisons Were Made

In order to make meaningful comparisons between the before and after versions of each model, it is necessary to express the modification as the percentage changed. This idea has also been extended to the comparisons between the different methodologies. The percentage changed figures obtained throughout this chapter have been calculated by using the following formula:

$$y = \frac{(a - b)}{b} \times 100$$

In the above equation *a* is the after value, *b* is the before value, and *y* is the answer as a percentage.

8.3. Structured Results

The first set of models to be discussed will be the structured versions of the Trusty Furniture Company's order processing system. Since the structured models were measured using two different methods to measure the complexity in the dataflow diagram and the entity relationship diagram, it is necessary to start by discussing the results obtained from the dataflow diagram since it was completed first.

8.3.1 Complexity of the dataflow diagram.

The complexity of the dataflow diagram was measured using McCabe's *cyclomatic complexity* metric. As was discussed earlier in chapter 5, in order to be able to apply McCabe's complexity metric it was first necessary to convert the lower level of the dataflow diagram into flowgraphs for both the original version and the revised version of the Trusty Furniture Company's order processing system. The results obtained from McCabe's complexity metric are shown below in table 1.

	Complexity Before Changes	***Complexity After Changes***
Trusty Furniture Company Order Processing System	3	4

Table 1: **Results of McCabe's Complexity Metric Before and After Changes Were Made to the Structured Model**

8.3.1.1 Analysis of the dataflow diagram results.

As can be seen from the above results, the complexity of the structured version of the order processing system has had a slight increase in complexity after the changes to the system were made. The change in the level of complexity from 3 to 4 is a change of 33.33 percent. This change is the result of the addition of one decision node, namely, *process payment*. Because both the before and after levels of complexity are less than 10, the structured version of the Trusty Furniture Company's order processing system remains easy to maintain. Had the values obtained for the structured system be over 10 then this would be an indicator that the system needed to be redesigned and partitioned further. Thus, the results obtained with McCabe's *Cyclomatic complexity* metric suggest that while the complexity of the system has increased, the level of complexity within the order processing system is still at an acceptable level that is likely to be less troublesome and more reliable than a system which obtained a result of 10 or more.

From the results given above, it can be argued that as a system evolves and new functions are added to meet new user demands, that the complexity of the system as a whole should increase due to a higher number of decisions being made by the system. Such a statement suggests that at some point in the system's life the use of McCabe's complexity metric will yield values greater than 10. Thus, the need to further partition a large system is necessary in order to increase its useful life.

In this particular case, the changed version of the Trusty Furniture Company's order processing system can only handle the addition of six more decision nodes before it will be necessary to redesign and repartition the order processing system.

The number of decisions appears to be determined by the content of what the new events process rather than by the addition of a new event itself. Because of this factor, an increase in the number of decisions in a system will lead to an increase in the complexity of the system, thereby making the process of maintenance more difficult to carry out.

8.3.2 Complexity of the entity relationship diagram.

The second metric which was used to measure the complexity both before and after changes to the structured model was the *Coupling Between Objects* metric from the MOOSE metric set. The results obtained using this method are shown below in table 2.

Entity Name	***CBO Before Changes***	***CBO After Changes***
Customer	1	1
Order	2	3
Payment	Not Applicable	1
Product	2	2
Product Brochure	1	1
TOTAL:	6	8

Table 2: **Results of the Coupling Between Objects Metric Applied to the Entity Relationship Diagram Before and After Changes Were Made to the Structured System**

8.3.2.1 Analysis of the entity relationship diagram results.

As can be seen from the above results, the increases in the *Coupling Between Objects* figure tends to be isolated to only those entities which are directly affected by the changes which were made to the Trusty Furniture Company's order processing system. In this particular case, the addition of a payment system has lead to increases in the *Customer* and *Payment* entities.

The results obtained using the *Coupling Between Objects* metric shows an increase in coupling and therefore an increase in complexity. In terms of the values obtained, lower values are generally considered to be less complex and easier to maintain by authors such as Sharble and Cohen (1993). Thus, in this particular system it appears that the original version of the Trusty Furniture Company's order processing system would be easier to maintain since it is less complex than the changed version.

The total value of the *Coupling Between Objects* metric for the structured system has changed from 6 couplings in the original version to 8 couplings in the changed version. In percentage terms, such a change is equal to a 33.33 percent change in complexity between the two versions as a result of the addition of a payment facility for those customers who wish to pay for their order at the time of purchase.

The results show that the addition of new user needs will result in an increase in coupling between entities. Further to this statement, it does not matter what type of entity is added to the system, as there needs to be a relationship between the new entity and at least one of the existing entities. Thus, as more entities are added during the system's life the complexity of the system will continue to grow as the system evolves. Such increases in complexity are inevitable in any system which has to be responsive to changes in user needs and an order processing system is a good example of such a system. The only ways of limiting increases in complexity would be to either redesign the system, or not add any new features to the system. While the last option is valid, it would lead to a system that would become obsolete very rapidly as it would not be responsive to the changing needs of the system's users.

The results that were obtained for the dataflow diagram showed a 33.33 percent change in complexity, in addition the results from the entity relationship diagram also indicated a 33.33 percent change in complexity. Both results are consistent in that they show the same level of change in the complexity of the structured system. This is despite the fact that more modifications were made to the dataflow diagram. Such a result tends to suggest that similar changes to the system will result in the dataflow diagram increasing in complexity by the same level as the entity relationship diagram.

While some of this similarity in complexity can be attributed to the different methods that were used in the measurement of the different diagrams, some of the difference can also be attributed to what was changed on each of the diagrams. In the case of the dataflow diagram, changes involved the addition of new datastores, dataflows, and a new process, whereas changes to the entity relationship diagram only involved the addition of a new entity and a new relationship. Further, a lot more changes were made to the dataflow diagrams than to the entity relationship diagram. This is because the changes to the order processing system had to be shown over several different levels of the dataflow diagram instead of just one level as was the case with the entity relationship diagram. Thus, the dataflow diagram is likely to increase by a smaller proportion than the entity relationship diagram due to the size of each model.

Despite the fact that several changes had to be made to the dataflow diagram on several different levels, McCabe's complexity metric only measured the changes on the lowest level of the dataflow diagram. This fact makes comparisons between the two diagram much easier to make.

8.4. Object-Oriented Results

The next set of models to be discussed will be that of the object-oriented version of the Trusty Furniture Company's order processing system.

Both versions of the order processing system were measured for complexity using all six of the metrics outlined in Chidamber and Kemerer's (1994/1991) MOOSE metrics set. A summary of the results obtained using the MOOSE metrics set is shown below in table 3 (for more detailed results see appendix G).

	WMC	*DIT*	*NOC*	*CBO*	*RFC*	*LCOM*
Before	32	29	16	34	49	25
After	37	32	19	42	53	28
Percentage Change	15.6	10.3	18.75	23.5	8.2	12

Table 3: Summary of the Results from the MOOSE Metric Set Before and After Changes Were Made to the Object-Oriented Model

As can be seen from the above results, the addition of the payment facility to the Trusty Furniture Company's order processing system has lead to an increase in the values of all six of the MOOSE metrics. In order to be able to better understand these results it is necessary to discuss each of the six metrics in turn.

8.4.1 Analysis of the weighted methods per class results.

The first of the MOOSE metrics to be discussed will be that of the *Weighted Methods per Class* (WMC) metric. The above results show that there was a 15.6 percent increase in the complexity of the classes in the object-oriented system after the change was made.

The result of this increase is that the classes in the changed version of the order processing system will need more maintenance as they are becoming more complex. This statement is supported by Chidamber and Kemerer (1994) who say that "the number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class" (p.482).

While the overall increase was 15.6 percent, the increase for the *customer* class alone was 16.7 percent, which is a slightly greater increase than that of the whole order processing system. It must be noted that the only class in the original version of the system to be affected by changes in the *Weighted Methods per Class* metric was that of the *customer* class. The other class that was affected was the new class, *payment*.

Another observation of the results obtained using the *Weighted Methods per Class* metric shows that all of the classes in both versions of the order processing system tend to have small numbers of methods. In comparison to the systems given by Chidamber and Kemerer (1994) the *Weighted Methods per Class* figures are generally much lower in this study.

Examples of this point from the systems studied by Chidamber and Kemerer (1994) include the fact that the *Weighted Methods per Class* figure ranges from 0 to 106 with a median of 5 in system A, and from 0 to 346 with a median of 10 in system B. While some of these figures are similar to this study, there is not such a wide range of extreme values in this case. Because there are a small number of methods in each class, it tends to suggest that the classes will be able to be reused more easily. Chidamber and Kemerer (1994) support this point of view as they argue that "classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse" (p.482).

8.4.2 Analysis of the depth of inheritance tree results.

The next member of the MOOSE metric set to be discussed is the *Depth of Inheritance Tree* (DIT) metric. *Depth of Inheritance Tree* is a metric which is used to measure the complexity of inheritance within an object-oriented system.

The results for this study show that overall, the *Depth of Inheritance Tree* figure increased from 29 to 32. This change in the figures represents a 10.3 percent increase in the complexity of inheritance in the new version of the Trusty Furniture Company's order processing system.

Closer examination of the results reveals that the entire increase in the *Depth of Inheritance Tree* value was caused by the addition of three new classes, *cash*, *cheque*, and *credit card*.

All three of these classes inherit features from the *payment* class, which is also a new addition to the order processing system. Thus, these results show that the changes to the inheritance within the order processing system were isolated to those classes which were involved in the payment process.

While there was no change in the values of any of the classes present in the original version of the order processing system, the *Depth of Inheritance Tree* values were highest amongst those classes that were part of the *Order Document Set*. The six classes involved in the *Order Document Set* each had a value of 3. This figure is the deepest level of inheritance in the order processing system. Because these six classes are at a deeper level in the inheritance structure, they are therefore considered to be more complex than other classes that may have only one or two levels of inheritance above them. According to Chidamber and Kemerer (1994) "the deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour" (p.483).

Despite this complexity, it can also be argued that classes at a deeper level in the hierarchy are generally better than those classes that are not at such a deep level. The reason for this advantage is because there is a greater potential for the reuse of inherited methods (Chidamber and Kemerer, 1994).

Because the *Depth of Inheritance Tree* metric is used to measure the depth of the inheritance hierarchy, Sharble and Cohen (1993) state that "generally, it is better to have depth rather than breadth, since this promotes reuse and reduces redundancy in the system" (p.72).

8.4.3 Analysis of the number of children results.

Related to the *Depth of Inheritance Tree* metric is the *Number Of Children* (NOC) metric which also measures complexity in the inheritance hierarchy. However, unlike the *Depth of Inheritance Tree* metric which measures the depth of the inheritance in a system, the *Number Of Children* metric measures "the breadth of the inheritance hierarchy" (Sharble and Cohen, 1993, p.72).

The results in table 3 show that the overall *Number Of Children* for the order processing system increased from 16 to 19. Such a change in the *Number Of Children* represents an increase of 18.75 percent over the original version of the Trusty Furniture Company's order processing system. Such an increase in the *Number of Children* means that the changed version of the system is more complex than the original version.

According to Chidamber and Kemerer (1994), "if a class has a large *number of children*, it may require more testing of the methods in that class" (p.485). Thus, the class *Order Document Set* will require a greater deal of testing due to the fact that in both versions of the order processing system it has a value of 6 for the *Number Of Children*.

Further to this, Sharble and Cohen (1993) argue that "a high value of *Number Of Children* indicates that classes high up in the inheritance hierarchy can potentially influence a large number of other classes.... This increases the amount of testing required to verify the system originally and makes it much more difficult to safely modify the system later on" (p.72). Therefore, the high value obtained by the class *Order Document Set* tends to suggest that there may be problems modifying the order processing system at a later stage in its life due to the complexity of this particular class.

Further examination of the results reveals that like the *Depth of Inheritance Tree* metric, the *Number Of Children* metric only shows an increase in one of the new classes, *Payment,* and no change in any of the other classes in the system. Such an observation suggests that like the *Depth of Inheritance Tree* metric, the *Number Of Children* metric shows that the changes to this particular system are isolated to the new classes that were added as a result of the changes to the order processing system.

8.4.4 Analysis of the coupling between objects results.

The next metric in the MOOSE metric set to be discussed will be that of the *Coupling Between Objects* (CBO) metric. *Coupling Between Objects* is used to measure the complexity of the interaction between classes (Sharble and Cohen, 1993). This particular metric shows the greatest increase in the complexity of the Trusty Furniture Company's order processing system.

The results obtained for the *Coupling Between Objects* metric shows an increase from a value of 34 in the original version, to a value of 42 in the changed version of the order processing system. This change represents a 23.5 percent increase in the *Coupling Between Objects*.

Such a significant increase in the overall *Coupling Between Objects* suggests that maintenance will become more difficult as changes to one object may have an effect on another object. This idea is supported by Chidamber and Kemerer (1994) who state that "the larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult" (p.486). Thus, those individual classes which obtained a low value from the *Coupling Between Objects* metric are considered to be better for maintenance since they are not coupled to a large number of other classes, and are therefore less complex.

On an individual basis, the only classes in the original system that changed were the *Customer* class and the *Order* class. Both of these classes had very significant increases in complexity.

For instance, the *Customer* class increased its *Coupling Between Objects* value by 42.9 percent, and the *Order* class increased its value by 33.33 percent. The fact that both of these classes have relatively high *Coupling Between Object* values supports the idea that "classes that control other classes exhibit a correspondingly high value of Coupling Between Objects" (Sharble and Cohen, 1993, p.70).

The reason for this fact is that both the *Customer* class and the *Order* class do control other classes within the order processing system. This is especially true in the case of the *Customer* class, as it controls classes such as *Payment* and *Order*.

On the other hand, it can also be argued that such high values of *Coupling Between Objects* are an indicator of excessive coupling between classes. Such excessive coupling is considered to be bad for modular design since it prevents reuse (Chidamber and Kemerer, 1994). Obviously, excessive coupling will increase the complexity of any maintenance later on. Thus, classes such as *Customer* and *Order* may need to be redesigned in order to make them more modular so that they can be better maintained and reused at a later stage in the system's life.

8.4.5 Analysis of the response for a class results.

Response For a Class (RFC) was the next MOOSE metric to be applied to the Trusty Furniture Company's order processing system. *Response For a Class* is a metric that is designed to measure the complexity of the classes and the interactions in the system.

The results show that once again an increase in complexity occurred. In this case the *Response For a Class* value rose from 49 in the original version to 53 in the changed version of the Trusty Furniture Company's order processing system. Such an increase is equal to an 8.2 percent rise in the value of the overall *Response For a Class*. Thus, the potential communication between classes in the order processing system has increased (Chidamber and Kemerer, 1994).

However, while the increase in the overall value of the *Response For a Class* metric can mean increased communication, it can also mean that the complexity of the individual classes has increased because more methods can be invoked. This point of view is supported by Chidamber and Kemerer (1994) who argue that "the larger the number of methods that can be invoked from a class, the greater the complexity of a class" (p.487).

Further to the previous point of view, an increased number of methods means that maintenance becomes a much more complicated task. The reason for this complexity is due to the fact that "if a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester" (Chidamber and Kemerer, 1994, p.487). Such an increase in the number of methods is evident in the class *Customer*, which has increased its complexity by 10 percent. This increase tends to suggest that maintenance of this class will be more difficult later on than some of the other classes in the order processing system.

While the value of the *Response For a Class* metric remained static at 13 for the *Valid Order* class, it is necessary to discuss this particular case since it is the highest value obtained for the order processing system. The reason that this particular class has such a high value is due to the number of methods which *Valid Order* invokes.

In the order processing system the methods invoked by the *Valid Order* class perform a lot of the critical functions that determine the validity of a customer's order. While the high value of *Response For a Class* means that this class communicates with a lot of other classes in the system, it also means that it is one of the more complex classes to maintain due to the high level of communication.

Of the four new classes that where added to the Trusty Furniture Company's order processing system, only *Payment* obtained a value for the *Response For a Class* metric. This value is reasonably low in comparison to the values obtained for the classes in the original version of the system. Such a low value suggests that the *Payment* class has a low level of communication with other classes in the system and will therefore be easier to maintain due to the decreased level of complexity.

8.4.6 Analysis of the lack of cohesion in methods results.

The last of the MOOSE metrics to be applied to the object-oriented versions of the Trusty Furniture Company's order processing system was that of the *Lack of Cohesion in Methods* (LCOM) metric.

The *Lack of Cohesion in Methods* metric is used to measure the complexity of a class in an object-oriented system in terms of "the lack of cohesion among the methods of a class, or how many unrelated activities a class is performing" (Sharble and Cohen, 1993, p.68).

The results gained from the use of the *Lack of Cohesion in Methods* metric show that once again there has been an increase in the complexity of the system as a result of the addition of a payment facility for the customers. These results show an increase from a value of 25 in the original version, to a value of 28 in the changed version of the order processing system. Such an increase constitutes a 12 percent increase in the complexity of the order processing system overall.

The biggest increase in an individual class is seen in the *Customer* class which increased its complexity by 25 percent. The *Customer* class and the *Order Document Set* class both have some of the higher values for the *Lack of Cohesion in Methods* metric. Such high results suggest that these classes are performing more unrelated activities than the rest of the classes in the system. It can also be argued that a high value of *Lack of Cohesion in Methods* is an indicator of a class which is being controlled (Sharble and Cohen, 1993). This is related to the idea that classes with a high *Coupling Between Objects* control other classes (Sharble and Cohen, 1993).

The fact that the class *Customer* has high values for both the *Lack of Cohesion in Methods* metric and the *Coupling Between Objects* metric tends to suggest that this class has "an apparent lack of support for the principle of encapsulation" (Sharble and Cohen, 1993, p.71). Thus, in order to reduce the values obtained for this class it would be necessary to redesign the *Customer* class to better support the idea of encapsulation.

An alternative view of the high values obtained for the *Customer* class and the *Order Document Set* class is provided by Chidamber and Kemerer (1994) who state that "low cohesion increases complexity, thereby increasing the likelihood of errors during the development process" (p.489).

Because a high value is an indicator of low cohesion, these two classes are considered to be more complex than those classes which obtained a low value from the *Lack of Cohesion in Methods* metric. If the *Customer* class and the *Order Document Set* class had provided good support for encapsulation, then they would be considered to be less complex because the methods that they used would be of a similar nature. Thus, maintenance would be less difficult to carry out as the methods would perform similar processes, rather than a number of different process which would be harder to control, as the affects of the different methods would need to be taken into account.

The only new class to obtain a result from the *Lack of Cohesion in Methods* metric was that of the *Payment* class which has a value of 2. This result shows that this class is less complex than that of the *Customer* class and therefore can be considered to have better support for encapsulation.

8.5. Comparisons Between the Structured and Object-Oriented Models

While a number of different metrics were used in order to find the change in complexity within the structured and the object-oriented models of the Trusty Furniture Company's order processing system, some comparisons can be made in terms of the percentage change in complexity rather than in terms of the actual figures obtained.

In general, it can be said that the changes to the structured model resulted in a greater increase in complexity than the changes to the object-oriented version. While the results for the structured model showed an increase of 33.33 percent for both the dataflow diagram and the entity relationship diagram, none of the metrics used to measure the object-oriented system showed as great an increase in complexity. In comparison, the metrics used to measure the object-oriented model measured increases ranging from 8.2 percent to 23.5 percent.

8.5.1 Comparisons between the coupling between objects results.

It is possible to make a direct comparison between the entity relationship diagram and the *Coupling Between Objects* value in the object-oriented system because they both used the same metric.

In the structured system, the *Coupling Between Objects* metric yielded an increase of 33.33 percent overall compared with a value of 23.5 percent for the object-oriented version of the model. The overall percentages for this metric show a slightly greater increase in the structured system's level of complexity. Such a result suggests that a greater number of relationships were added to the structured system, even though this is not the case. In order to compare the individual results from both systems it is necessary to compare both systems on an equal basis. As a result, only those classes that have an equivalent entity in the entity relationship diagram are shown in appendix I. The results in appendix I will be discussed below.

8.5.2 Comparing coupling between objects on an individual basis.

The results of the comparison between the structured model and the object-oriented model in terms of the *Coupling Between Objects* metric on the individual entities/classes shows a different result to the overall increases in complexity. On an equivalent basis, it would appear that the increase in the *Coupling Between Objects* is slightly higher for the object-oriented version which shows an increase of 38.5 percent as opposed to the 33.33 percent increase in the structured system.

The removal of those classes which did not have a similar entity in the entity relationship diagram has resulted in a slight increase in the complexity of the object-oriented model. The most obvious area in the object-oriented model which has caused this effect is that of the *Customer* class which has increased by three points from 7 to 10.

In comparison, the *Customer* entity in the entity relationship diagram has not increased at all. The high values in the object-oriented model suggest that the *Customer* class is more important in terms of control in the object-oriented model than it is in the structured model.

While several classes in the object-oriented model have lead to a higher result than the structured model, it is important to keep things in perspective. In general, the increases in most of the classes/entities have been of similar proportions. That is, in most of the classes/entities, the values have either stayed the same, or they have increased by the same proportion in both models. An example of this point can be seen in the *Order* class/entity which has increased its value by one in both the structured model and the object-oriented model. Thus, in the context of the changes overall, it would appear that the values obtained for *Customer* class are outliers as they do not represent the overall level of increases within the system. Therefore, had the value for the *Customer* class increased by the same proportion as the *Customer* entity, that is, no increase at all, then the results would have shown an overall increase of 15.4 percent as opposed to 38.5 percent for the object-oriented model.

8.6. Validation of the Metrics Used in this Study

This section will address the issue of the validation of metrics used in this study. The purpose of this validation is to prove that the values obtained from the metrics used in this study are consistent.

8.6.1 Structured metrics validation.

In order to validate the results obtained from the structured models, the metrics were applied to two models that were developed by several extramural students. All of these students had given prior permission for their work to be used for research purposes. These students were members of Massey University's 57.221 Information Systems Analysis class in 1996. As a result both of the models that were used for validating the structured models were based on the same case study, that is, the Trusty Furniture Company. However, in comparison to this study, the student's diagrams had a wider system boundary. Thus, the students also modelled the delivery and the accounts subsystems as well as the order processing system.

8.6.1.1 Dataflow diagrams.

In the case of the dataflow diagrams, it was necessary to convert the lowest level of each dataflow diagram into a flowgraph so that McCabe's *cyclomatic complexity* measure could be applied. These flowgraphs are shown in appendix C. Once the dataflow diagrams had been turned into flowgraphs, McCabe's metric was applied. The results obtained from each of the dataflow diagrams can be seen below in table 4.

	Student Number 1	*Student Number 2*	*Order Processing System(Before)*	*Order Processing System(After)*
Trusty Furniture Company's Information System	2	3	3	4

Table 4: **Results of McCabe's Complexity Metric After Being Applied to the Student Models in Comparison to the Trusty Furniture Company's Order Processing System**

As can be seen above, the results from both models are indeed similar to the figures gained in both the before and after versions of the model developed in this study, especially in the case of the results obtained from *Student 2*. In this study the before value was 3 and the after value was 4. Thus, the results of this validation exercise have proved that McCabe's complexity metric does indeed give consistent values in similar systems.

8.6.1.2 Entity relationship diagrams.

One of the problems that arose early on in this study was how to measure the complexity of an entity relationship diagram. Originally, McCabe's complexity metric was going to be applied, however this proved to be very difficult, if not impossible to do.

Thus, after using the MOOSE metric set on the object-oriented models it was found that the *Coupling Between Objects* metric could probably be applied to the entity relationship diagram.

This decision raises the question of *how valid is it to apply a metric that was initially designed to measure the coupling in an object-oriented system to the coupling in a structured system?*. In order to answer this question, it is necessary to think about what the *Coupling Between Objects* metric is ultimately trying to achieve. The *Coupling Between Objects* metric is used to measure the complexity of each class in terms of the number of relationships that exist between one class and another class, other than those linked by inheritance relationships. Thus, the more relationships that exist, the more complex the class. While inheritance does not exist in a structured system as such, inheritance is similar to the super entity/sub-type entity combination of an is-a relationship in some entity relationship diagrams. Thus, if the number of relationships that an entity has are totalled, less those relationships that are part of an is-a relationship, then it is possible to show the complexity of each entity in an entity relationship diagram.

As in the object-oriented model a higher *Coupling Between Object* value in the structured system is an indicator of an entity that is more complex than another entity with a lower value. In the case of a structured system, it would be better to refer to the *Coupling Between Objects* metric as the *Coupling Between Entities* metric instead, since this better reflects what is being measured.

In terms of the results obtained from the *Coupling Between Objects* metric when it was applied to the student's entity relationship diagrams it was found that the student's results had a range of 4 values. That is, the value obtained was within 4 values of each other, unlike the results from the entity relationship diagram in this study which had a range of 2 values. The results for the student entity relationship diagrams can be seen below in table 5.

	Student Number 1	***Student Number 2***	***Order Processing System(Before)***	***Order Processing System(After)***
Trusty Furniture Company's Information System	18	14	6	8

Table 5: **Results of the Coupling Between Objects Metric After Being Applied to the Student Models in Comparison to the Trusty Furniture Company's Order Processing System**

As can be seen from the above results, the values from the student systems are over twice the value of the results obtained in this study. In this case the *Coupling Between Objects* metric returned values of 6 and 8 for the before and after models respectively. The fact that the student models obtained higher values is consistent with the fact that the student's models are more complex since they also model other aspects of the order processing system. Thus, it can be seen that the *Coupling Between Objects* metric is capable of producing consistent results from entity relationship diagrams which are based on the same system.

8.6.2 Object-oriented metrics validation.

The process of trying to validate the MOOSE metrics set was done by applying each of the metrics where possible to an order processing system developed by Booch (1994, p.390). One of the problems that occurred was the fact that Booch's (1994) model does not make any use of inheritance whatsoever, further to this, the model does not give any details of the attributes of the classes. These problems meant that some of the metrics were unable to be calculated. Of those metrics that were able to be calculated, a summary of the results obtained is shown below in table 6.

	WMC	*DIT*	*NOC*	*CBO*	*RFC*	*LCOM*
Total for Booch's Order System	21	0	0	20	28	Not Able to be Calculated
Trusty Furniture Company (Before)	32	29	16	34	49	25
Trusty Furniture Company (After)	37	32	19	42	53	28

Table 6: **Summary of the Results From the Application of the MOOSE Metrics to Booch's (1994) Ordering System in Comparison to the Trusty Furniture Company's Order Processing System**

For a more detailed view of the results summarised above see appendix H. It must be noted that the class diagram presented by Booch (1994, p.390) is considerably less complex than the class diagrams developed for this study, as it appears that Booch (1994) is only trying to give an example of typical applications that have had object-oriented techniques applied to them. This lack of complexity is evident in the results obtained after the MOOSE metric set was applied to Booch's (1994) class diagram.

One indicator of this lack of complexity is the number of classes in Booch's class diagram. In Booch's case there are only 9 classes in comparison to 25 classes in the original class diagram of the Trusty Furniture Company's order processing system and 29 classes in the modified version of the same diagram. Thus, one would expect the class diagrams in this study to obtain higher values from the MOOSE metric set, as the more classes that are added to a system the more complex the system becomes. Closer inspection of the number of classes used in this study show that there are 2.8 times as many classes in the original model than there are in Booch's model, and 3.2 times as many classes in the modified version of the Trusty Furniture Company's order processing system. Thus, it can be argued that the MOOSE metrics should give a higher value for a system that has more classes, and is therefore more complex.

8.7. Model Validation

Because neither the structured or the object-oriented model were implemented it is necessary to validate these models in order to show that they are a fair representation of the Trusty Furniture Company's order processing system.

In general, the basis for the validation is the amount of similarity between the models developed in this study and the models of similar systems which have been used for validation.

8.7.1 Structured model validation.

In order to validate the model developed using Yourdon's (1989) modern structured methodology, the model will be compared with the two models developed by the extramural students. Once again it is important to remember that while the models developed by the extramural students used the same case study, they also modelled a system with a wider specification than was the case here. Thus, it is expected that some differences will arise between the models.

8.7.1.1 Dataflow diagram validation.

In order to validate the dataflow diagram the similarities between the terminators and the processes used in each model will be examined.

In terms of the terminators or external entities used, there is a great deal of similarity between the models. This similarity can be seen below in table 7.

	Terminator 1	*Terminator 2*	*Terminator 3*	*Terminator 4*	*Terminator 5*	*Terminator 6*
Trusty Furniture Company (Before)	Customer	Production	Accounts	Finance Director	Despatch	Sales
Trusty Furniture Company (After)	Customer	Production	Accounts	Finance Director	Despatch	Sales
Student 1	Customer	Production	Finance and Operations	Finance and Operations	Despatch	
Student 2	Customer	Production	Accounts			

Table 7: Summary of the Terminators Used on the Context Diagram

As can be seen above, both of the student's models agree in terms of the terminators *Customer* and *Production*. Thus it can be said that these terminators are valid since they are in agreement with both of the models used for validation.

In terms of the four remaining terminators, the terminator *Accounts* is in agreement with student 2 but not with student 1. The reason for this appears to be caused by the fact that student 1 has combined the *accounts* terminator into the *finance and operations* terminator as seen by the inclusion of the dataflow *order details*.

Therefore, it can be argued that the inclusion of the terminator *accounts* is valid since it uses the same type of dataflow as its equivalent terminators in the validation models that were created by the extramural students. This argument is further strengthened by the fact that the terminator is in agreement with student 2's model.

The process of validating the next three terminators is made more difficult by the fact that student 2 has no equivalent terminators. In the case of the terminator *Finance Director*, it would appear that while student 1 has not used the same name, the terminator *finance and operations* is carrying out the job of approving orders on credit as evidenced by the dataflow *approved order* on student 1's context diagram. This fact tends to suggest that the approval of credit orders is a common function in both the student's model and in the models developed here. Thus, the terminator *Finance Director* can be seen as being valid since it is receiving the data needed to perform the processing of approving credit orders.

The next terminator to be validated is that of *Despatch*. In this particular case, the terminator from student 1's model is also called *Despatch*. Thus, suggesting that *despatch* is a common terminator in the Trusty Furniture Company's system. It can therefore be argued that since student 1's model agrees with the inclusion of an *despatch* terminator, that the terminator must be valid.

The final terminator to be validated is *Sales.* The fact that this terminator is not present on the student models suggests that it may not be a valid terminator. However, this is not the case as the *Sales* terminator is needed to represent the sales department which receives copies of the order form. Thus, the *Sales* terminator needs to be included and therefore it is valid.

In terms of the processes which are present on the lowest level of the dataflow diagrams, it is necessary to see the similarities in order to see whether the processes are valid. A summary of the processes is shown below in table 8.

Trusty Furniture Company (Before)	**Trusty Furniture Company (After)**	**Student 1**	**Student 2**
Respond to Order Information Query	Respond to Order Information Query	Accept New Customer	Accept Customer Information
Respond to Receipt of Order	Respond to Receipt of Order	Maintain Existing Customer Information	Accept Order Details
Validate Order	Validate Order	Verify Product Details	Accept Approved Order
Determine Orders Status	Determine Orders Status	Record Despatches	Produce Outstanding Order Report
Process Orders	Process Orders	Maintain Order Information	Ensure Order Fulfilment
Produce Order Summaries	Produce Order Summaries	Manage Reports	Produce Summary Reports
Produce Delivery Plan	Produce Delivery Plan	Prepare Delivery Plan	Create Delivery Plan
	Process Payment		Produce Chair Order
			Record Chair Deliveries
			Generate Special Instructions Form
			Fulfil Delivery Plan
			Fulfil Order
			Accept Chair Supplier Details

Table 8: Summary of the Processes Used on the Lowest Level of the Dataflow Diagrams

The reason that the above table only concentrates on processes at the lowest level is that at the lowest level the process are in their most basic form, and thus they are easier to compare.

The first process to be discussed will be that of the *Respond to Order Information Query*. It would appear that both students have not included a similar process in their dataflow diagrams. This raises the question of how do the customers request brochures, order forms, or other information from the company in relation to the products? Such processing is obviously needed in an order processing system, thus it can be argued that the *Respond to Order Information Query* processes is valid because it is performing a process which is needed by the rest of the system, as without this process it is not possible to keep track of which customers have received information packs with an order form. This point is important because new customers must place their first order with an order form. Thus, if the new customer has no order form then they need to be sent one.

The *Respond to Receipt of Order* process is similar to the *Accept New Customer* process in student 1's model and the *Accept Customer Information* and *Accept Order Details* processes in student 2's model. The reason for this similarity is due to the fact that the *Respond to Receipt of Order* process is involved in the process of accepting the customer's information and their order. The fact that both students have equivalent processes suggests that the *Respond to Receipt of Order* process is valid.

The *Validate Order* and *Determine Orders Status* processes are similar to the *Verify Product Details* process in student 1's model and the *Accept Approved Order* process in student 2's model as they are involved in the validation of order forms and the approval of those orders being paid on credit. Such a similarity helps to support the idea that both the *Validate Order* and *Determine Orders Status* processes are indeed valid.

The *Process Orders* process appears to be similar to the functions performed by the *Maintain Order Information* and the *Manage Reports* processes in student 1's model. Student 2 does not have an equivalent process. Since this process is involved in the generation of the six part document set it is an important part of the Trusty Furniture Company's system since it is used to coordinate the production, despatch, finance, and ordering systems. The similarities between the *Process Orders* process and the equivalent processes in student 1's model tends to suggest that it is a valid process.

The next process to be validated is that of the *Produce Order Summaries* process. This process has no equivalent process in student 1's model, however the *Produce Order Summaries* process is very similar to the *Produce Summary Reports* process in student 2's model. Both the *Produce Order Summaries* process and the equivalent process in student 2's model perform the same functions in the production of the daily and weekly summaries. This fact tends to support the idea that the *Produce Order Summaries* process is valid despite the fact that there is no equivalent process in student 1's model.

A further process that needs to be validated is that of the *Produce Delivery Plan* process. This process is equivalent to the *Prepare Delivery Plan* process in student 1's model and the *Create Delivery Plan* process in student 2's model. This agreement on the inclusion of a process to produce a delivery plan is evidence that the *Produce Delivery Plan* process is indeed a valid process.

The final process to be validated is that of the *Process Payment* process from the modified version of the Trusty Furniture Company's order processing system. Since neither of the students had to make the changes that were made in this study, it is not surprising to see that there are no equivalent processes in the student's models. It can be argued that this process is valid because it is essential to the processing of the different types of payment in the modified order processing system. Without this process the system is not able to process those orders which have payment included. Thus, the *Process Payments* process is valid, as without it the system cannot process those orders that are not on credit.

As a result of the above validation of the terminators and the processes, it can be argued that the dataflow diagrams developed in this study are reasonable models of the Trusty Furniture Company's order processing system.

8.7.1.2 Entity relationship diagram validation.

The entity relationship diagrams used in this study will be validated on the basis of the similarity between the entities used. The reason for using the similarities on the entities is that the entities are key component of an entity relationship diagram. Further to this, it is likely that most order processing systems will have a number of entities that will be the same, thus if there is a great deal of similarity, then it can be assumed that the entity relationship diagrams used in this case are reasonable models of the Trusty Furniture Company's order processing system. The table below shows the entities used by each system.

Trusty Furniture Company (Before)	**Trusty Furniture Company (After)**	**Student 1**	**Student 2**
Customer	Customer	Customer	Customer
Order	Order	Order	Order
Product	Product	Product	Product
Product Brochure	Product Brochure	Production Batch	Product Batch Requisition
	Payment	Production	Chair Supplier
		Salesperson	Chair Order
		Delivery Plan	Chair Order Line
		Despatch	
		Finance and Operations	

Table 9: Summary of the Entities Used in the Entity Relationship Diagrams

As can be seen above, there is a general agreement over the first three entities, that is *Customer*, *Order*, and *Product*. Thus, we will assume that these three entities are valid on the basis that they are present in both of the student's models.

Product Brochure is one entity which is not present on either of the models developed by the students. The reason for this appears to be due to the fact that both students are more concerned with the production system at this point which is outside the specifications of the system developed in this study. While *Product Brochure* may not be present in the student's models, this does not mean that it is invalid. *Product Brochure* is a valid entity because the Order Processing System needs it to keep track of the brochure inventory levels as well as determine which products are in each brochure.

The final entity to be validated is that of the *Payment* entity in the changed version of the Trusty Furniture Company's order processing system. As can be seen in table 9, *Payment* does not have an equivalent entity in either of the student's models. The reason for this would appear to be due to the fact that the models presented by the students do not show the changes which were made in this study. Thus, it would not be expected to see these modifications in the student's models. *Payment* can be considered to be a valid entity since it is used to show the type of payment method used by a customer, this is information that needs to be remembered by the system.

The above results show that overall, most of the entities used in the entity relationship diagrams are valid due to their similarities with the models developed by the extramural students. While the entity relationship diagram in this study does not include many of the entities that were in the student's models due to differences in scope, the general results support the idea that the entity relationship diagrams used in this study are a fair and reasonable view of the Trusty Furniture Company's order processing system.

8.7.2 Object-oriented model validation.

In order to validate the systems developed using Booch's (1994) methodology, it is necessary to compare the class diagram against the class diagram from a similar system. As with the object-oriented metrics validation, the validation of the object-oriented model will be done against the ordering system developed by Booch (1994, p.390).

Because Booch's (1994) model is not based on the Trusty Furniture Company case study it is to be expected that there will obviously be some major differences between the model. However, due to the fact that Booch's (1994) model is modelling a type of order processing system it is expected that the key classes will be present. It is these classes which are the most important since they are likely to be present in most ordering systems.

A further point which needs to be made is that Booch's (1994) model is far less complex than the Trusty Furniture Company's order processing system, as is evidenced by the fact that Booch's (1994) model has only nine classes compared to 29 classes in the modified version of the order processing system developed in this study. Thus, the validation will concentrate on those classes that have equivalent classes in Booch's (1994) model.

In addition, this study has made extensive use of inheritance throughout the class diagrams. This is one area where the class diagram provided by Booch (1994, p.390) is lacking. All of the relationships in Booch's (1994) model are all of the association relationship kind. The fact that inheritance has been used in this study helps to explain why there are considerably more classes.

The class diagram will be validated on the basis of the similarities between the classes. For a full overview of the classes present in both the Trusty Furniture Company system and Booch's (1994) system see appendix H. A summary of the classes used in each of the systems is shown below in table 10.

Trusty Furniture Company Order Processing System	***Booch's (1994) Ordering System***
Customer	Customer / Customer Record
Order	Order
Product	Product Record
Delivery Plan / Despatch Instruction / Despatch Note	Packing Order
	Order Agent
	Stock Person
	Supplier
	Supplier Record

Table 10: Summary of the Equivalent Classes Used in the Class Diagrams

As can be seen above, six of the classes in the Trusty Furniture Company's order processing system have equivalent classes in Booch's (1994) model. Of these six classes, half of them can be considered to be key classes that would appear in most order processing systems, that is, the *Customer*, *Order*, and *Product* classes.

In the case of the *Customer* class it can be seen that in Booch's (1994) model, the equivalent functions are carried out by two separate classes, *Customer* and *Customer Record*. This suggests that at least in Booch's (1994) case it may have be better to distinguish between the actual customer and the record of the customer which the system keeps.

This does not mean that the *Customer* class in the Trusty Furniture Company's order processing system is totally invalid, as the use of such a system would have been difficult to implement in either of the class diagrams developed for this study. One of the main reasons that it was necessary to combine the classes which Booch (1994) used into one class was due to the use of inheritance to show the different types of customers. This situation is particularly important as the Trusty Furniture Company case study makes a point of distinguishing between the different types of customers and their privileges within the system. Such a situation is not present in Booch's (1994) system, as a customer is considered to be a customer no matter whether they are a new customer or an existing customer. The fact that *Customer* is obviously a key class lends weight to the argument that it is indeed a valid class.

Order is another key class which is obviously needed in all order processing systems, as without an order the system cannot operate. The fact that *Order* is present in Booch's (1994) model as well tends to suggest that *Order* is a valid class.

The class *Product* is equivalent to the *Product Record* class in Booch's (1994) model. The reason for this is due to the similarities between these two classes. One such similarity is the fact that both classes deal with the function of keeping track of the amount of product in stock. Further to this, *Product* is a key class that needs to be in an order processing system, thus *Product* is valid.

Finally, the *Packing Order* class in Booch's (1994) model is equivalent to the functions carried out by three classes in the Trusty Furniture Company's order processing system. Those three classes are the *Delivery Plan*, the *Despatch Instruction*, and the *Despatch Note* classes. All three of these classes are involved in various aspects of the functions performed by the *Packing Order* class such as the plan for delivery and the contents of the order. While they may not be key classes as such, the *Delivery Plan*, *Despatch Instruction*, and *Despatch Note* classes are all valid classes since they all share similarities with the *Packing Order* class in Booch's (1994) model.

The rest of the classes in Booch's (1994) model have no equivalent classes in this study. The reason for this is due to the fact that Booch (1994) also covers the re-stocking of products as well as the scheduling of people to fill the orders into account, and as such these functions are outside the specifications of this study. As a result of this, it is not possible to compare any of the other classes which exist in the Trusty Furniture Company's order processing system. Thus, as was stated earlier the validation has been limited to those classes which can be considered to be key classes. Since all of the key classes were valid, it can be assumed that overall the class diagrams developed in this study are valid.

8.9. Summary

This chapter has presented the results of this study and discussed the relevance of these results. It was found that both the structured and the object-oriented models increased in complexity to different degrees as a result of the addition of a payment facility to the order processing system.

Results from the structured version showed a similar increase in the level of overall complexity of both the dataflow diagram and the entity relationship diagram. The entity relationship diagram had a much greater increase than the whole object-oriented class diagram. This situation was further strengthened when the two models were compared on an individual basis. As a result of an outlier in the object-oriented model, the structured system proved to be more complex with an increase of 33.33 percent as opposed to 15.4 percent.

Further to the above findings, the object-oriented model obtained smaller increases in complexity overall, with the values ranging from increases of 8.2 percent to 23.5 percent for the system as a whole. This was in comparison to the structured system which showed a change of 33.33 percent in both diagrams. So, overall it would appear that the object-oriented system handled the changes to the system better, as it resulted in smaller increases in complexity.

Finally, the metrics and the models used in this study were validated in order to prove that they could produce consistent results in similar systems. It was found that overall the models provide a fair and reasonable view of the Trusty Furniture Company's order processing system.

CHAPTER 9: DISCUSSION

9.1. Introduction

This chapter discusses the results obtained from this study in terms of the models and metrics used. Some of the problems that were encountered in adopting the approach taken will be discussed as well as an attempt to relate the work to other work on metrics and maintenance. Finally, some general conclusions about this approach are discussed.

9.2. Depth of Model Development

One noticeable feature of the results in this study was the fact that the different models have been developed to different extents. It can be argued however that this is due to the functions of the different diagrams, such as the class diagram developed for the object-oriented version of the order processing system. When the class diagram is compared to the entity relationship diagram in the structured version of the same system, it appears as though the entity relationship diagram is not developed to the same level. However, this apparent disparity is due to the fact that some of the contents of the class diagram are also required to implement several of the processes which are present on the dataflow diagram in the structured system such as the validation of an order.

Thus, while there appears to be a difference in the extent to which the structured and object-oriented models have been developed, it can be argued that this is not the case. In order to compare the different systems it is necessary to compare both the dataflow diagram and the entity relationship diagram to the class diagram. When the two systems are compared in this context it is possible to see that the structured system has indeed been developed to a similar extent as the object-oriented system.

Because the individual models have been developed to different extents, there will obviously be an effect on the metrics when any changes are made to the models. The outcome of this problem is that the simpler models will result in a greater proportional change in complexity from a relatively small change to the model, such as adding a single entity, than those models which are considerably more complex.

A good example of this point is the entity relationship diagram which is relatively simple in comparison to the class diagram. In this particular case, the addition of the payment facility had a much greater effect on the complexity of the entity relationship diagram than was the case with the class diagram. This was because the entity relationship diagram had only a small number of entities and relationships present to begin with in comparison to the large number of objects and relationships that were present in the class diagram.

Hence, the addition of the payment facility had a greater proportional effect on the entity relationship model as a result of its size. Such a situation has the potential to distort the results, thus one possible way around this problem would be to weight any many-to-many relationships on the entity relationship diagram.

9.3. Weighting Many-to-Many Relationships

While the results obtained from the entity relationship diagram gave an indication of the level of complexity, the results did not take into account the effect that a many-to-many relationship may have on complexity in the future. One way around this problem would be to weight the relationship in order to reflect its influence on the future complexity of the entity relationship diagram.

Instead of counting a many-to-many relationship as one coupling it could be counted as two couplings. The reason for this is due to the inclusion of an intersection record later on in the development of the system. An intersection record is basically used to split a many-to-many relationship into two one-to-many or many-to-one relationships, thus doubling the number of couplings between the two entities involved in the many-to-many relationship. Since each coupling in the entity relationship diagram is counted from both entities point of view, the number of couplings overall would effectively be equal to four couplings instead of two couplings as is the case with any other relationship.

This weighting process would also help to resolve some of the problems that may arise due to the object-oriented system being more complex than the entity relationship diagram. The reason for this is that the weighting of any many-to-many relationships would result in a greater number of couplings in the entity relationship diagram. This means that the percentage change in complexity will decrease if a higher level of coupling is present in the diagram.

9.4. GQM

At this point, it is necessary to see whether the goals of this study have been met by the metrics which were used. One way of checking if the right metrics were used in order to meet the goals is to apply the GQM (Goal, Question, Metric) paradigm (Basili et al., 1988) to the study.

Basically, the GQM paradigm works by setting a goal, in this case to measure and compare the changing complexity of a specific model. Next, questions are asked about "how we are going to understand the extent of the problem and to assess the efficacy of the solution" (Sallis et al., 1995, p.183). And finally "these questions lead us to select or define and use one or more metrics appropriate to the goal" (Sallis et al., 1995, p.183).

It is important to state that the choice of metrics is largely "dependent on one's point of view" (Sallis et al., 1995, p.183). For example, a customer is likely to choose a different metric than a producer would because they have a different point of view about a product.

9.4.1 Goal.

As has been stated previously, the goal of this study has been to measure the relative changes in complexity of an object-oriented system and a structured system when subjected to changing requirements. There are questions which can help to understand the extent of the problem caused by the above goal.

9.4.2 Question.

The first question to be answered in this case is, *how is it possible to measure the complexity of a diagram so that it can be easily compared to another diagram developed using a different paradigm?*

The second question to be answered is, *can the complexity of a dataflow diagram be measured using existing measures designed to measure the complexity of structured program code?*

9.4.3 Metric.

As a result of the above goal and its questions, it is now possible to identify the metric(s) that can be used to measure the complexity of the two systems. One of the most important considerations was the fact that the metric had to be able to be applied to a diagram rather than to program code.

The metric needs to concentrate on the similarities between the different models since this will enable comparisons to be made on a similar basis. This definition would support the idea of using the *Coupling Between Objects* metric to measure the complexity in both the object-oriented system and the structured system's entity relationship diagram. The reason for this is that both the class diagram and the entity relationship diagram are being measured on the basis of the number relationships that are present other than inheritance relationships.

It can also be said that in order to measure the relative change in complexity between the different systems it is necessary to use a measure which is unaffected by the paradigm used to develop the system.

The best way of showing this relative change is to use a metric which shows the relative change as a percentage. Such a metric will work best if the diagrams are either of an equivalent size or if some method of weighting the smaller diagram is used. An example of such a weighting technique would be the weighting of the many-to-many relationships in the entity relationship diagram as discussed earlier.

While the above definitions answer the first question posed in the GQM paradigm, they do not really answer the second question in relation to the problem of measuring a dataflow diagram. In order to measure the dataflow diagram any metric must be able to measure the dataflow diagram on its bottom level in order to maximise the complexity.

Thus, McCabe and Schulmeyer's (1983) method of converting the dataflow diagram into a flowgraph solves the problem of *how do you apply a metric such as McCabe's (1976) cyclomatic complexity metric to a dataflow diagram when the metric was originally intended to measure program code*? since it concentrates on the bottom level of the dataflow diagram.

9.5. Alternative Metrics

While several different metrics were used in this study, the question of *what other metrics could have been used and how valid would their use be?* still remains.

9.5.1 Token counts.

One possible metric which could have been used is that of *Token Counts* as put forward by Levitin (1986). Tokens are "the basic syntactic units from which a program can be constructed.... Each token represents a sequence of characters which can be treated as a single logical entity.... Identifiers, strings, numbers, operators and separation symbols, such as commas and semicolons, are all typical tokens of programming languages" (Sallis et al., 1995, p.189).

As can be seen from the above definition, tokens were obviously designed to be used with program code. However, it is possible to apply the concept of tokens to a model such as an entity relationship diagram.

This process is done by describing the entities, relationships, relationship names, and cardinality of the model in a natural language context (Sallis et al., 1995). For example, consider the relationship *zero-to-many Customers place one-to-many Orders* on the entity relationship diagram. In this particular example the result is that of 5 tokens. This process of counting the tokens continues until all tokens in the model have been counted. The total number of tokens in a model is supposed to give an indication to the size of the code that will be produced.

While such a metric could have proved useful during the validation of models in terms of the size of each model, it is doubtful that this metric would have been very useful in measuring the complexity of the various models. The reason that *token counting* would not have been useful in measuring the relative complexity of the two systems is that it is designed to measure code size rather than complexity. Hence, it is not a true complexity metric.

It could be argued that the larger the code size of a program, the higher the complexity. However, the purpose of this study was to compare the relative complexity of two systems during the analysis stage rather than during the design stage since the results are less likely to be influenced by any design constraints that may be posed by the capabilities of a particular programming language or any other software or hardware related factors.

9.5.2 Information flow complexity.

Another alternative metric that could have been used in this study is that of *information flow complexity* as put forward by Henry and Kafura (1981). This metric is designed to measure the flow of information into and out of a procedure, module or other program unit (Sallis et al., 1995).

In order to measure the information flow complexity it is necessary to count the "flows into and out of a module" (Sallis et al., 1995, p.195). The number of inflows are called *fanin*, and the number of outflows are called *fanout*.

Once the *fanin* and *fanout* are measured they are used in a formula to calculate the *information flow complexity*. The formula used to calculate the *information flow complexity* is:

$(\text{fanin} * \text{fanout})^2$.

According to Sallis et al. (1995), "it is thought that modules with high values are more error prone than those with low values.... High information complexity measures may indicate a lack of cohesion (more than one function) or excessive functional complexity (which is similar) in the module" (p.195).

If such a metric were used in this study then it would be best suited to the structured system rather than the object-oriented system due to the fact that this metric appears to have been designed to measure the complexity of structured programs. Thus, it may be possible to apply the *information flow complexity* metric to the dataflow diagram. It is doubtful that the *information flow complexity* metric could be applied to the entity relationship model since it is not concerned with the actual processing of data.

In terms of validity, it is debatable as to whether the application of the *information flow complexity* metric to a diagram is valid. The reason for this is that the *information flow complexity* metric was clearly designed to measure the complexity of program code rather than the complexity of a diagram, and as such the validity of any results may be questionable.

One possible way around this problem would be to apply the *information flow complexity* metric to the dataflow diagram by counting the number of attributes in each of the dataflows that flows into or out of each process on the dataflow diagram. Thus, the ingoing dataflows would be used to determine *fanin*, and the outgoing dataflows would be used to determine *fanout*. By applying the *information flow complexity* metric to the dataflows it would be possible to calculate the *fanin* and *fanout* for each process.

9.6. Problems With the Study

There were several problems throughout the course of this study which resulted in changes being made in some way.

One of these changes was the reduction of the system to just the analysis rather than completing the whole process through to the generation of code for the systems. The main reason behind this change was caused by the inadequacy of the CASE tools used in terms of their code generation facilities.

Early on it was found that Rational Rose / C++ only generated skeleton code rather than the actual code. This means that it generates a template or an outline which needs to be filled in with the specific code for the system.

Later on it was found that Oracle CASE's code generation facilities were limited to the generation of embedded SQL (Structured Query Language), which was not really appropriate for what was initially intended. Thus, as a result of the above it was decided to limit the system and exclusively concentrate on the analysis results.

Another problem which resulted in a change was again associated with one of the CASE tools. In this case it was found that Oracle CASE did not support the symbols used in Yourdon's (1989) methodology. As a result of this, the system was modelled using Yourdon's (1989) methodology, however the symbols used on the dataflow diagram are those used in DeMarco's methodology.

9.7. Discussion of Results

As can be seen from this chapter, there have been a lot of issues that have affected the outcomes of this study.

While the results of the metrics show that overall the structured system is indeed subject to a greater change in complexity than its equivalent object-oriented system, the question that needs to be answered is *what do these results really show?*

The results of this study show that the type of development paradigm used will have an effect of the final system's complexity, and hence its future maintainability. Thus, the outcome of this study favours the use of object-oriented methodologies over structured methodologies if maintainability is one of the goals in the development of new systems.

Further, the results obtained here support the general idea put forward by authors such as Meyer (1981), that object-oriented methodologies are able to handle complexity better, and are therefore easier to maintain.

As was discussed earlier in this chapter, the results from each system were influenced by the initial complexity of the relevant models. Because of this factor the reliability of the results may be questioned since those models that are initially more complex will give relatively smaller increases compared to those models that were less complex.

The differences in the model are due to the fact that the functionality of the system is distributed across the diagrams in different ways. The result of this is that in the structured model each diagram only contains some aspects of the design, and therefore the structured diagrams appear to be less complex than the object-oriented diagrams.

These problems which lead to different increases in proportion lend support to the idea that the use of percentages may not be a good way to compare the changes between the object-oriented and structured systems. The reason for this is that their reliability is largely influenced by the complexity of the individual models. If all the individual models had a similar level of complexity then it could be argued that the use of such a method is reliable. However, this is not the case due to the functions that each individual diagram performs in the system as a whole.

In retrospect, the use of percentages to compare the relative increases in complexity may not be the best way to approach the problem of making comparisons between two completely different paradigms.

Any metric that is able to be used to make such comparisons needs to be able to take into account the unique differences of both the structured and object-oriented methodologies. An example of this point, is the fact that the class diagram in Booch's (1994) object-oriented methodology is performing many of the same functions and processes that are present in both a dataflow diagram and an entity relationship model in a structured methodology. Such a metric would need to combine the complexity results of both the dataflow diagram and the entity relationship model in order to make an equivalent, and therefore reliable comparison to the complexity of a class diagram in an object-oriented methodology.

9.8. Summary

This chapter has discussed a number of topics relating to the outcomes of this study. Among these topics was the issue of the depth that each system was developed to. It was argued that due to the functions of the various diagrams it was necessary to model the contents of the diagrams to different extents, even though the actual systems being modelled had an equivalent level of functionality.

Another topic that was discussed was the possibility of weighting some of the results obtained from the entity relationship diagram in order to take into account the future effects of many-to-many relationships on the complexity of the diagram.

It was found that it would be necessary to double the value of every many-to-many relationship so that the future effect on the complexity could be taken into account. As a result it would be possible to compare two diagrams that may be modelled to different extents.

An attempt was also made to apply the GQM paradigm to the metrics which were used in this study in order to see if they were the right metrics to meet the goals of this study. It was found that these metrics were indeed suitable for meeting the goals and answering the resulting questions.

There was also a discussion on several alternative metrics which could have been used and their validity in relation to this study. It was argued that while the use of *token counts* would have been possible in both systems, their validity would be questionable due to the fact that they are designed to measure code size rather than the complexity of analysis diagrams.

In the case of the *information flow complexity* metric, it could have been used in the structured system. However, the results obtained from the *information flow complexity* metric would be subject to debate since it was originally intended to measure the complexity of structured program code rather than the complexity of structured analysis diagrams. It was also found that the *information flow complexity* metric could not be applied to the object-oriented diagram because the metric appears to have been designed to measure the complexity of structured programs and therefore it will not take some of the special features of the object model into account.

Some of the problems that were encountered throughout the study were discussed as well as the resulting changes to the study.

Finally, the results were discussed in terms of what they show and how reliable these results are. Some discussion was also focused on whether the idea of comparing two different paradigms on a percentage basis was flawed. It was found that overall, there needs to be a better way of comparing the changes in complexity between object-oriented and structured methodologies.

CHAPTER 10: CONCLUSIONS AND FUTURE WORK

10.1. Conclusions

The results of this study tend to suggest that overall, information systems that are developed using object-oriented methodologies, such as Booch (1994), do not increase in relative complexity as much as similar systems which are developed using a structured methodology such as Yourdon (1989).

While the results for the structured model showed an increase of 33.33 percent for both the dataflow diagram and the entity relationship diagram, none of the metrics used to measure the object-oriented system showed as great an increase in complexity. In comparison, the metrics used to measure the object-oriented model showed increases ranging from 8.2 percent to 23.5 percent. This difference in the range of increases is further evidence that the complexity of the object-oriented system was not affected by the changes as much as the structured system was.

The results obtained from the *Coupling Between Objects* metric when it was applied to both the entity relationship diagram and an equivalent class diagram showed that despite the effect of the class *Customer*, both systems increased their complexity by the same proportions in terms of the number of couplings that each class/entity had after the changes to the system were made.

The outcome of the validation process indicated that the metrics used in this study were capable of giving consistent results in similar systems. In addition, the validation process also showed that both the structured and object-oriented model developed for this study were fair representations of the Trusty Furniture Company's order processing system.

In terms of the approach used in the measurement of the models in this study, it was found that the use of percentages to compare systems that were developed using different paradigms had some drawbacks due to the effect that the initial complexity of each model had on the resulting change in complexity. This is because the addition of a function to a smaller model would result in a greater change in proportion than would be the case with a larger model.

In general the results of this study show that the type of development paradigm used will have a direct effect on the system's complexity and hence its future maintainability. Thus, the outcome of this study favours the use of object-oriented methodologies over structured methodologies if maintainability is one of the goals in the development of new systems.

Thus, it can be argued that the hypothesis has been proven since the results of this study suggest that the object-oriented version of the system had a relatively smaller increase in complexity overall, and therefore it should be easier to maintain.

10.2. Future Work

A number of possibilities exist for future work that could extend the results obtained in this particular study.

One such extension would be to conduct a similar study that measured the effects of different changes to an object-oriented system. Such a study would help to determine whether certain changes to an object-oriented system affected the level of complexity more than other changes.

Another possibility for future work in this area might be to measure the complexity of several systems developed using different object-oriented techniques. Such a study would help to show which different object-oriented methodologies resulted in a more complex system. This study would also allow for easy comparison between the models as they would all be object-oriented, and therefore it would be easier to apply the MOOSE metric set.

The major problems encountered during this study arose from the unavailability of a suitable tool which could be used to measure the complexity of different systems and which could cope with the different constructs and formulations of the models. The most important need that is indicated from this research is to investigate ways of making complexity measurements that could be applied to a wide variety of systems. Such a tool would allow future studies to make meaningful comparisons between systems that were designed using either structured or object-oriented methodologies, and bring some objectivity to debates regarding the relative merits of the different methodologies.

APPENDIX A: Outputs from Structured Model Before Changes

Appendix A contains the following structured output for the Trusty Furniture Company's Order Processing System:

- One Statement of Purpose
- One Event List
- One set of Dataflow Diagrams (4 DFD's)
- One Entity Relationship Diagram
- One set of Event Response Diagrams (7 ERD's)
- Selected Data Dictionary Output

Trusty Furniture Company Order Processing System

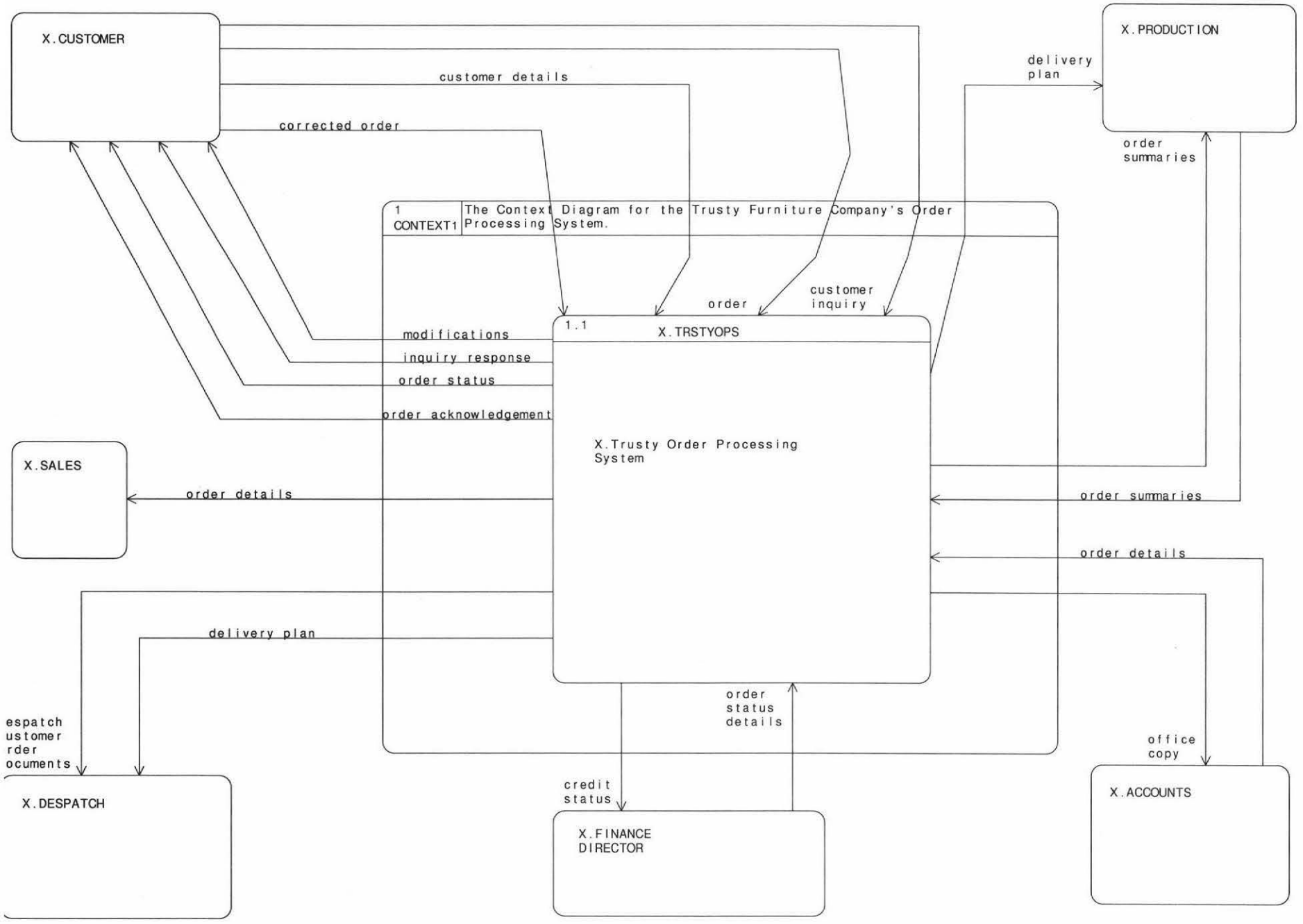
Statement of Purpose

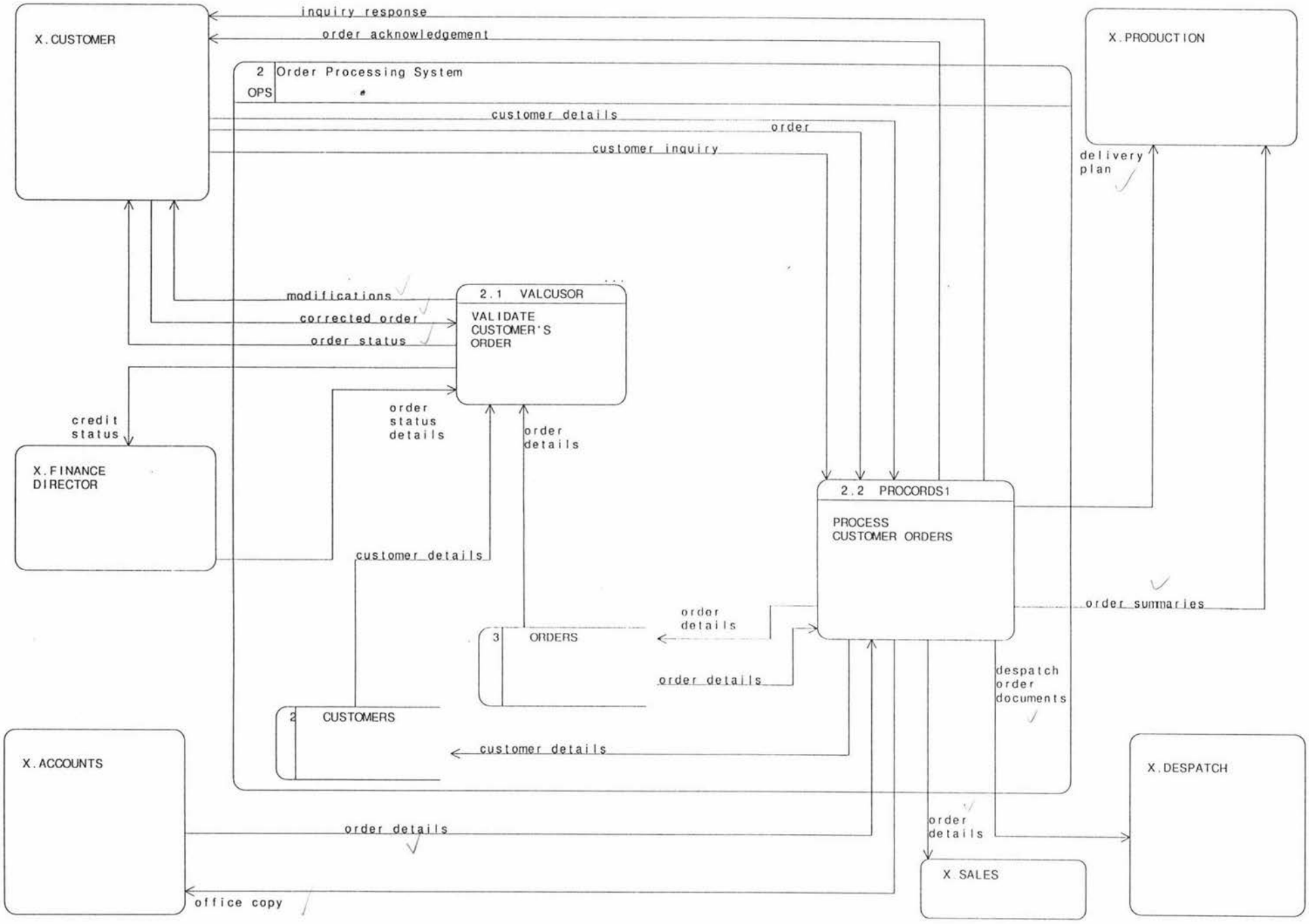
The purpose of the Trusty Furniture Company's order processing system is to take orders from customers, check those orders to see if they are valid, and process the orders while coordinating the ordering process and the delivery of the orders through the production of delivery plans. The order processing system is also used to provide financial and order information to interested parties both within and outside the organisation.

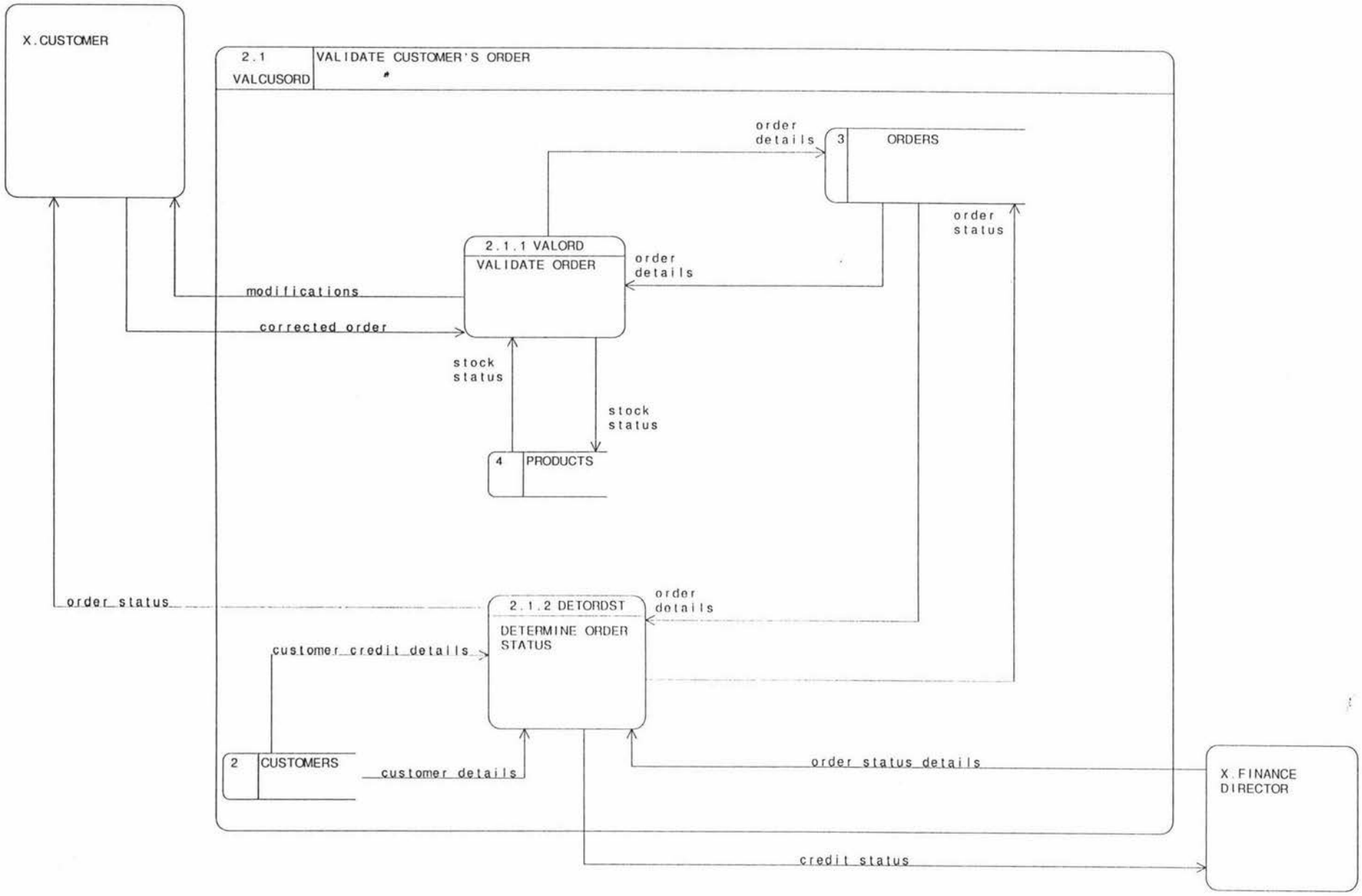
Trusty Furniture Company Order Processing System

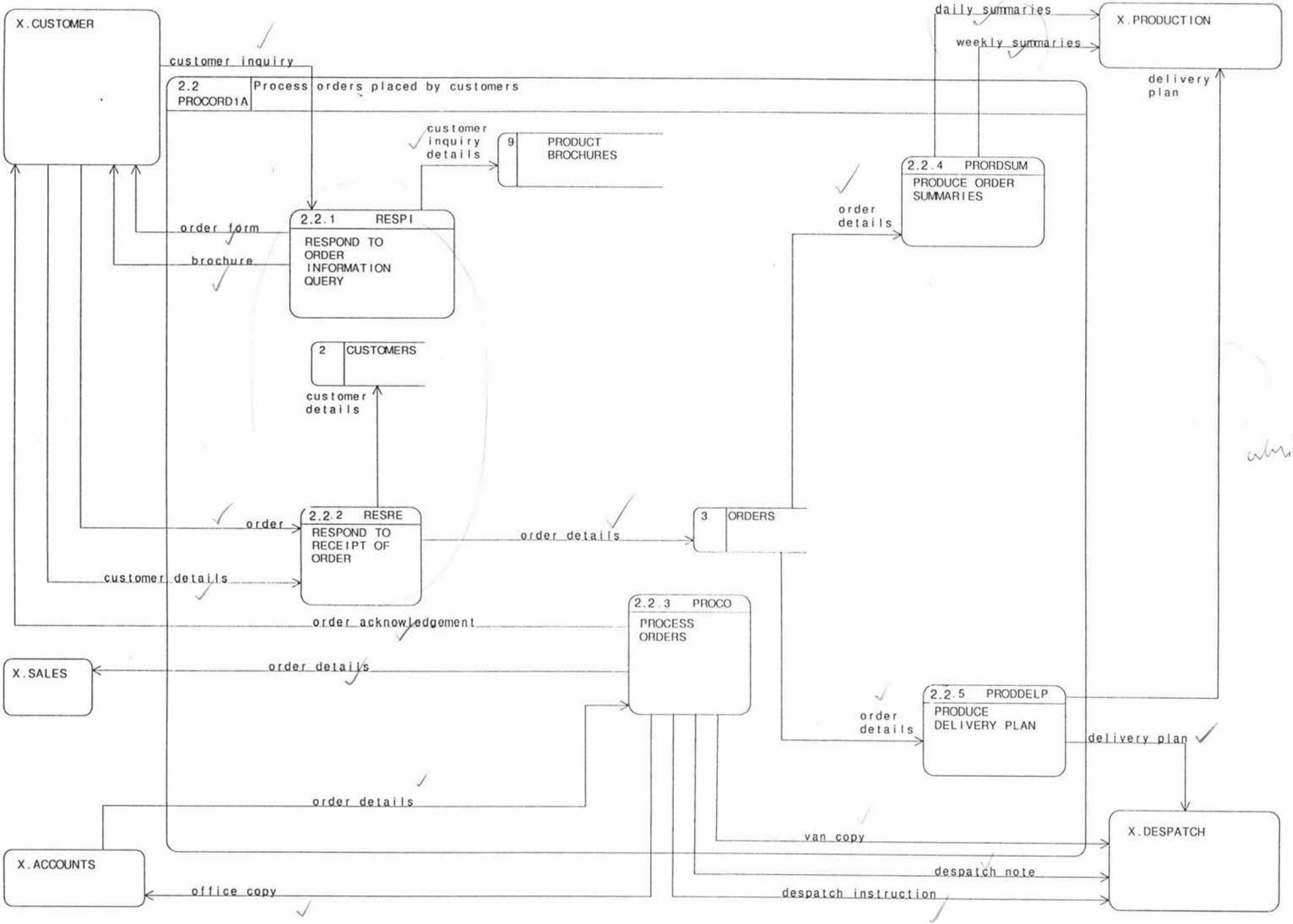
Event List

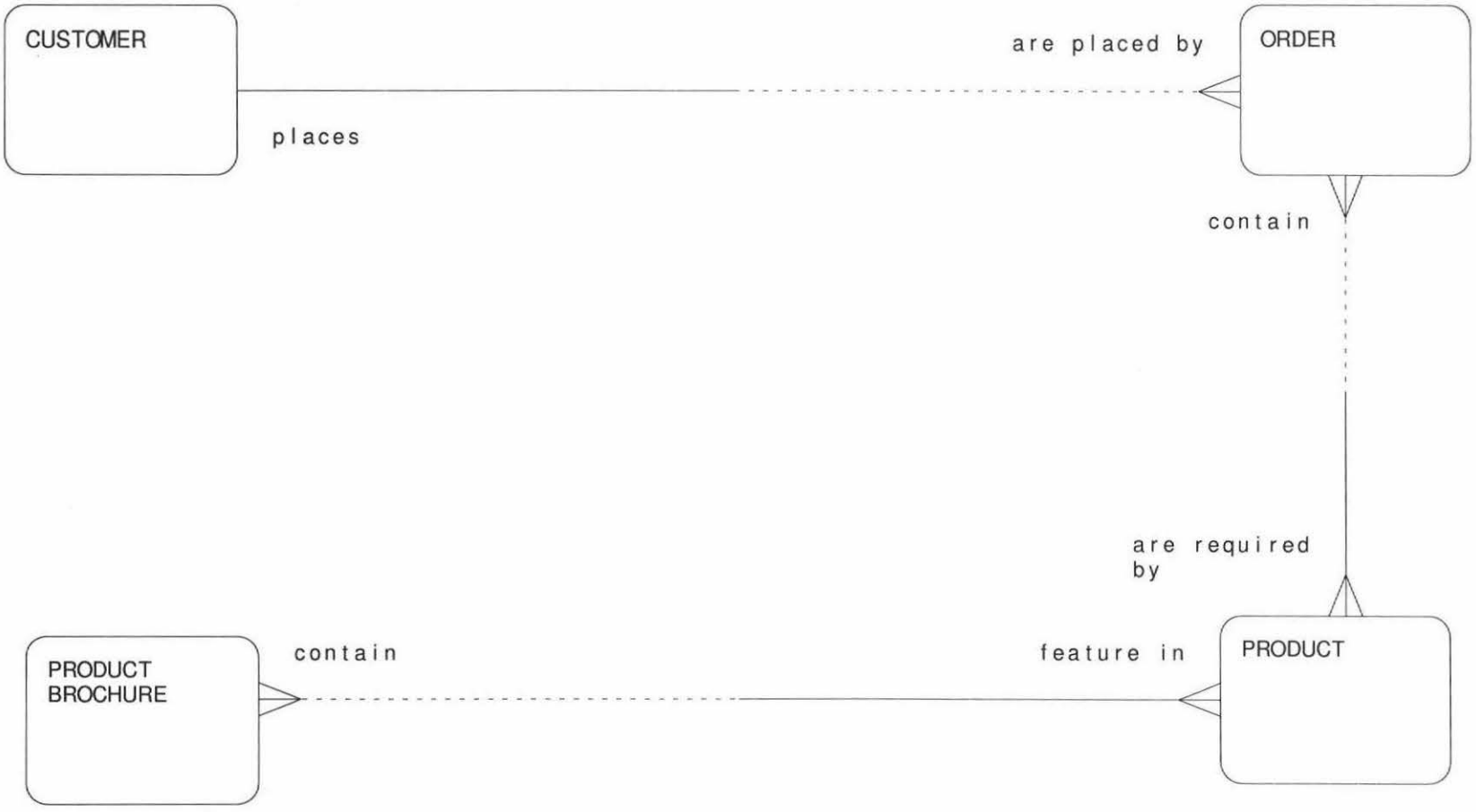
1. Customer requests information
2. Customer places order
3. Customer modifies invalid order
4. Finance Director determines order status
5. Copies of order arrive from Accounts
6. Production needs order summaries
7. Production needs delivery plan


X.CUSTOMER
X.PRODUCTION
delivery plan
customer details
corrected order
order summaries
1 CONTEXT1
The Context Diagram for the Trusty Furniture Company's Order Processing System.
order
customer inquiry
1.1
X.TRSTYOPS
modifications
inquiry response
order status
order acknowledgement
X.Trusty Order Processing System
X.SALES
order details
order summaries
order details
delivery plan
order status details
espatch ustomer rder ocuments
office copy
credit status
X.DESPATCH
X.FINANCE DIRECTOR
X.ACCOUNTS


X.CUSTOMER
inquiry response
order acknowledgement
2 OPS Order Processing System
customer details
order
customer inquiry
X.PRODUCTION
delivery plan
modifications
corrected order
order status
2.1 VALCUSOR
VALIDATE CUSTOMER'S ORDER
credit status
order status details
order details
X.FINANCE DIRECTOR
2.2 PROCORDS1
PROCESS CUSTOMER ORDERS
customer details
order summaries
order details
3 ORDERS
order details
despatch order documents
2 CUSTOMERS
customer details
X.ACCOUNTS
order details
order details
X.SALES
X.DESPATCH
office copy

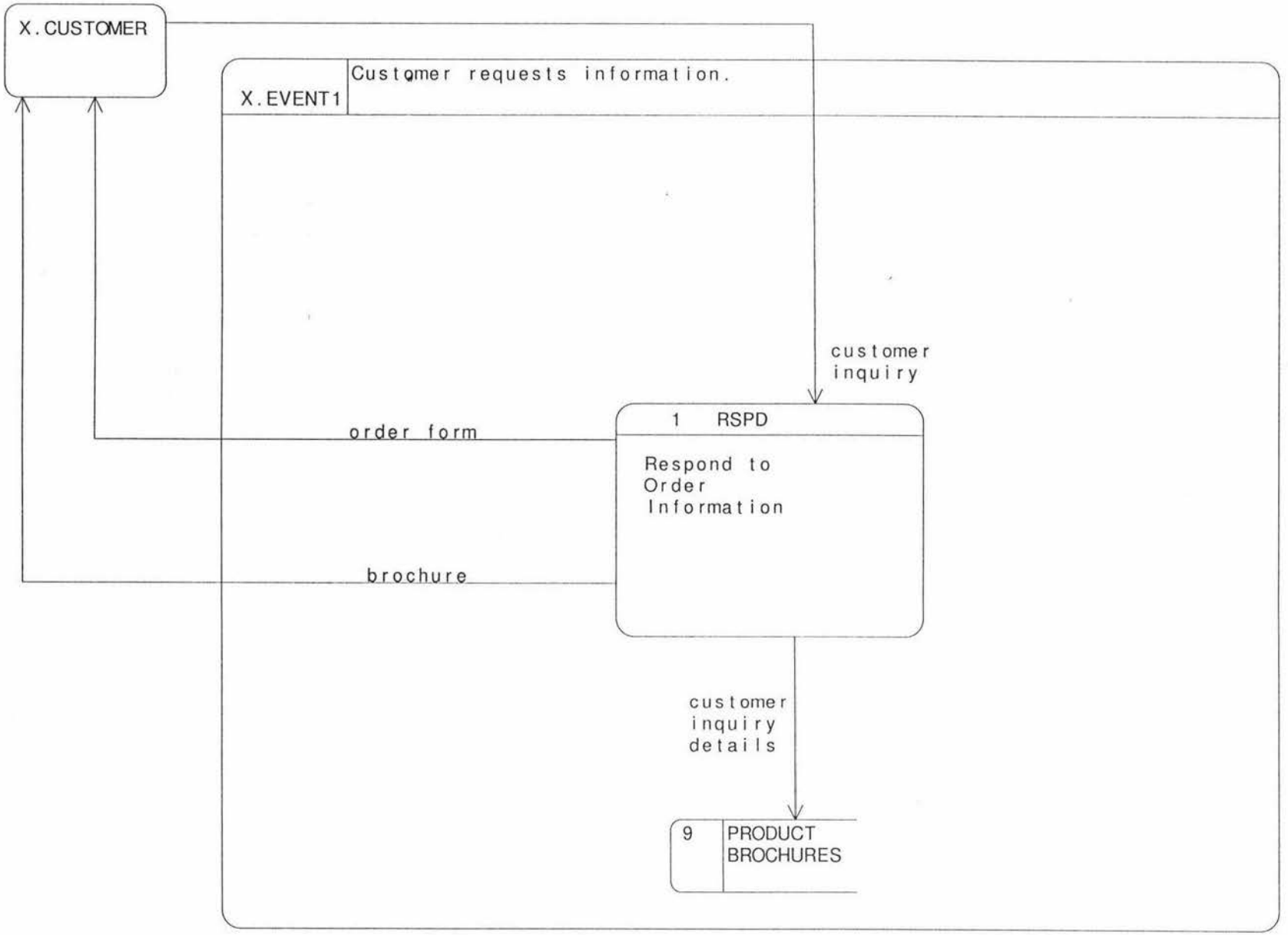

X.CUSTOMER
2.1
VALCUSORD
VALIDATE CUSTOMER'S ORDER
order details
3
ORDERS
order status
2.1.1 VALORD
VALIDATE ORDER
order details
modifications
corrected order
stock status
stock status
4
PRODUCTS
order status
2.1.2 DETORDST
DETERMINE ORDER STATUS
order details
customer credit details
2
CUSTOMERS
customer details
order status details
X.FINANCE DIRECTOR
credit status

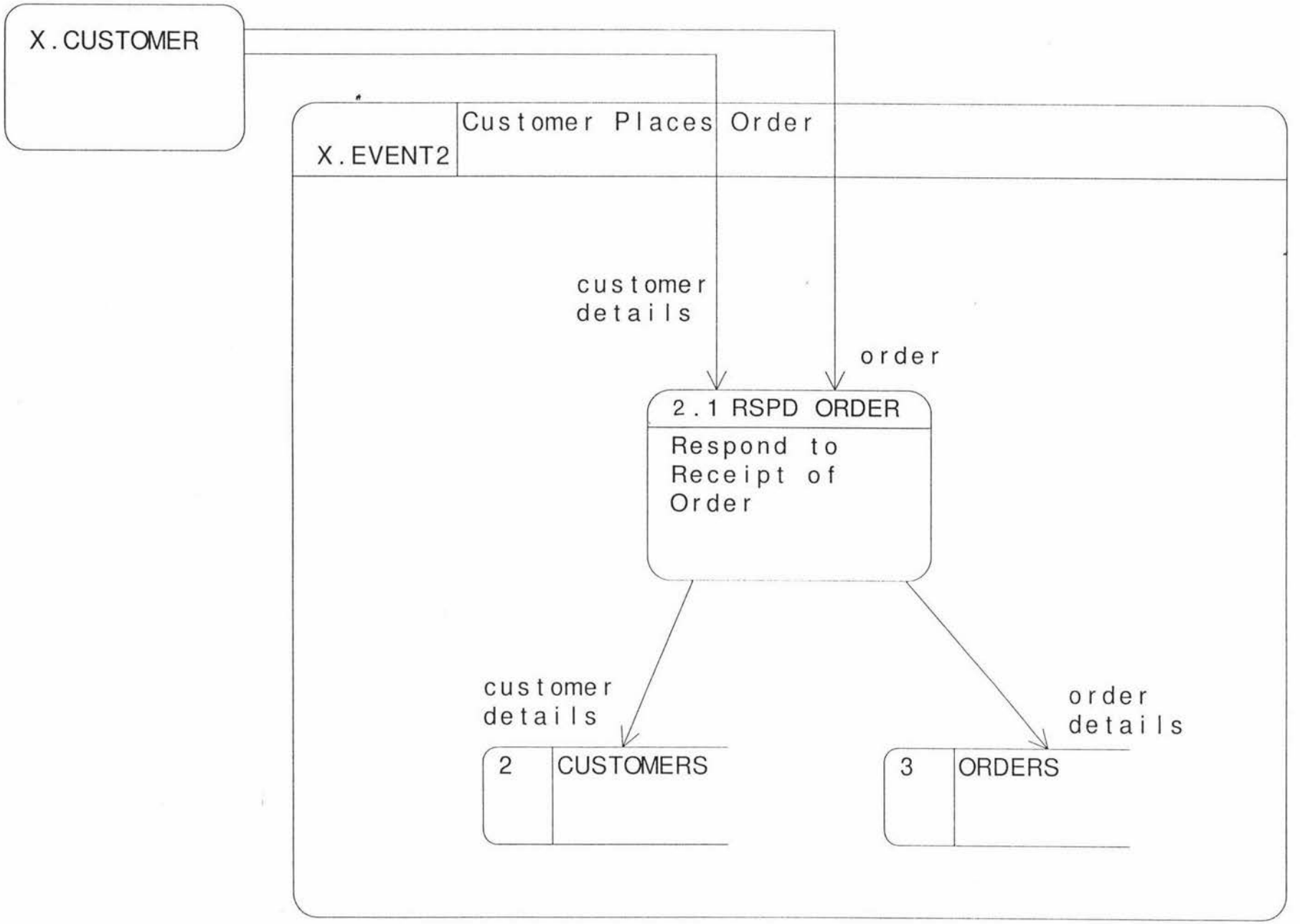

X.CUSTOMER
customer inquiry
2.2 PROCORD1A
Process orders placed by customers
daily summaries
weekly summaries
X.PRODUCTION
delivery plan
customer inquiry details
9 PRODUCT BROCHURES
2.2.4 PRORDSUM
PRODUCE ORDER SUMMARIES
order details
order form
brochure
2.2.1 RESPI
RESPOND TO ORDER INFORMATION QUERY
2 CUSTOMERS
customer details
order
2.2.2 RESRE
RESPOND TO RECEIPT OF ORDER
order details
3 ORDERS
customer details
2.2.3 PROCO
PROCESS ORDERS
order acknowledgement
X.SALES
order details
2.2.5 PRODDELP
PRODUCE DELIVERY PLAN
order details
delivery plan
order details
van copy
X.DESPATCH
despatch note
X.ACCOUNTS
office copy
despatch instruction

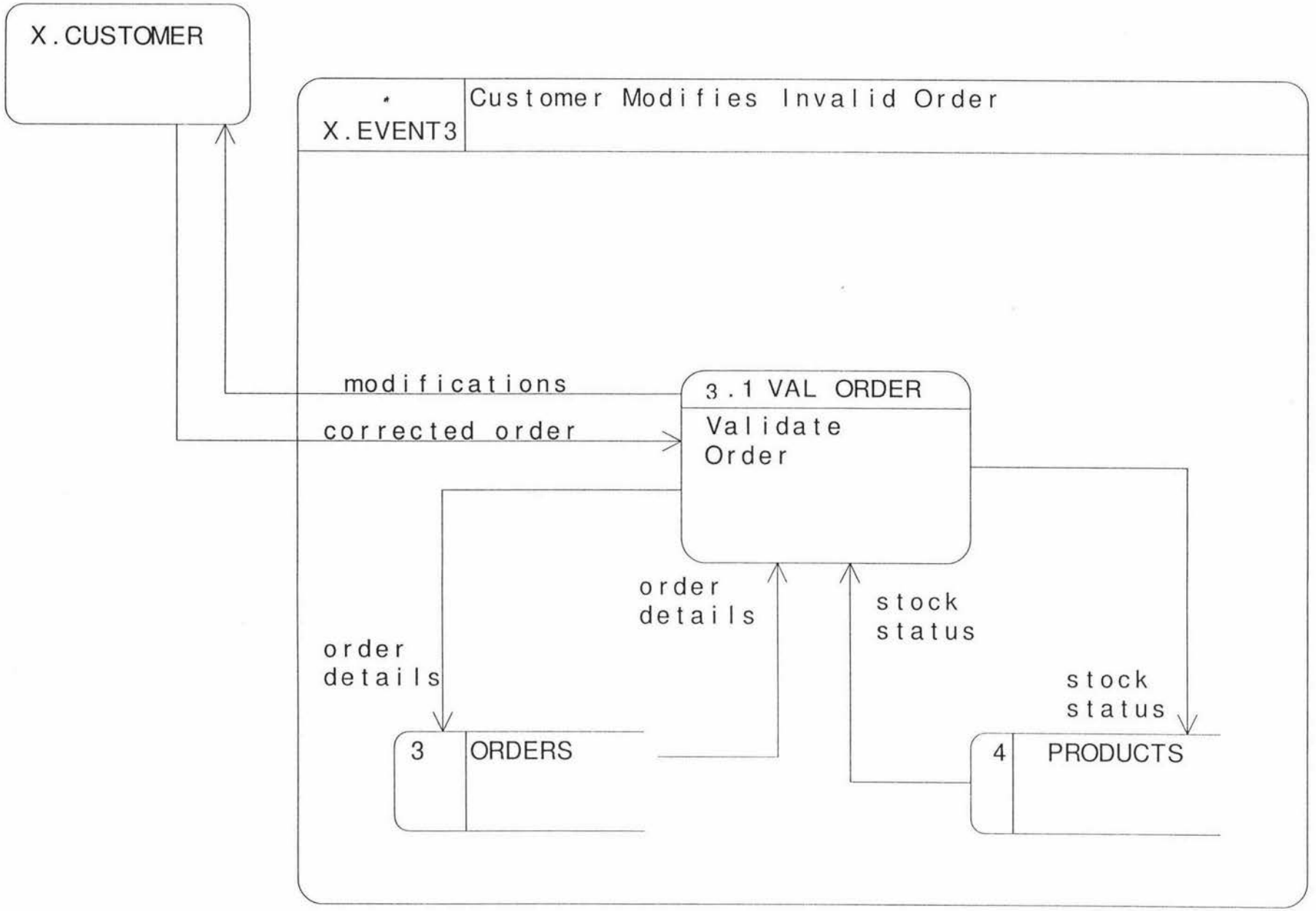


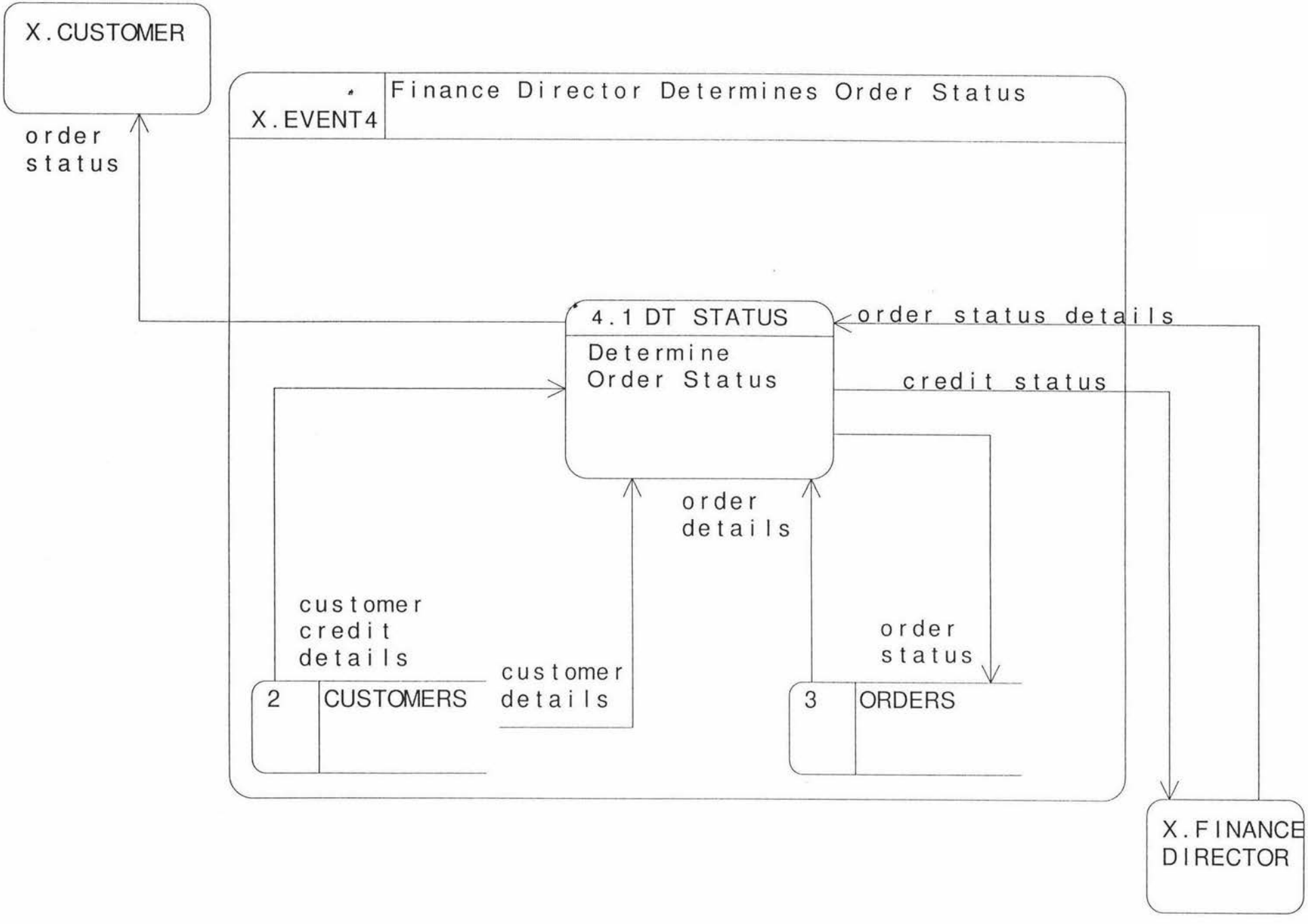
ENTITY RELATIONSHIP DIAGRAM OF TRUSTY FURNITURE COMPANY'S ORDER PROCESSING SYSTEM.

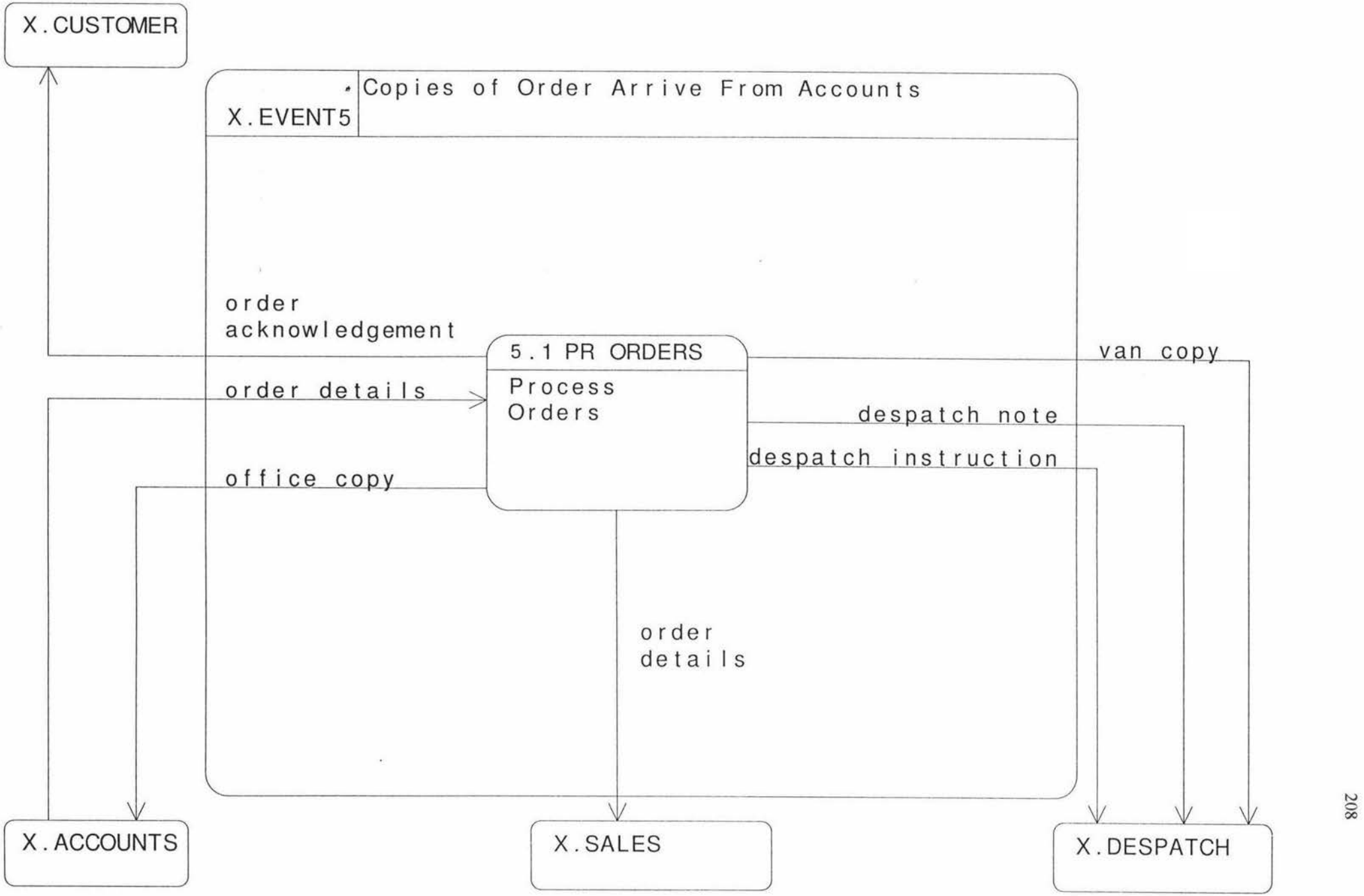
(Original Version)

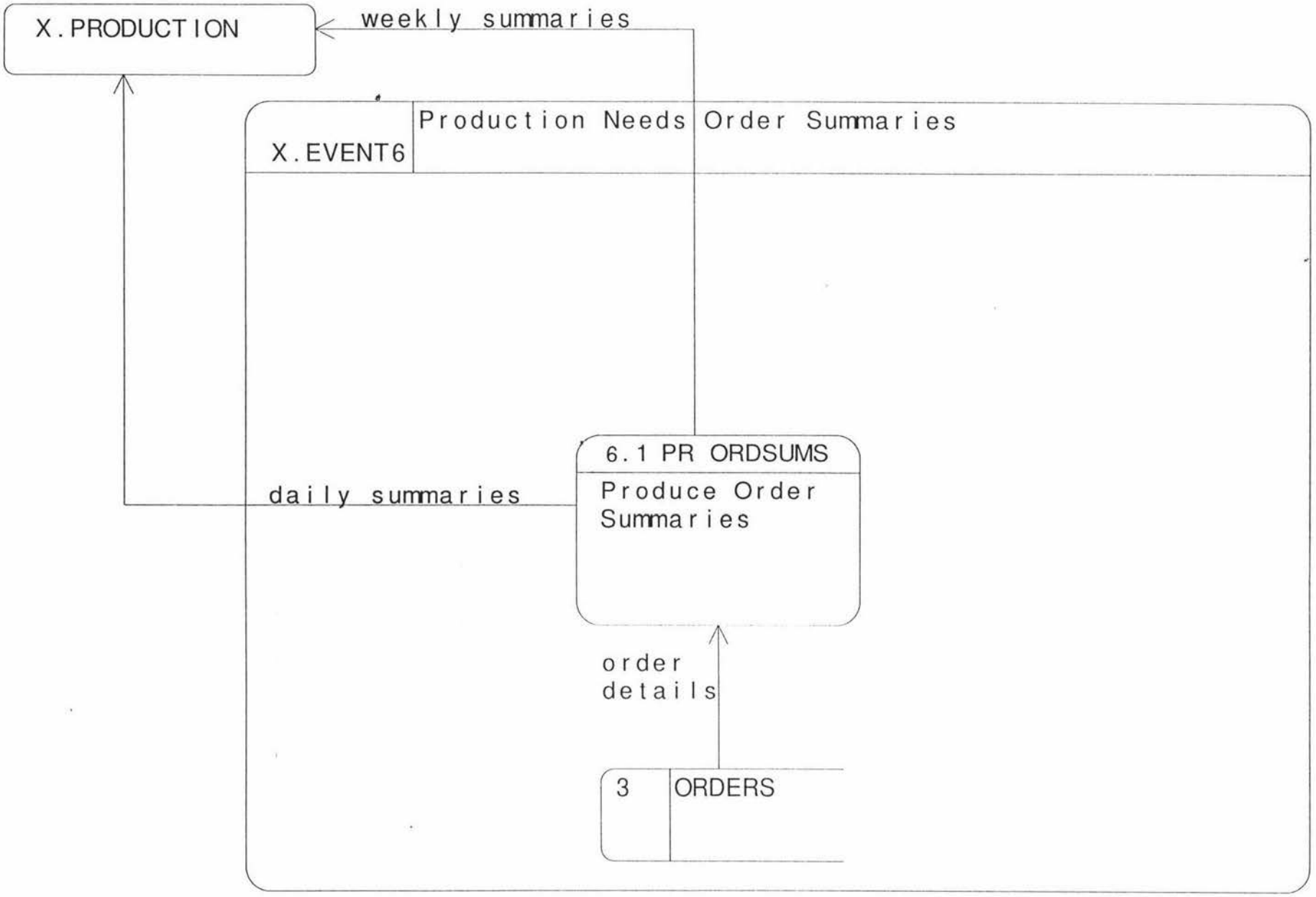

X.CUSTOMER
X.EVENT1
Customer requests information.
customer inquiry
1 RSPD
Respond to Order Information
order form
brochure
customer inquiry details
9 PRODUCT BROCHURES

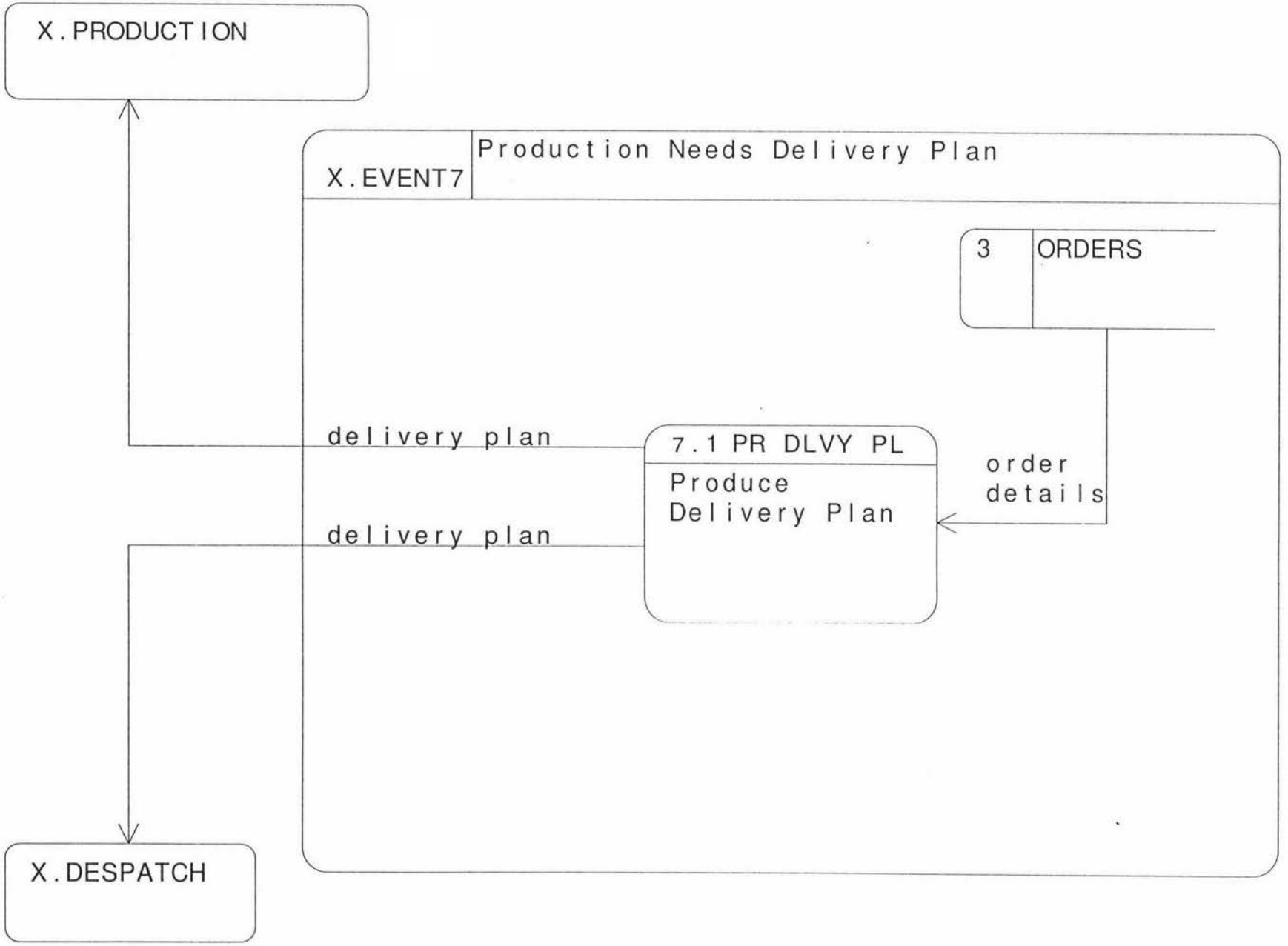

X.CUSTOMER
X.EVENT2
Customer Places Order
customer details
order
2.1 RSPD ORDER
Respond to Receipt of Order
customer details
order details
2
CUSTOMERS
3
ORDERS


X.CUSTOMER
X.EVENT3
Customer Modifies Invalid Order
modifications
corrected order
3.1 VAL ORDER
Validate
Order
order
details
order
details
stock
status
stock
status
3
ORDERS
4
PRODUCTS


X.CUSTOMER
order
status
X.EVENT4
Finance Director Determines Order Status
4.1 DT STATUS
Determine
Order Status
order status details
credit status
order
details
customer
credit
details
customer
details
order
status
2
CUSTOMERS
3
ORDERS
X.FINANCE
DIRECTOR


X.CUSTOMER
X.EVENT5
Copies of Order Arrive From Accounts
order
acknowledgement
5.1 PR ORDERS
Process
Orders
van copy
order details
despatch note
despatch instruction
office copy
order
details
X.ACCOUNTS
X.SALES
X.DESPATCH


X.PRODUCTION
weekly summaries
X.EVENT6
Production Needs Order Summaries
6.1 PR ORDSUMS
Produce Order Summaries
daily summaries
order details
3
ORDERS


X.PRODUCTION
X.EVENT7
Production Needs Delivery Plan
3
ORDERS
delivery plan
7.1 PR DLVY PL
Produce
Delivery Plan
order
details
delivery plan
X.DESPATCH

CUSTOMER

Which has significance as :
The entity customer is a person or organisation that wishes to make a purchase from the Trusty Furniture Company. The Customer entity is use d to store the customer's personal details such as name and address.

Information includes customer city address,
customer first name,
customer home phone number,
customer last name,
customer number,
customer street address,
customer suburb address,
customer title,
customer work phone number, etc.

Each must be places one or more ORDERS

ORDER

Which has significance as :
An order is a set of products that the customer wishes to purchase fro m the Trusty Furniture Company. New Customers must place an order usin g the required order form. Existing Customers are allowed to place an order by phone or by order form. The entity order is used to store inf ormation about each customers order.

Information includes customer number,
order amount,
order date,
order description,
order number,
order quantity,
product type number, etc.

Each may be contain one or more PRODUCTS
and may be are placed by one and only one CUSTOMER

PRODUCT

Which has significance as :
A product is an item produced by the Trusty Furniture Company for sale to its customers. The entity product is used by the system to store i nformation abott the stock levels of different products.

Information includes product description,
product price,
product type number,
quantity on hand, etc.

Each must be feature in one or more PRODUCT BROCHURES
and must be are required by one or more ORDERS

PRODUCT BROCHURE

Which has significance as :
Product Brochure is used top store information about products availabl e for sale from th
e Trusty Furniture Company. Different brochures contain specific produ cts such as chairs. However one brochure contains all the products ava ilable from the Trusty Furniture Company.

Information includes brochure description,
brochure number,
product type number, etc.

Each may be contain one or more PRODUCTS

APPENDIX B: Outputs from Structured Model After Changes

Appendix B contains the following structured output for the Trusty Furniture Company's Order Processing System:

- One Statement of Purpose
- One Event List
- One set of Dataflow Diagrams (4 DFD's)
- One Entity Relationship Diagram
- One set of Event Response Diagrams (8 ERD's)
- Selected Data Dictionary Output

Trusty Furniture Company Order Processing System

Statement of Purpose

(*for the modified system*)

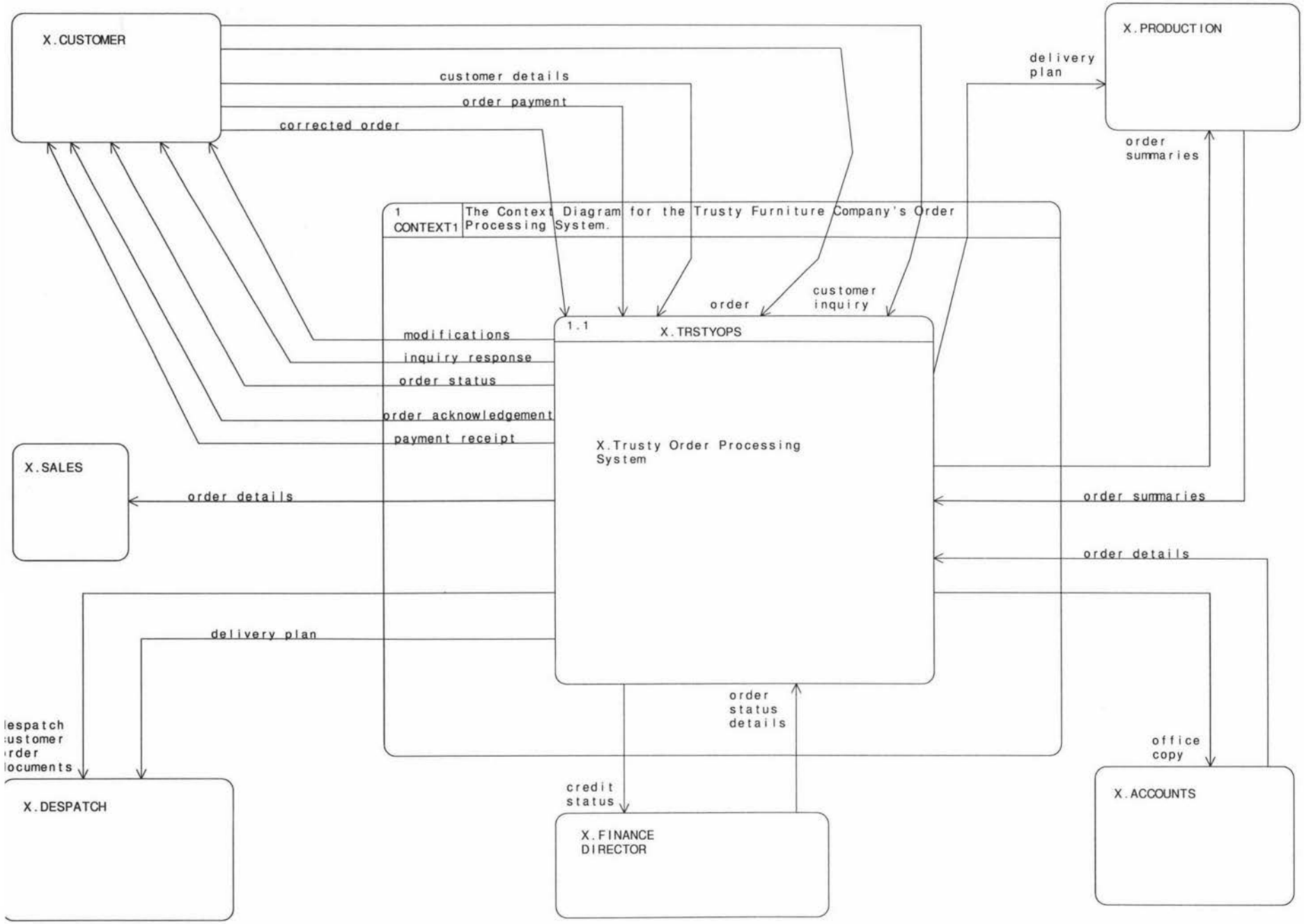
The purpose of the Trusty Furniture Company's order processing system is to take orders from customers, check those orders to see if they are valid, process customer payments and orders while coordinating the ordering process and the delivery of the orders through the production of delivery plans. The order processing system is also used to provide financial and order information to interested parties both within and outside the organisation.

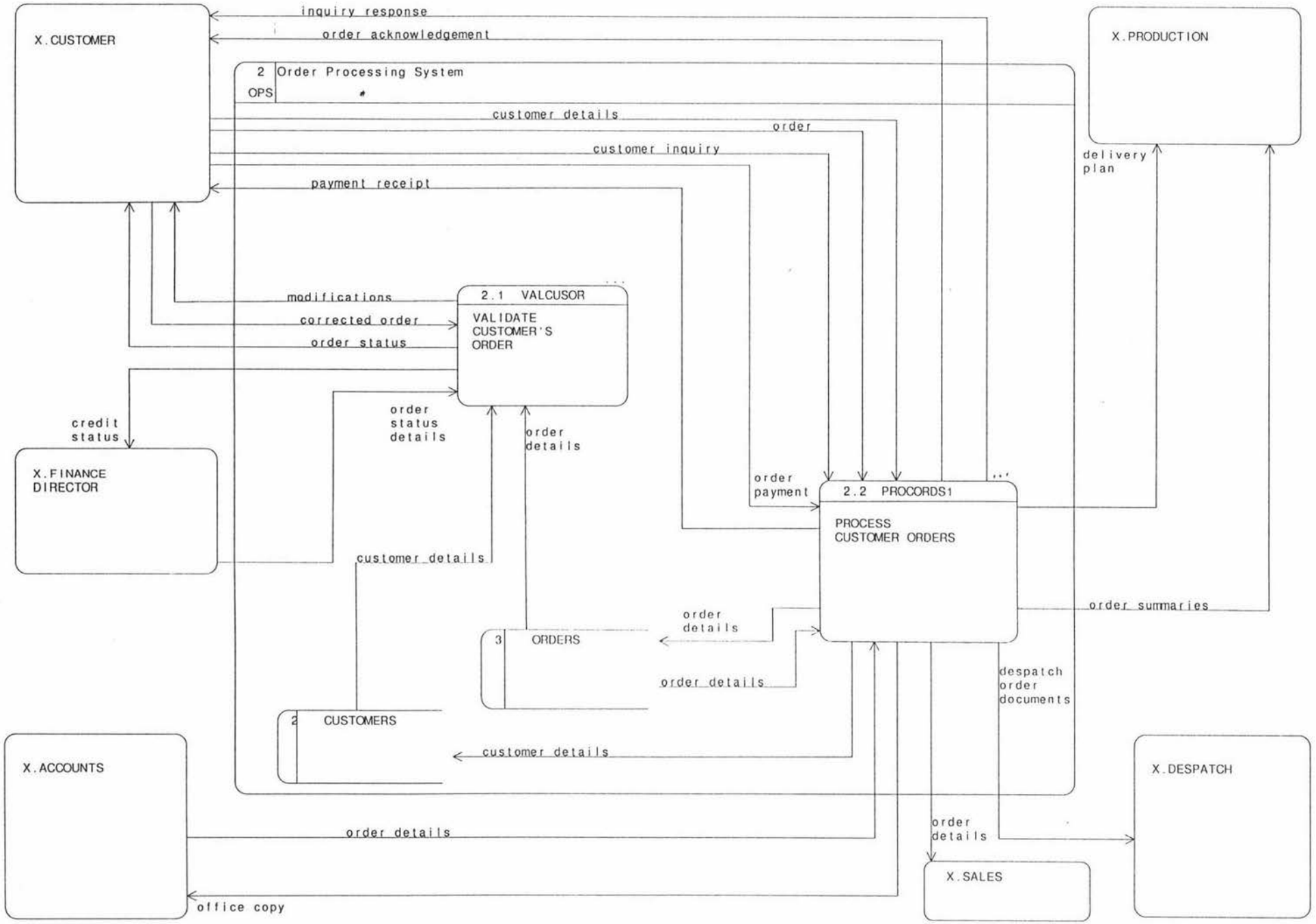
Trusty Furniture Company Order Processing System

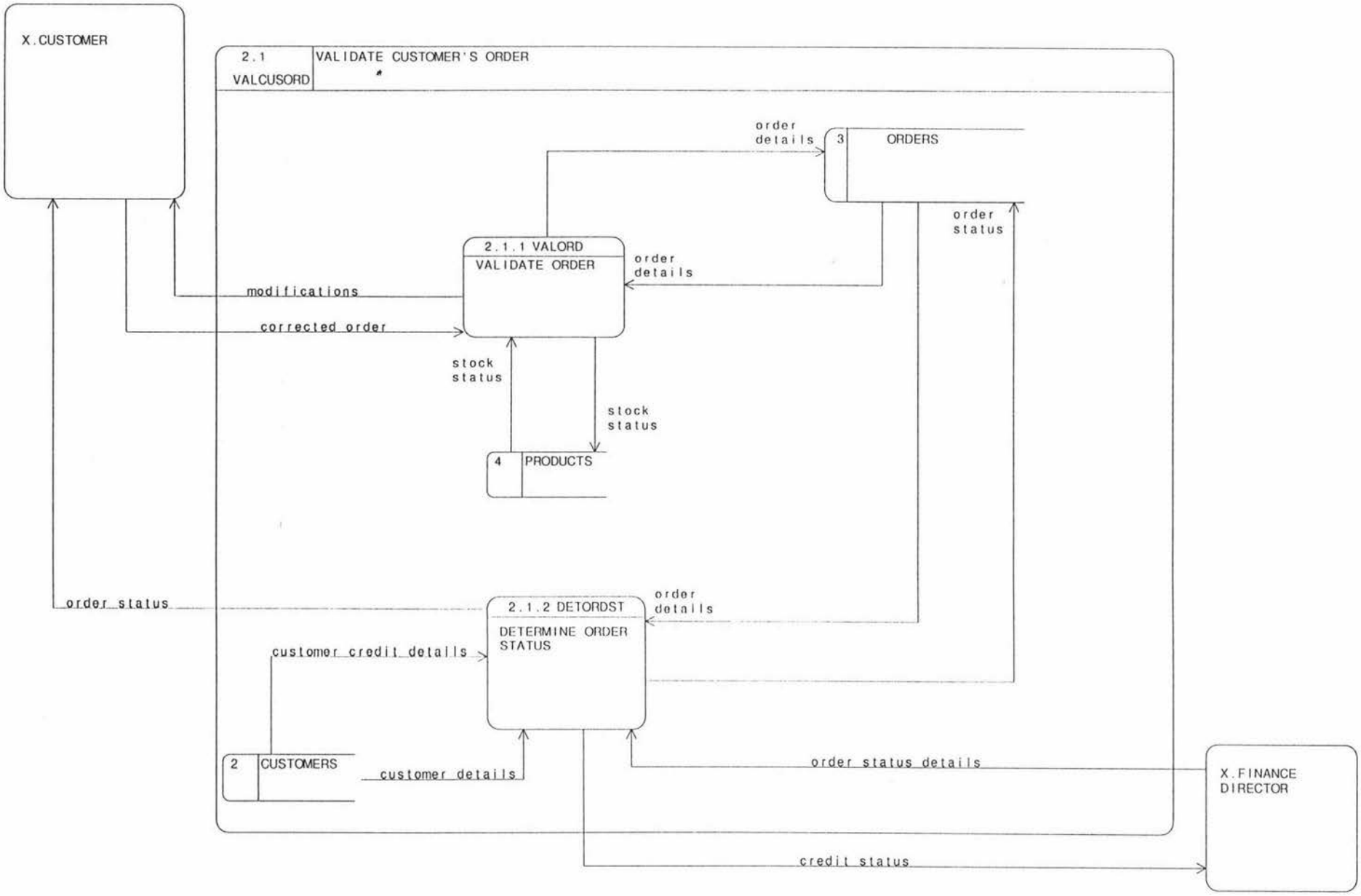
Event List

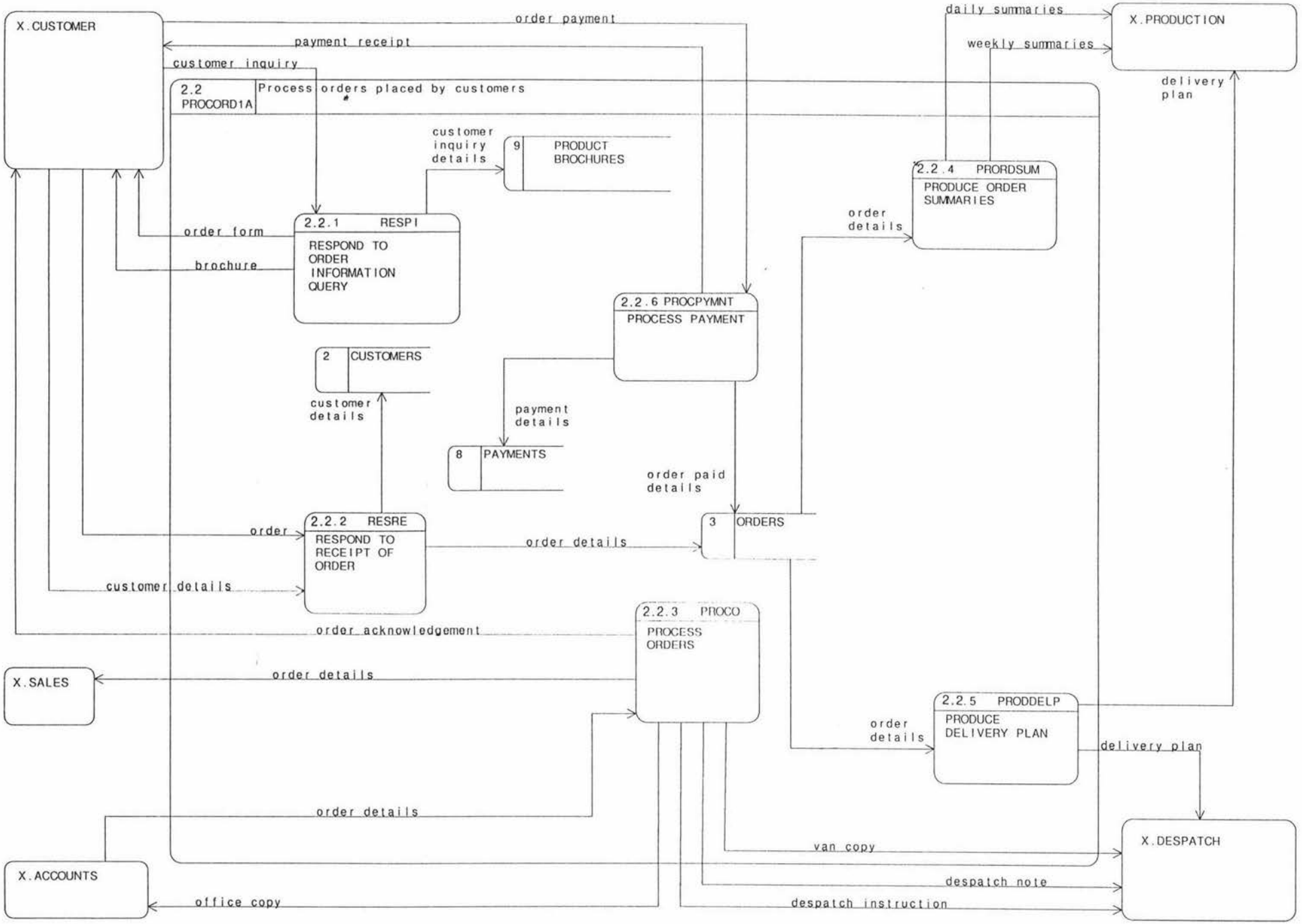
(*for the modified system*)

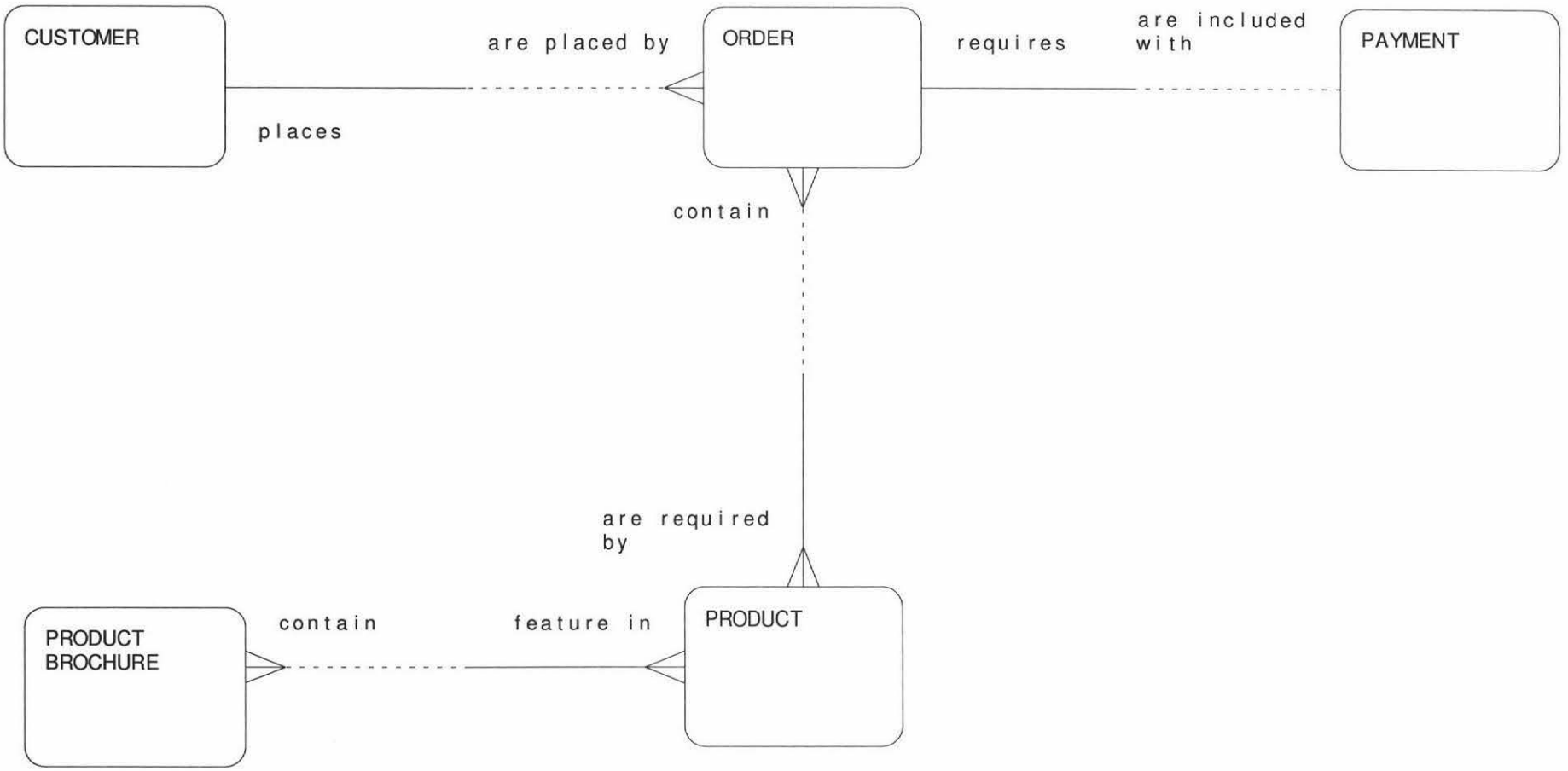
1. Customer requests information
2. Customer places order
3. Customer pays for order at time of purchase
4. Customer modifies invalid order
5. Finance Director determines order status
6. Copies of order arrive from Accounts
7. Production needs order summaries
8. Production needs delivery plan


X.CUSTOMER
X.PRODUCTION
delivery plan
customer details
order payment
corrected order
order summaries
1 CONTEXT1
The Context Diagram for the Trusty Furniture Company's Order Processing System.
order
customer inquiry
1.1
X.TRSTYOPS
modifications
inquiry response
order status
order acknowledgement
payment receipt
X.Trusty Order Processing System
X.SALES
order details
order summaries
order details
delivery plan
order status details
espatch
ustomer
rder
ocuments
office copy
credit status
X.DESPATCH
X.FINANCE DIRECTOR
X.ACCOUNTS


X.CUSTOMER
inquiry response
order acknowledgement
2 OPS Order Processing System
customer details
order
customer inquiry
payment receipt
X.PRODUCTION
delivery plan
modifications
corrected order
order status
2.1 VALCUSOR
VALIDATE CUSTOMER'S ORDER
credit status
X.FINANCE DIRECTOR
order status details
order details
order payment
2.2 PROCORDS1
PROCESS CUSTOMER ORDERS
customer details
order details
order summaries
3 ORDERS
order details
despatch order documents
2 CUSTOMERS
customer details
X.ACCOUNTS
order details
order details
X.SALES
X.DESPATCH
office copy

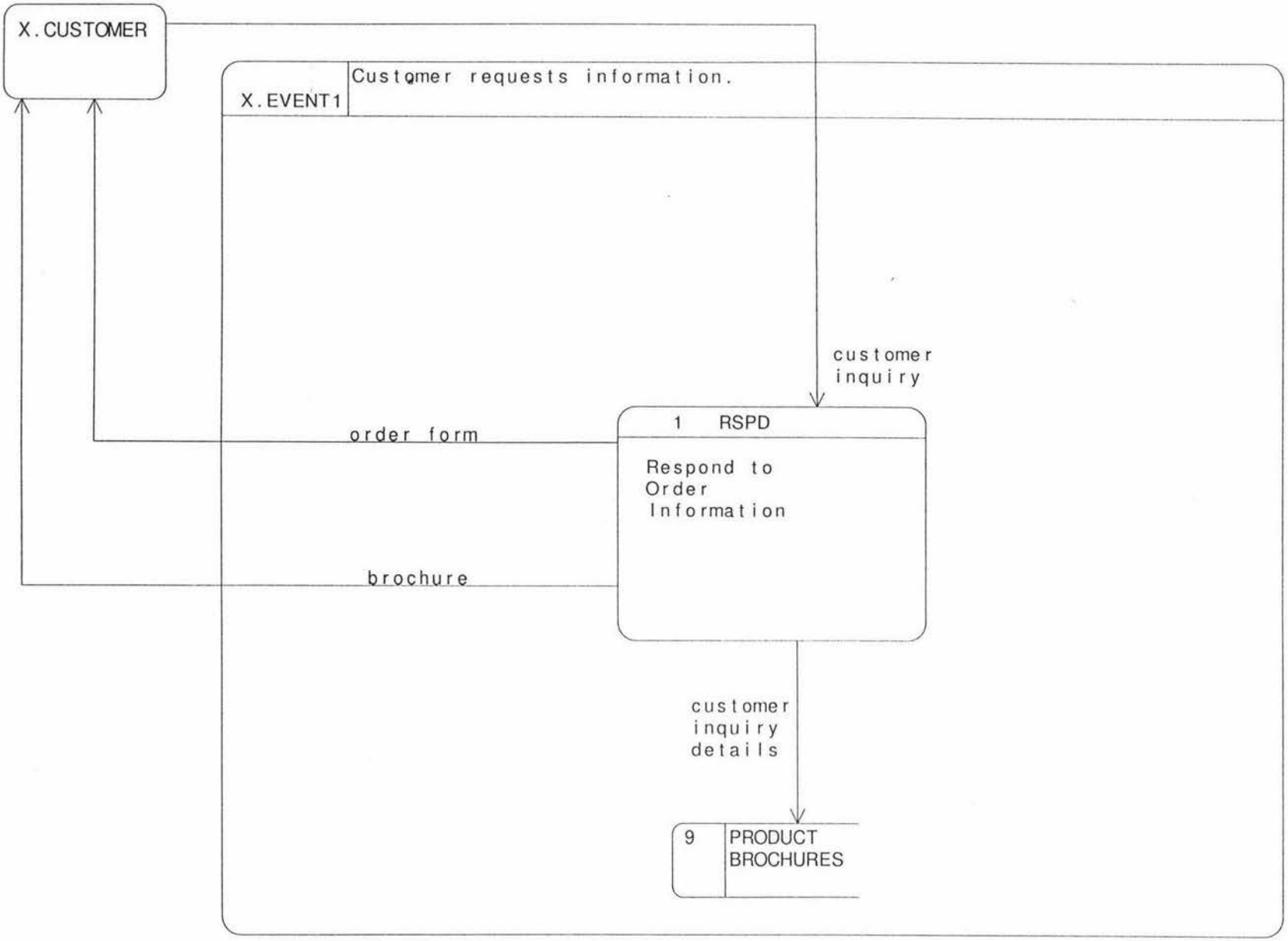

X.CUSTOMER
2.1
VALCUSORD
VALIDATE CUSTOMER'S ORDER
order details
3
ORDERS
order status
2.1.1 VALORD
VALIDATE ORDER
order details
modifications
corrected order
stock status
stock status
4
PRODUCTS
order status
2.1.2 DETORDST
DETERMINE ORDER STATUS
order details
customer credit details
2
CUSTOMERS
customer details
order status details
X.FINANCE DIRECTOR
credit status

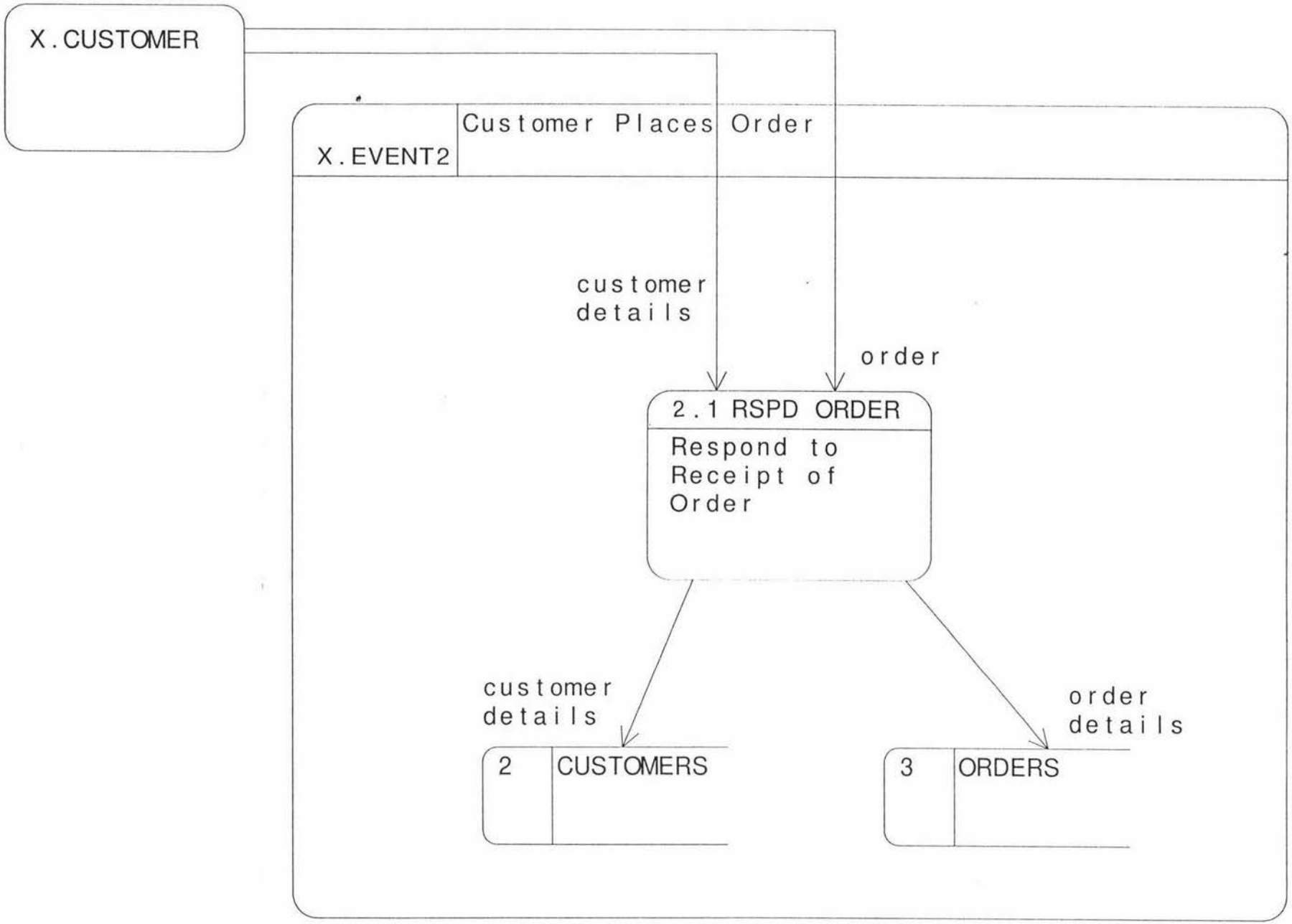

X.CUSTOMER
order payment
payment receipt
customer inquiry
daily summaries
weekly summaries
X.PRODUCTION
delivery plan
2.2 PROCORD1A
Process orders placed by customers
customer inquiry details
9 PRODUCT BROCHURES
2.2.4 PRORDSUM
PRODUCE ORDER SUMMARIES
order details
order form
brochure
2.2.1 RESPI
RESPOND TO ORDER INFORMATION QUERY
2.2.6 PROCPYMNT
PROCESS PAYMENT
2 CUSTOMERS
customer details
payment details
8 PAYMENTS
order paid details
order
2.2.2 RESRE
RESPOND TO RECEIPT OF ORDER
order details
3 ORDERS
customer details
order acknowledgement
2.2.3 PROCO
PROCESS ORDERS
X.SALES
order details
2.2.5 PRODDELP
PRODUCE DELIVERY PLAN
order details
delivery plan
order details
van copy
X.DESPATCH
X.ACCOUNTS
despatch note
office copy
despatch instruction

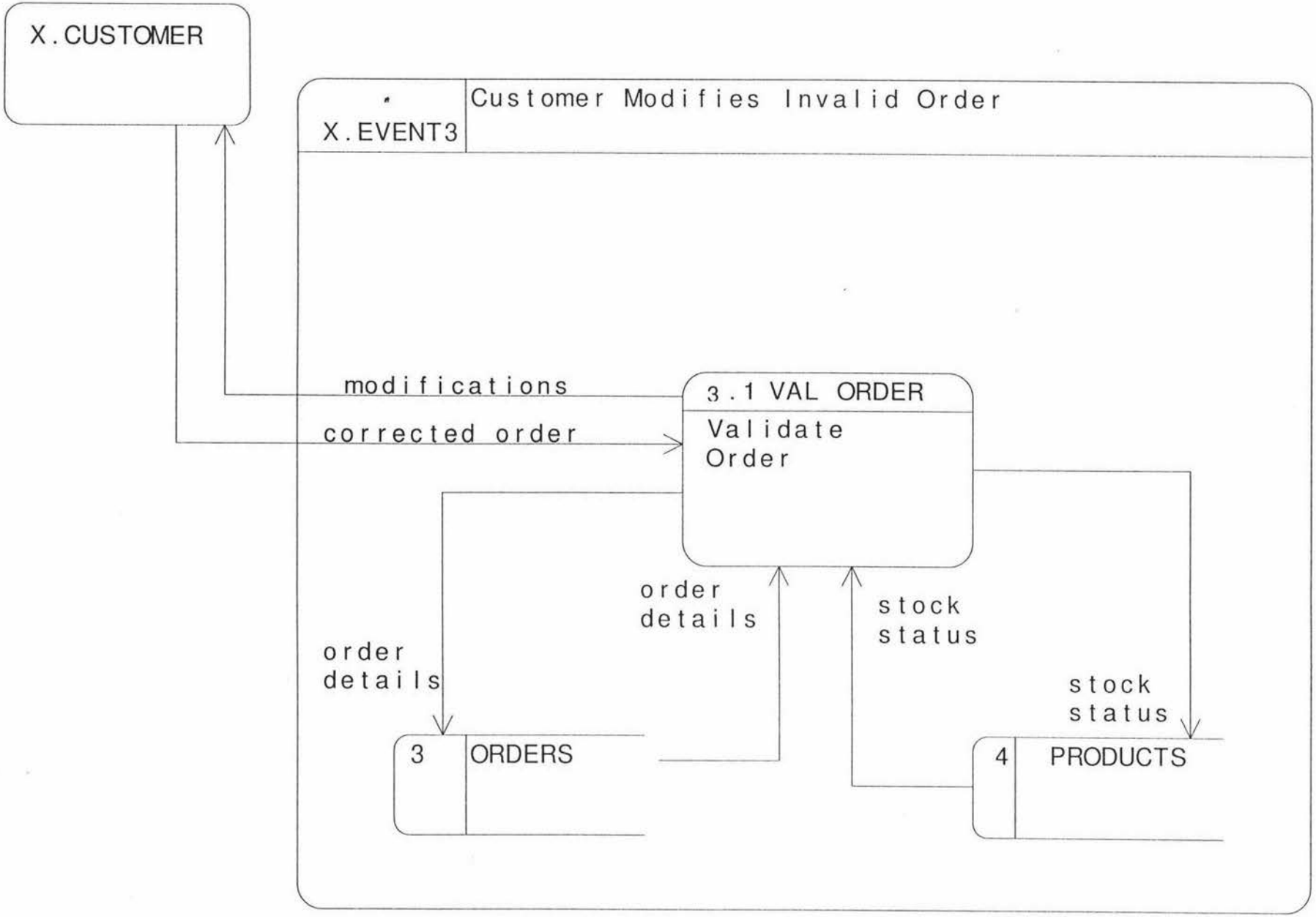


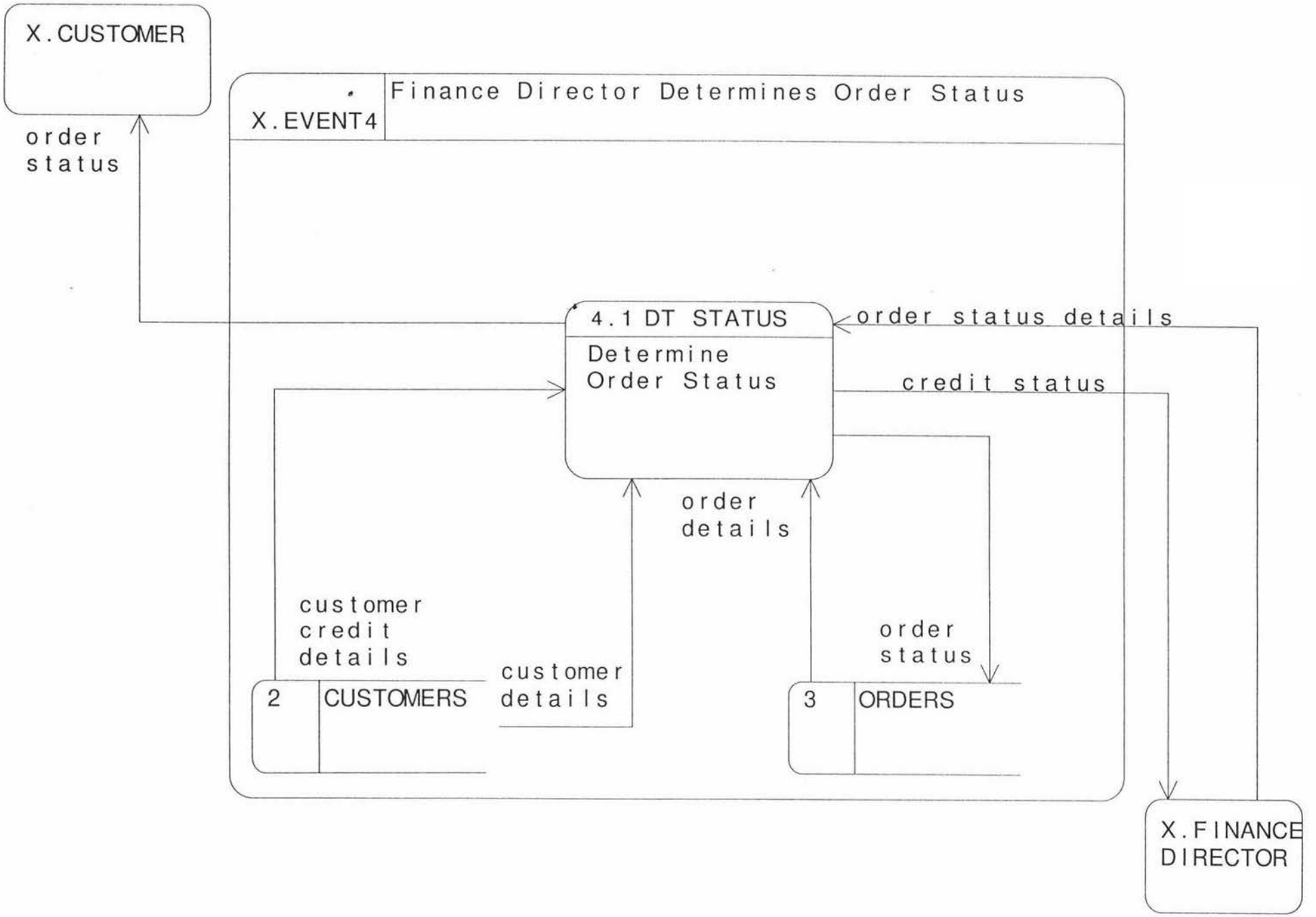
ENTITY RELATIONSHIP DIAGRAM OF TRUSTY FURNITURE COMPANY'S ORDER PROCESSING SYSTEM.

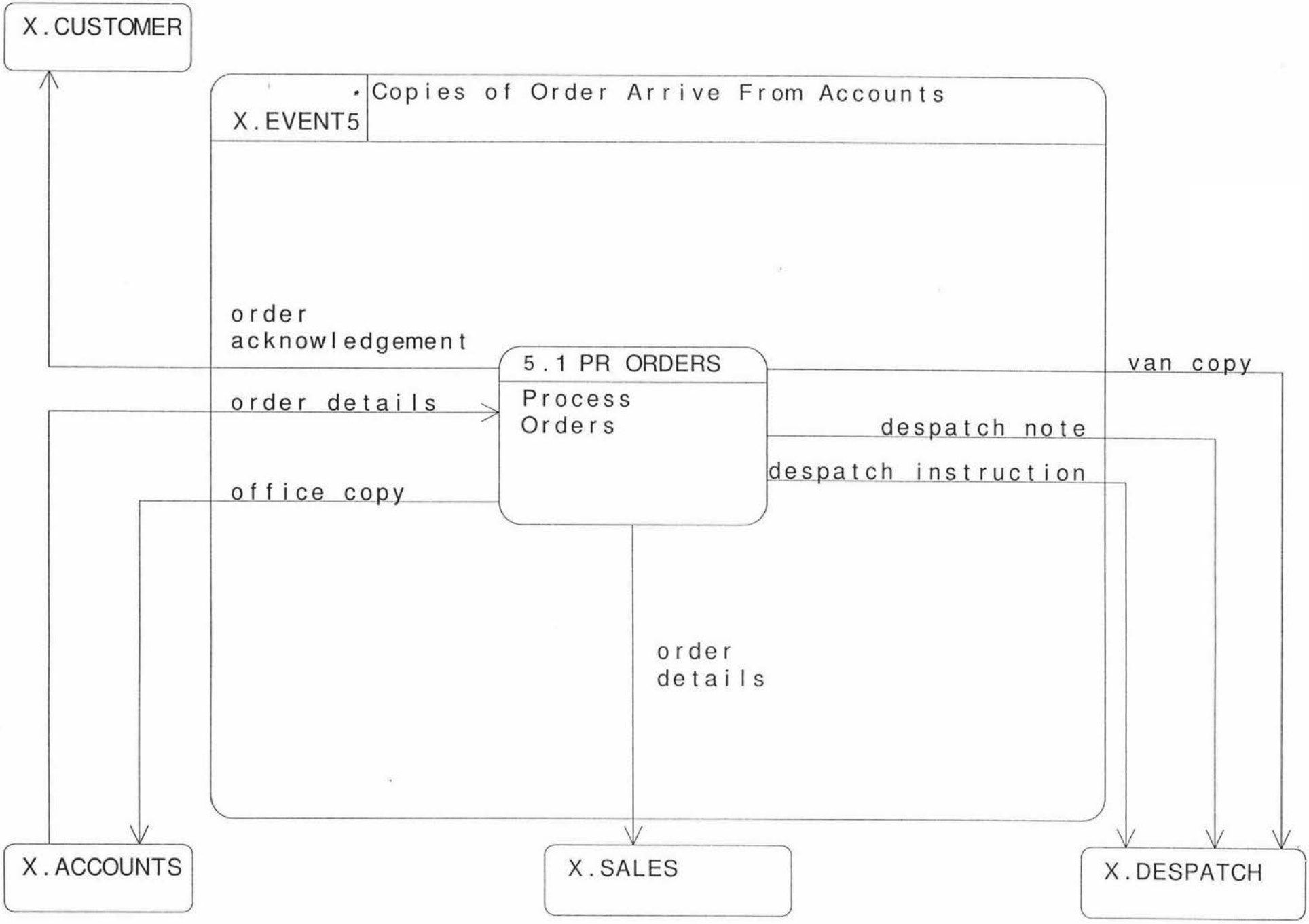
(Changed Version)

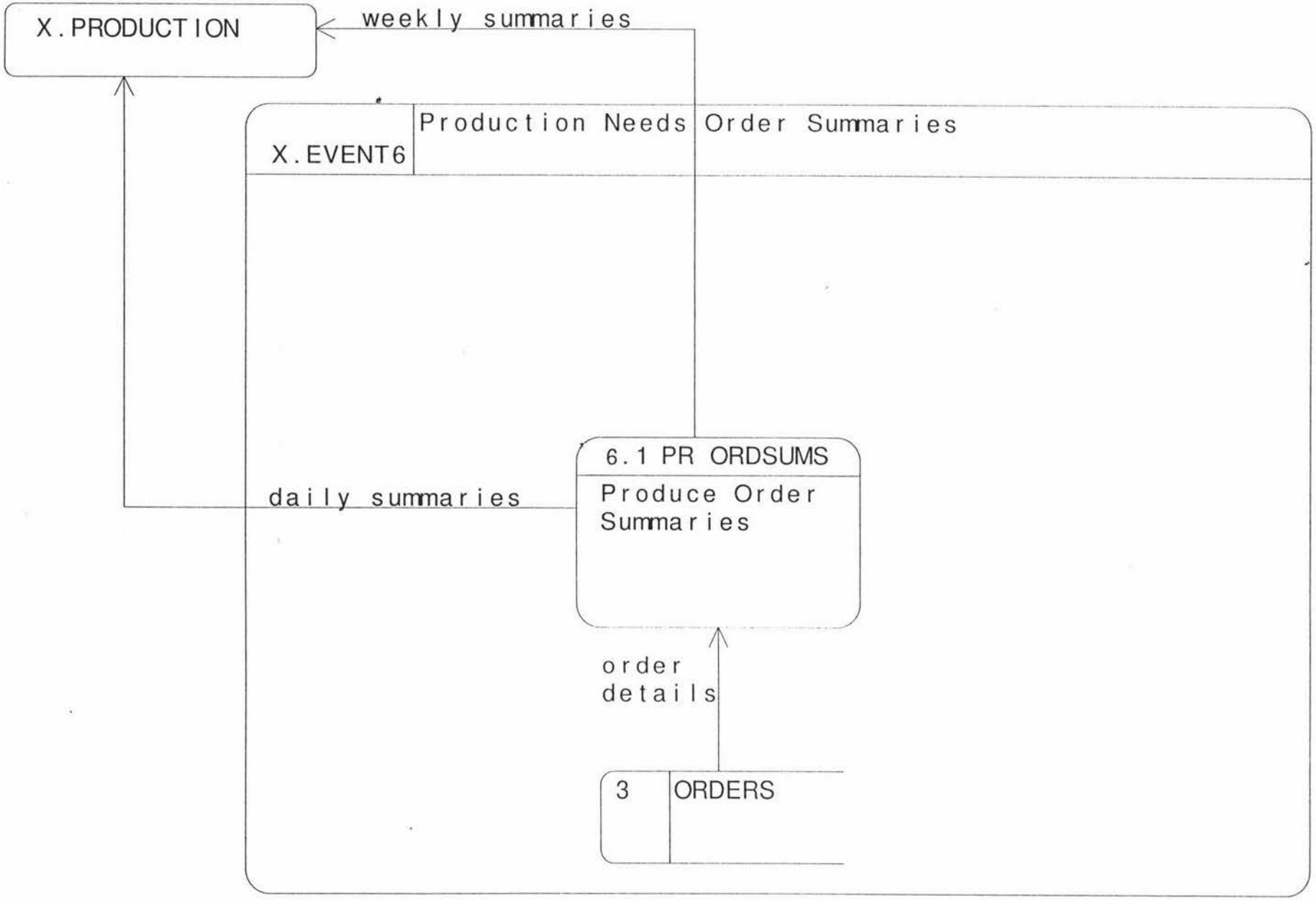

X.CUSTOMER
X.EVENT1
Customer requests information.
customer inquiry
1 RSPD
Respond to Order Information
order form
brochure
customer inquiry details
9 PRODUCT BROCHURES

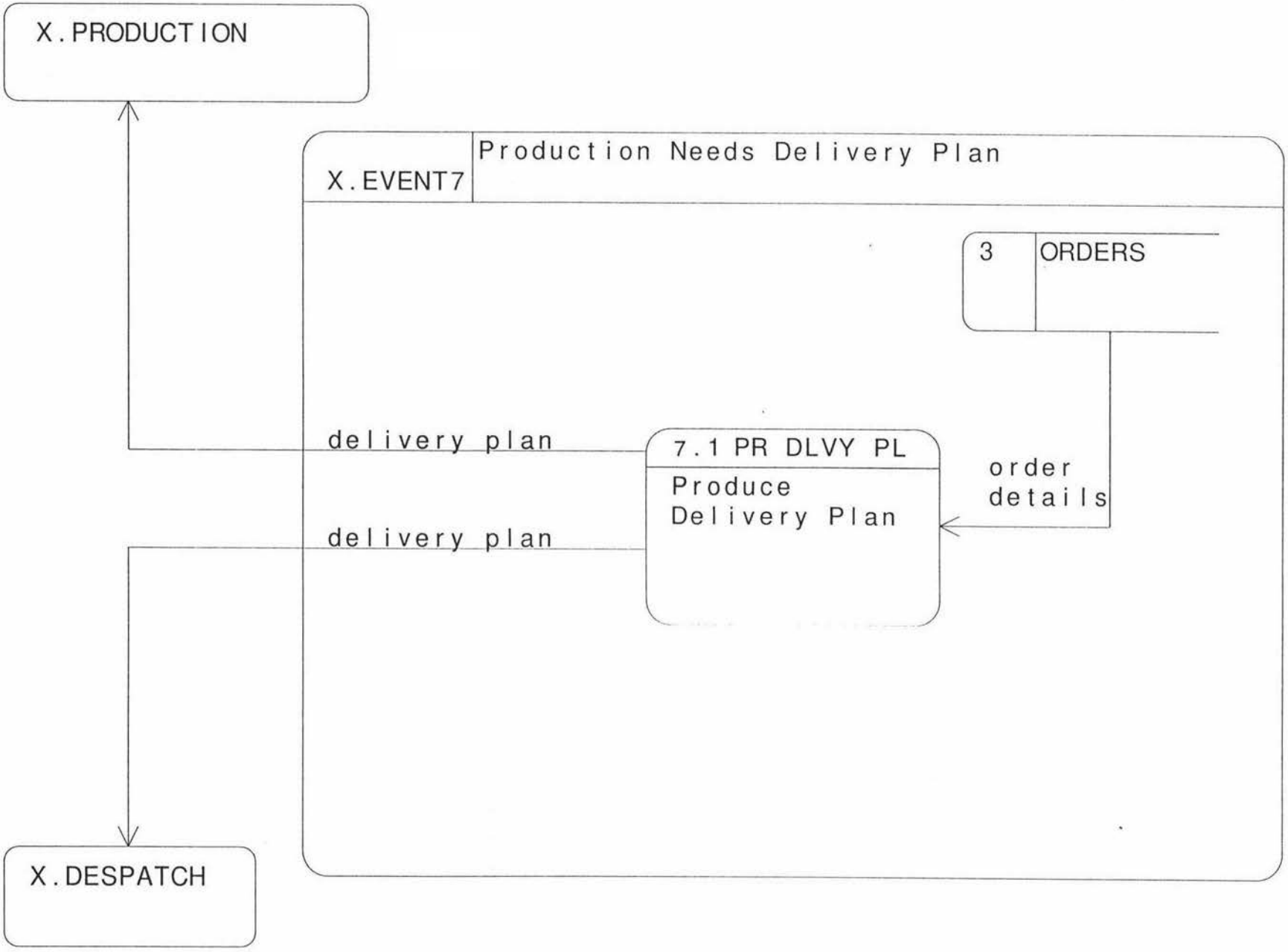

X.CUSTOMER
X.EVENT2
Customer Places Order
customer details
order
2.1 RSPD ORDER
Respond to Receipt of Order
customer details
order details
2
CUSTOMERS
3
ORDERS

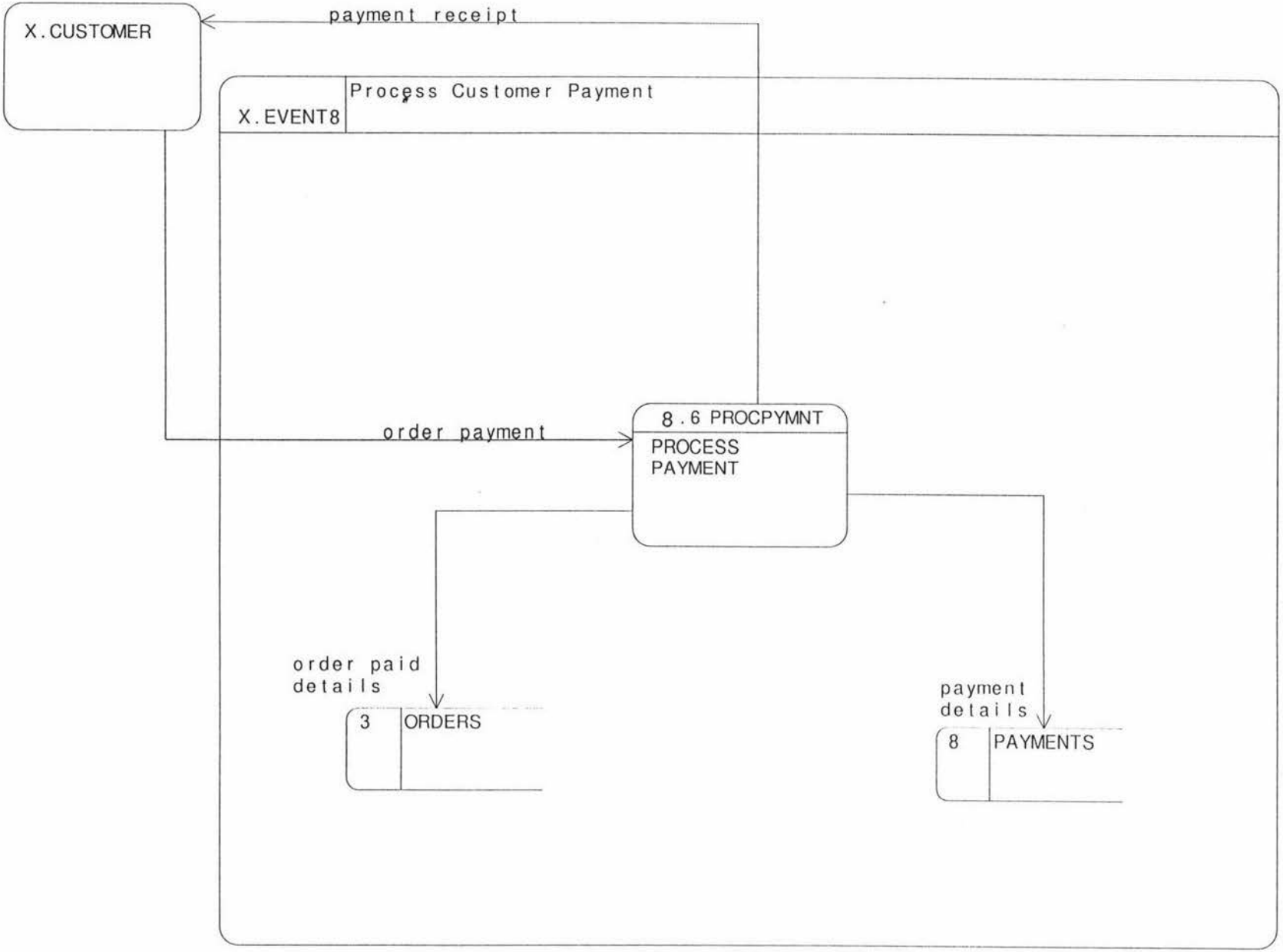

X.CUSTOMER
X.EVENT3
Customer Modifies Invalid Order
modifications
corrected order
3.1 VAL ORDER
Validate Order
order details
order details
stock status
stock status
3
ORDERS
4
PRODUCTS


X.CUSTOMER
order
status
X.EVENT4
Finance Director Determines Order Status
4.1 DT STATUS
Determine
Order Status
order status details
credit status
order
details
customer
credit
details
customer
details
order
status
2
CUSTOMERS
3
ORDERS
X.FINANCE
DIRECTOR


X.CUSTOMER
X.EVENT5
Copies of Order Arrive From Accounts
order
acknowledgement
5.1 PR ORDERS
Process
Orders
van copy
order details
despatch note
despatch instruction
office copy
order
details
X.ACCOUNTS
X.SALES
X.DESPATCH


X.PRODUCTION
weekly summaries
X.EVENT6
Production Needs Order Summaries
6.1 PR ORDSUMS
Produce Order Summaries
daily summaries
order details
3
ORDERS


X.PRODUCTION
X.EVENT7
Production Needs Delivery Plan
3
ORDERS
delivery plan
7.1 PR DLVY PL
Produce
Delivery Plan
order
details
delivery plan
X.DESPATCH


X.CUSTOMER
payment receipt
X.EVENT8
Process Customer Payment
order payment
8.6 PROCPYMNT
PROCESS
PAYMENT
order paid
details
3
ORDERS
payment
details
8
PAYMENTS

CUSTOMER

Which has significance as :
The entity customer is a person or organisation that wishes to make a purchase from the Trusty Furniture Company. The Customer entity is use d to store the customer's personal details such as name and address.

Information includes customer city address,
customer first name,
customer home phone number,
customer last name,
customer number,
customer street address,
customer suburb address,
customer title,
customer work phone number, etc.

Each must be places one or more ORDERS

ORDER

Which has significance as :
An order is a set of products that the customer wishes to purchase fro m the Trusty Furniture Company. New Customers must place an order usin g the required order form. Existing Customers are allowed to place an order by phone or by order form. The entity order is used to store inf ormation about each customers order.

Information includes customer number,
order amount,
order date,
order description,
order number,
order quantity,
product type number, etc.

Each may be contain one or more PRODUCTS
and must be requires one and only one PAYMENT
and may be are placed by one and only one CUSTOMER

PAYMENT

Which has significance as :
The entity payment is used to store information about which orders hav e payment included with them at the time of purchase. Customers may ch oose to pay for their order at the time of purchase rather than paying on credit.

Information includes order number,
payment amount,
payment number, etc.

Each may be are included with one and only one ORDER

PRODUCT

Which has significance as :
A product is an item produced by the Trusty Furniture Company for sale to its customers. The entity product is used by the system to store i nformation about the stock levels of different products.

Information includes product description,
product price,
product type number,
quantity on hand, etc.

Each must be feature in one or more PRODUCT BROCHURES
and must be are required by one or more ORDERS

PRODUCT BROCHURE

Which has significance as :
Product Brochure is used top store information about products availabl e for sale from th
e Trusty Furniture Company. Different brochures contain specific produ cts such as chairs. However one brochure contains all the products ava ilable from the Trusty Furniture Company.

Information includes brochure description,
brochure number,
product type number, etc.

Each may be contain one or more PRODUCTS

APPENDIX C: Structured Model Metric Results

Appendix C contains the following structured metric results from the Trusty Furniture Company's Order Processing System:

- One Flowgraph showing the system before changes were made

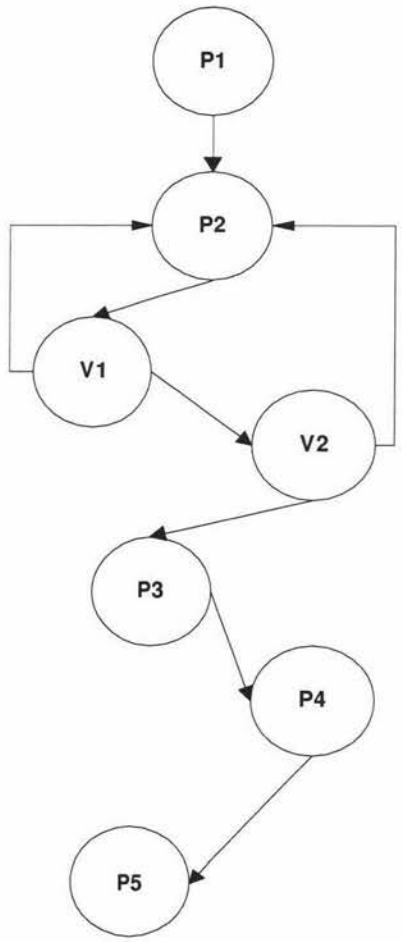
- One Flowgraph showing the system after changes were made

Trusty Furniture Company Order Processing System Flowgraph Before Changes

The flowgraph node numbers are prefixed by a letter to show which subsystem the process belongs to.

V = Validation subsystem

P = Process Customer Order subsystem



Flowgraph Node Number	Process
V1	Validate Order
V2	Determine Order Status
P1	Respond to Order Information
P2	Respond to Receipt of Order
P3	Process Orders
P4	Produce Order Summaries
P5	Produce Delivery Plan

Working:

Number of Compares = 2

McCabes Complexity = Number of Compares + 1

= 2 + 1

= 3

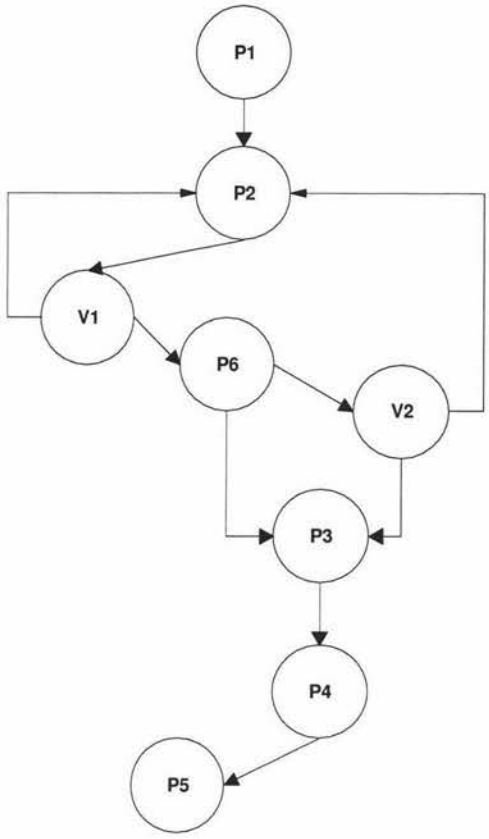
Trusty Furniture Company Order Processing System Flowgraph After Changes

The flowgraph node numbers are prefixed by a letter to show which subsystem the process belongs to.

V = Validation subsystem

P = Process Customer Order subsystem

Flowgraph Node Number	Process
V1	Validate Order
V2	Determine Order Status
P1	Respond to Order Information
P2	Respond to Receipt of Order
P3	Process Orders
P4	Produce Order Summaries
P5	Produce Delivery Plan
P6	Process Payment



Working:

Number of Compares = 3

McCabes Complexity = Number of Compares + 1

= 3 + 1

= 4

APPENDIX D: Structured Model Validation Results

Appendix D contains the following structured validation results from the Trusty Furniture Company's Order Processing System:

- One Flowgraph showing the system developed by student 1

- One Flowgraph showing the system developed by student 2

Trusty Furniture Company Order Processing System Flowgraph:

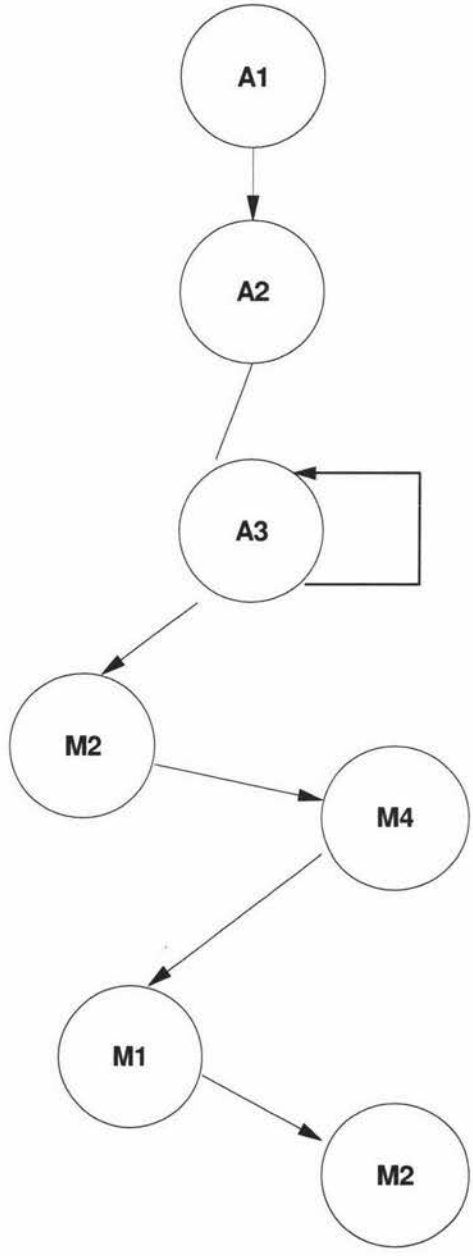
Results from Student 1

The flowgraph node numbers are prefixed by a letter to show which subsystem the process belongs to.

A = Accept Customer Order subsystem

M = Manage Orders subsystem

Flowgraph Node Number	Process
A1	Accept New Customer
A2	Maintain Existing Customer Information
A3	Verify Product Details
M1	Prepare Delivery Plan
M2	Maintain Order Information
M3	Record Despatches
M4	Manage Reports



Working:

Number of Compares = 1

McCabes Complexity = Number of Compares + 1

= 1 + 1

= 2

Trusty Furniture Company Order Processing System Flowgraph:

Results from Student 2

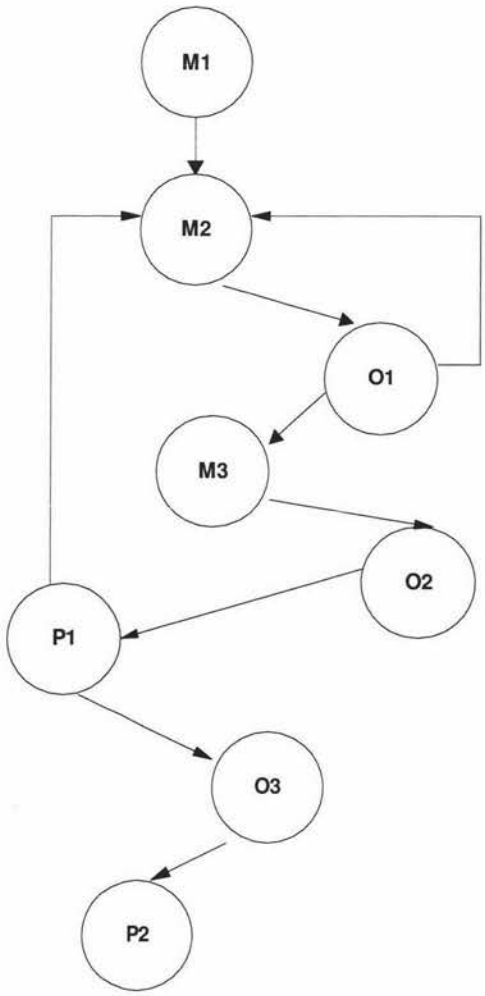
The flowgraph node numbers are prefixed by a letter to show which subsystem the process belongs to.

M = Manage Customers subsystem

O = Manage Orders subsystem

P = Manage Products subsystem

Flowgraph Node Number	Process
M1	Accept Customer Information
M2	Accept Order Details
M3	Accept Approved Order
O1	Handle Orders
O2	Complete Order
O3	Generate Reports
P1	Ensure Order Fulfilment
P2	Manage Chair Supplies



Working:

Number of Compares = 2

McCabes Complexity = Number of Compares + 1

= 2 + 1

= 3

APPENDIX E: Outputs from Object-Oriented Model Before Changes

Appendix E contains the following object-oriented output for the Trusty Furniture Company's Order Processing System:

- One System Function Statement
- One set of Class Diagrams (2 Levels)
- One set of Interaction Diagrams (6 Diagrams)
- One set of Object Diagrams (6 Diagrams)
- Selected Data Dictionary Output

Trusty Furniture Company Order Processing System

System Function Statement (Before Changes)

1. Customer requests Order Information Pack

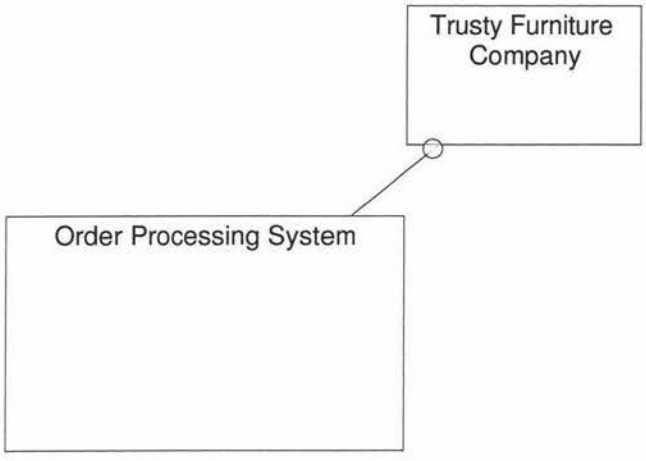
2. Customer places Order

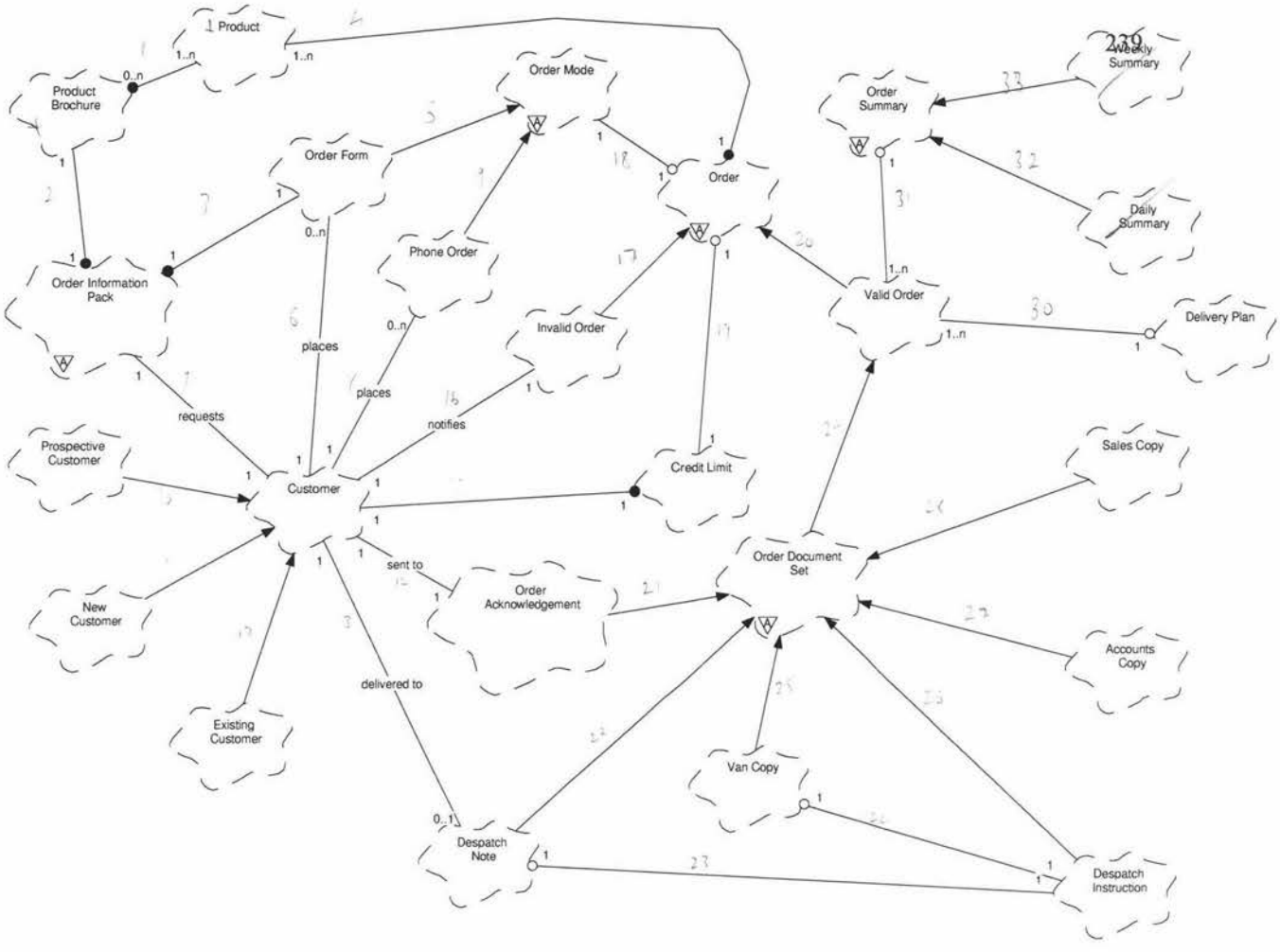
3. Order validated

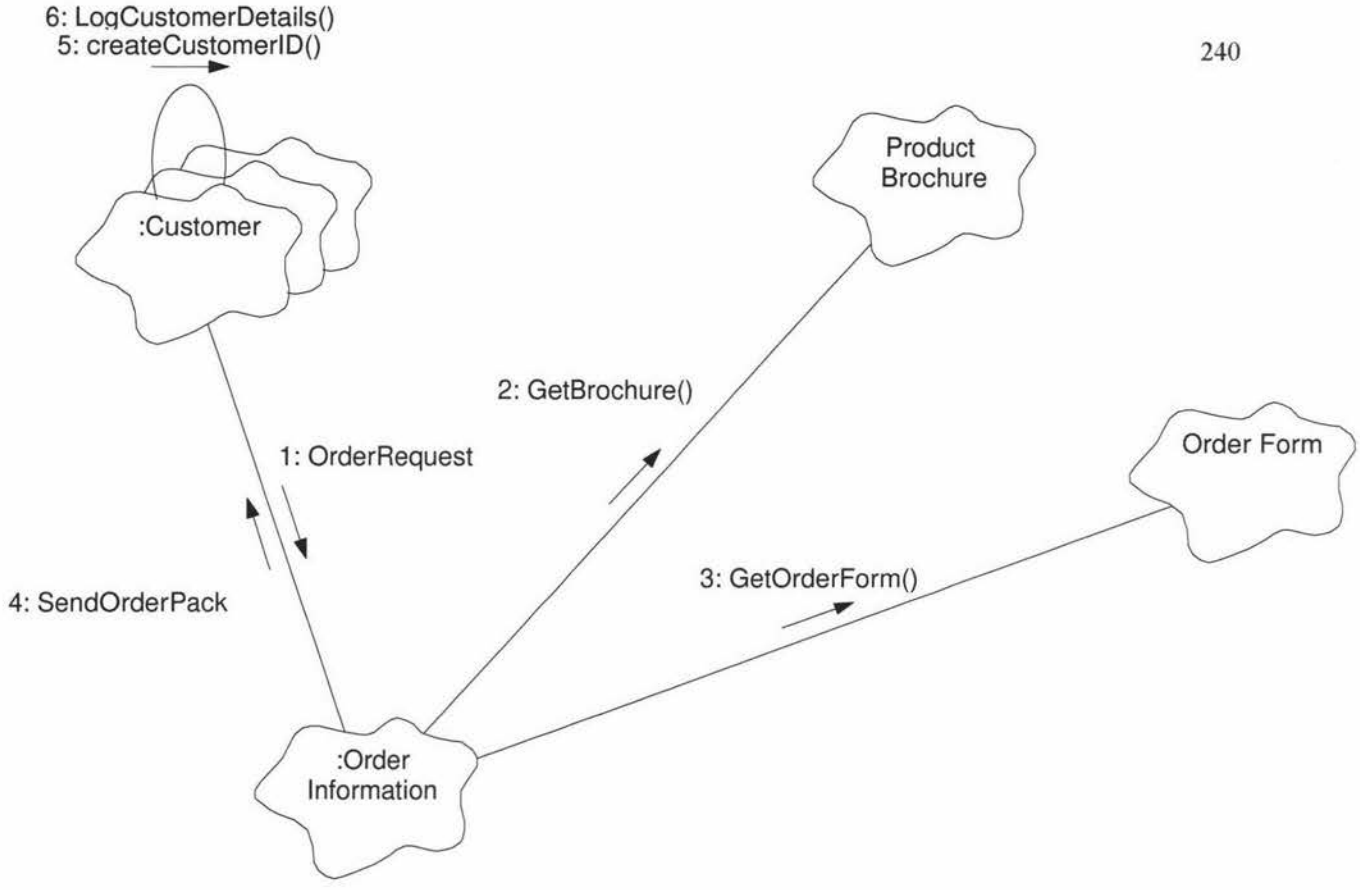
4. Copies of Order made

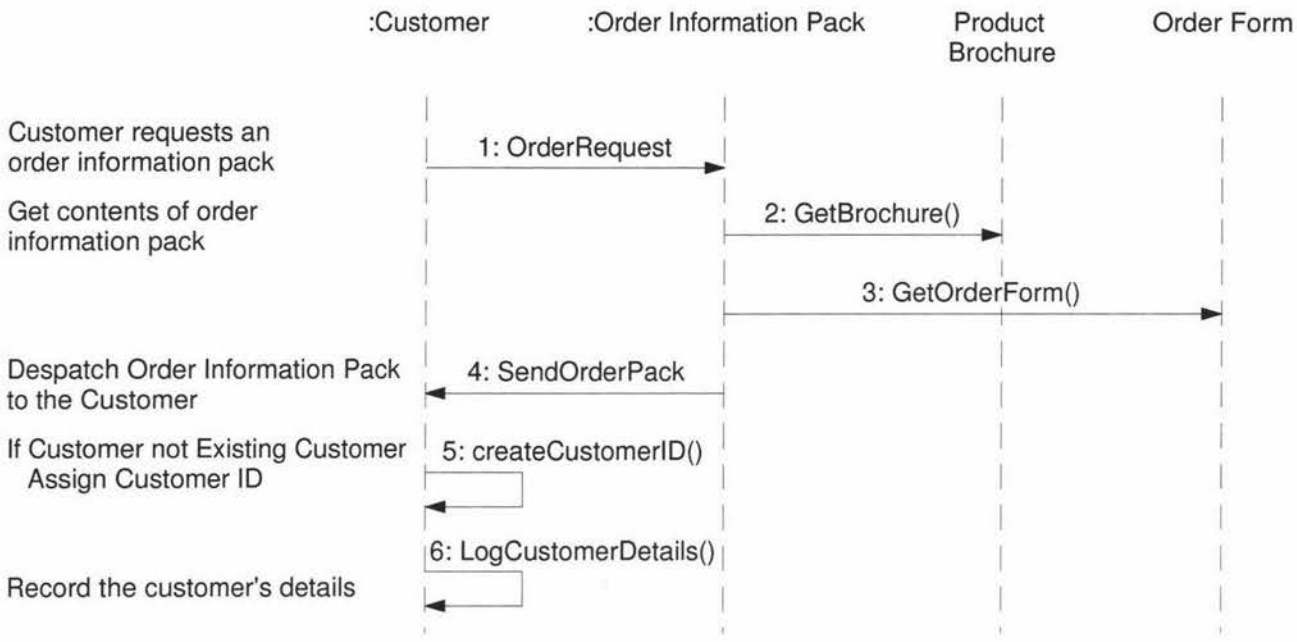
5. Order Summaries generated

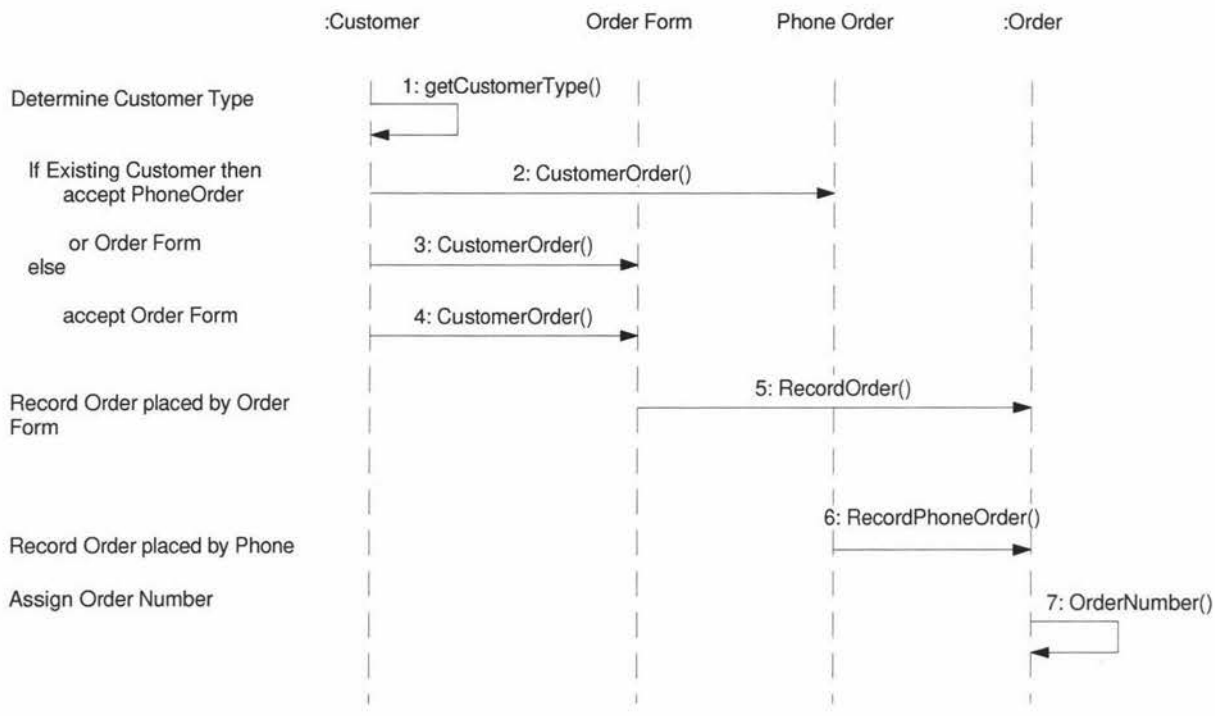
6. Delivery Plan generated

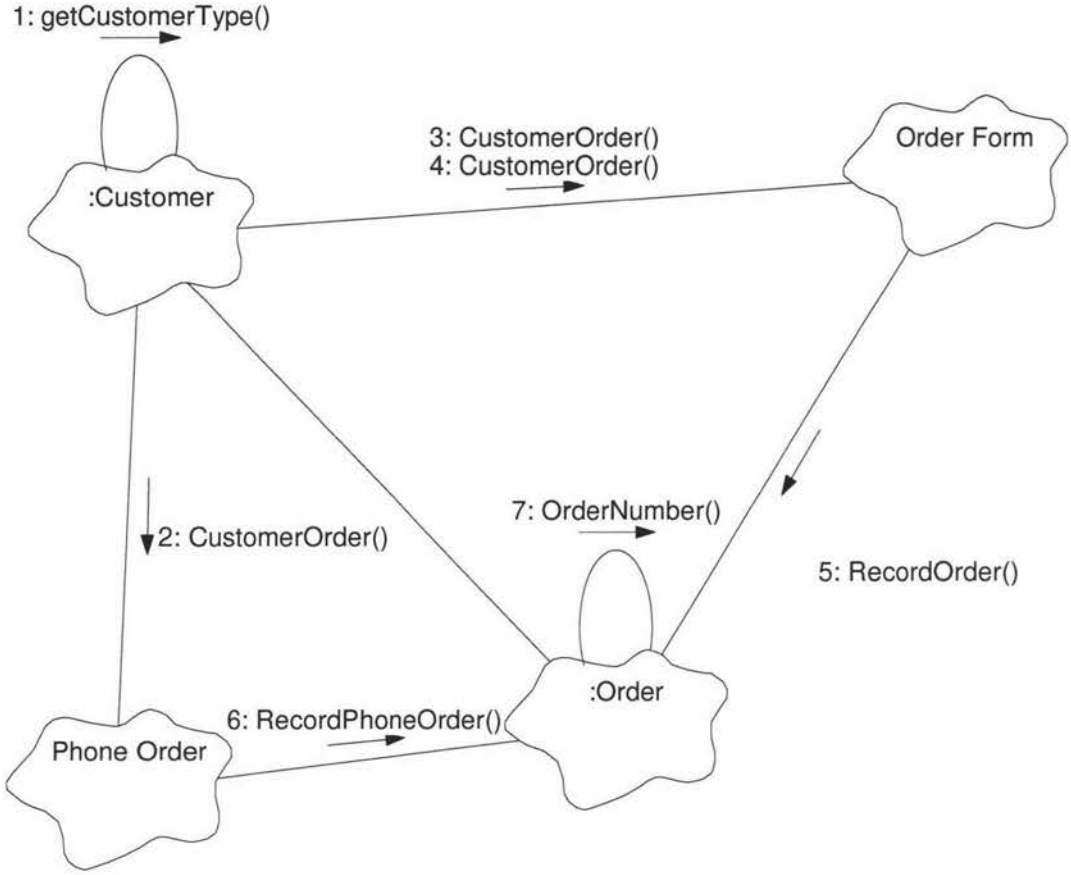

Trusty Furniture Company
Order Processing System

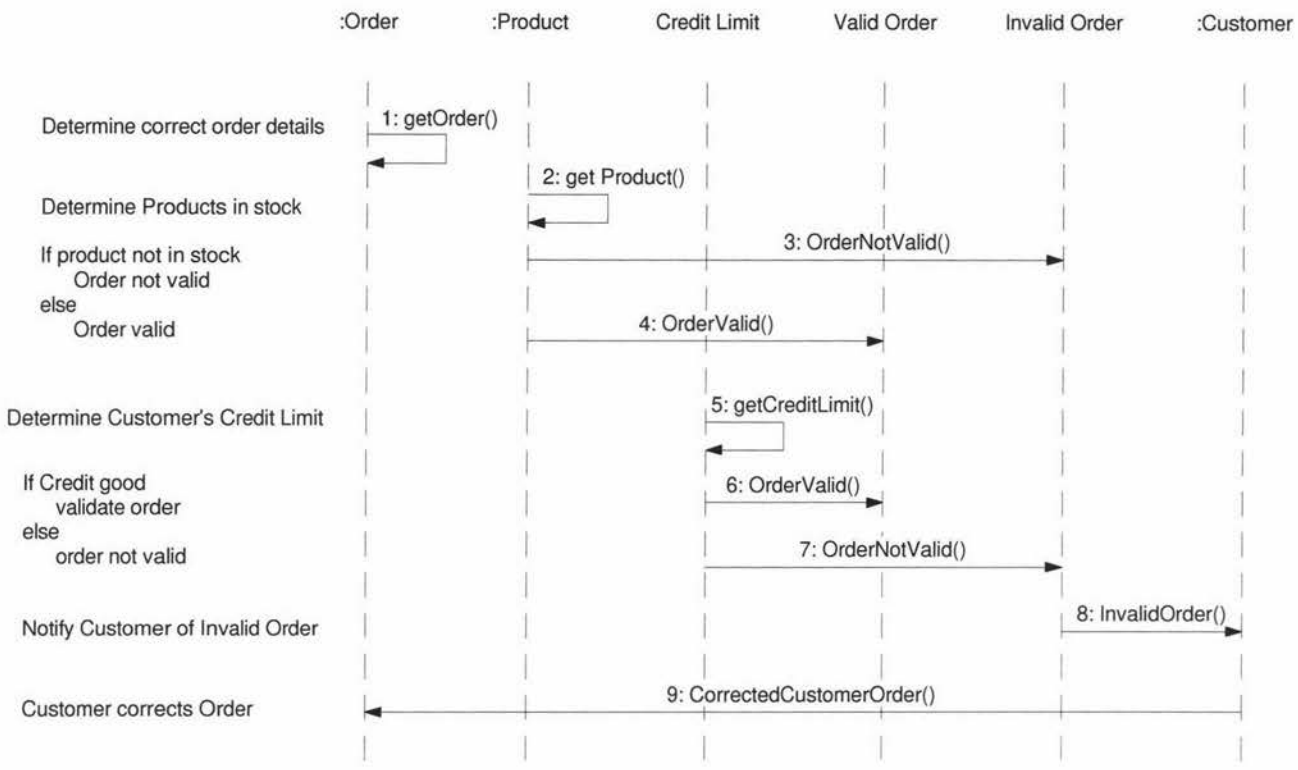

Product
Product Brochure
Order Mode
Order Form
Order
Order Summary
Weekly Summary
Daily Summary
Order Information Pack
Phone Order
Invalid Order
Valid Order
Delivery Plan
places
places
requests
notifies
Prospective Customer
Customer
Credit Limit
Sales Copy
sent to
Order Acknowledgement
Order Document Set
New Customer
Existing Customer
delivered to
Accounts Copy
Van Copy
Despatch Note
Despatch Instruction

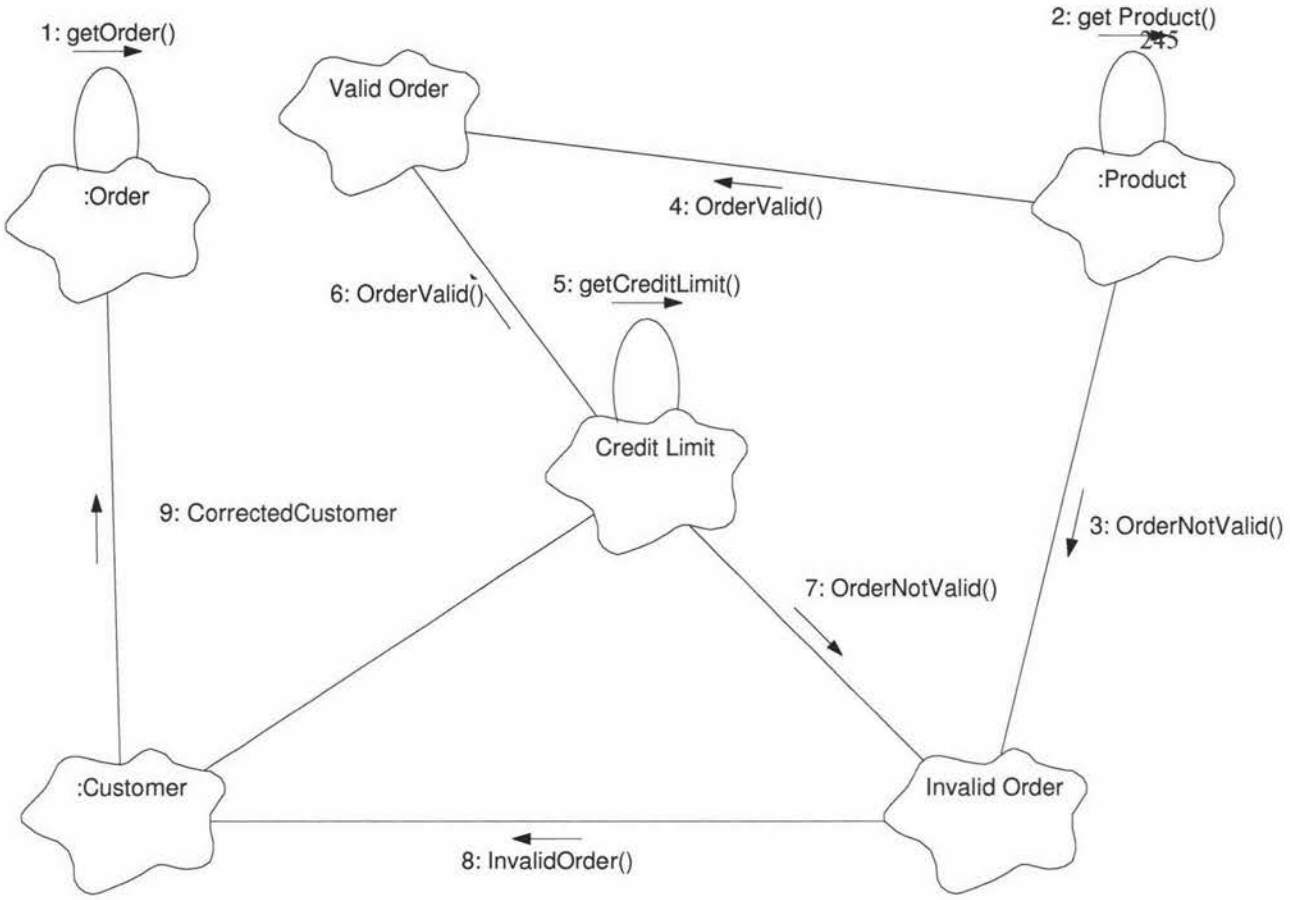

6: LogCustomerDetails()
5: createCustomerID()
:Customer
Product Brochure
2: GetBrochure()
1: OrderRequest
Order Form
3: GetOrderForm()
4: SendOrderPack
:Order Information

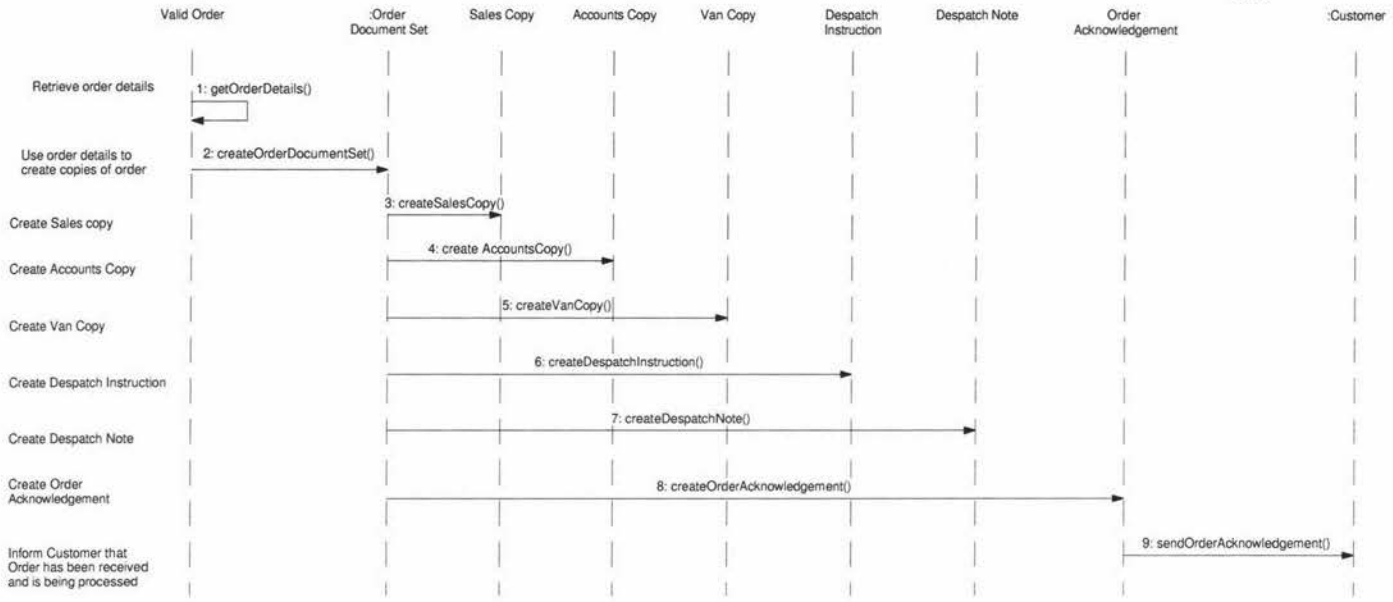

:Customer
:Order Information Pack
Product Brochure
Order Form
Customer requests an order information pack
1: OrderRequest
Get contents of order information pack
2: GetBrochure()
3: GetOrderForm()
Despatch Order Information Pack to the Customer
4: SendOrderPack
If Customer not Existing Customer Assign Customer ID
5: createCustomerID()
6: LogCustomerDetails()
Record the customer's details

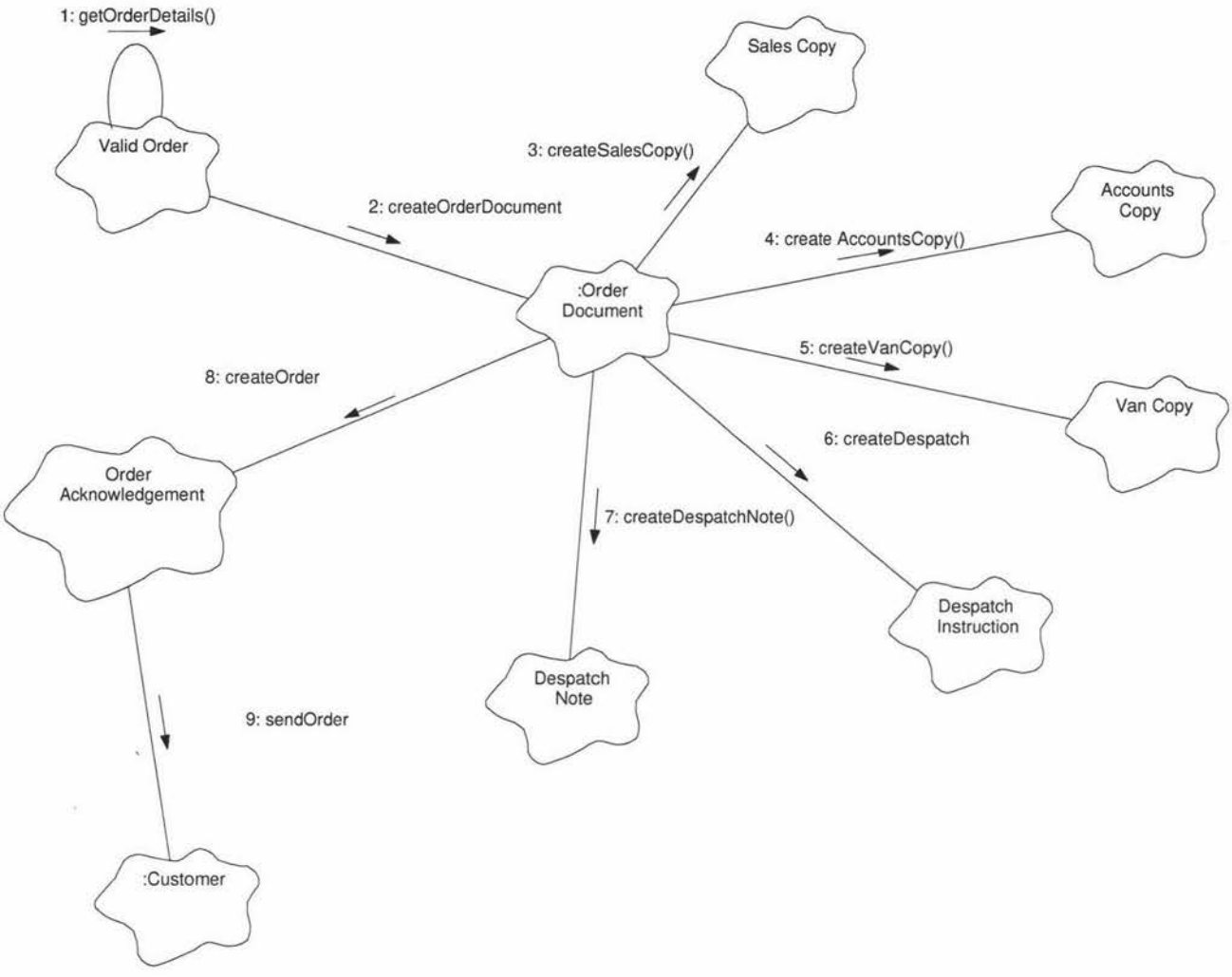

:Customer
Order Form
Phone Order
:Order
Determine Customer Type
1: getCustomerType()
If Existing Customer then
accept PhoneOrder
2: CustomerOrder()
or Order Form
else
3: CustomerOrder()
accept Order Form
4: CustomerOrder()
Record Order placed by Order Form
5: RecordOrder()
Record Order placed by Phone
6: RecordPhoneOrder()
Assign Order Number
7: OrderNumber()

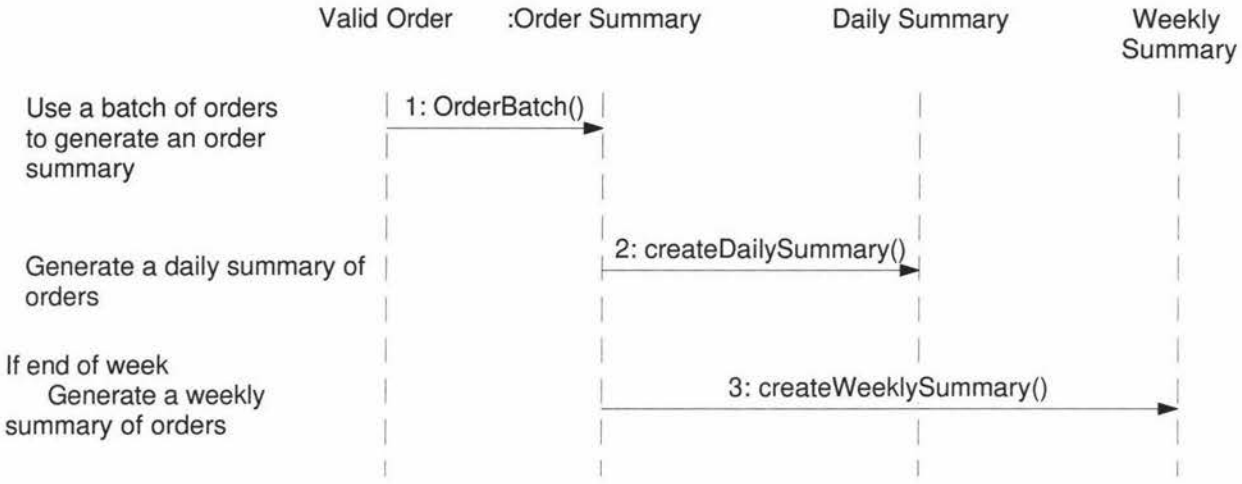

1: getCustomerType()
:Customer
3: CustomerOrder()
4: CustomerOrder()
Order Form
2: CustomerOrder()
7: OrderNumber()
5: RecordOrder()
:Order
6: RecordPhoneOrder()
Phone Order

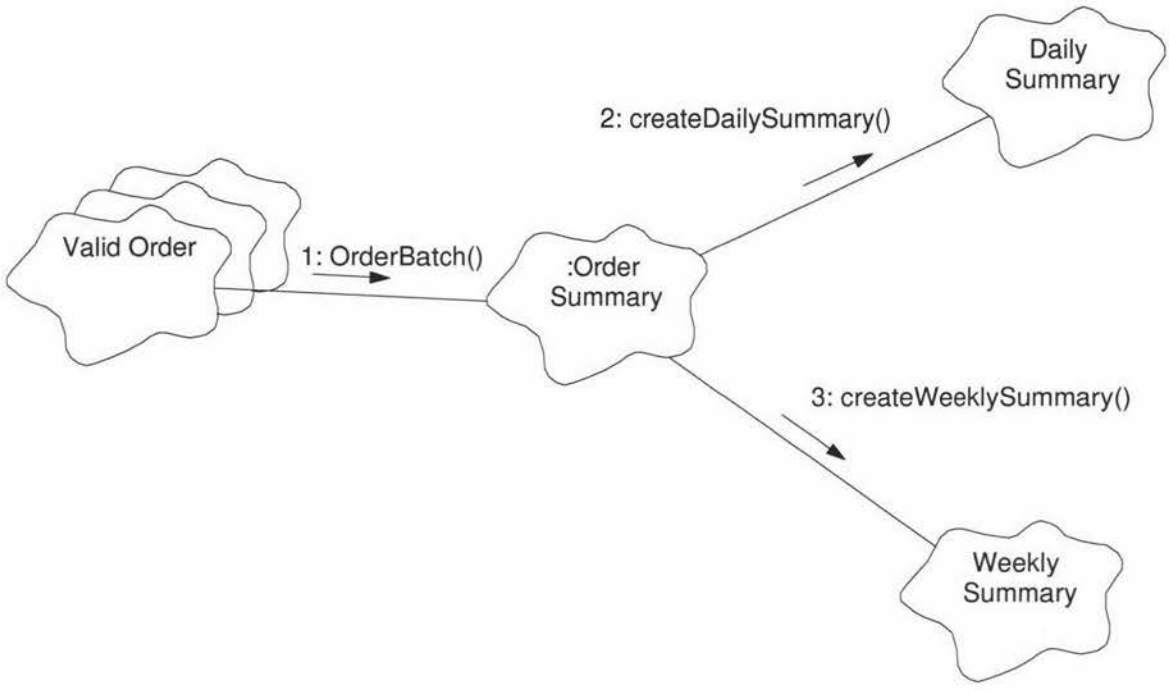

:Order
:Product
Credit Limit
Valid Order
Invalid Order
:Customer
Determine correct order details
1: getOrder()
2: get Product()
Determine Products in stock
If product not in stock
Order not valid
else
Order valid
3: OrderNotValid()
4: OrderValid()
Determine Customer's Credit Limit
5: getCreditLimit()
If Credit good
validate order
else
order not valid
6: OrderValid()
7: OrderNotValid()
Notify Customer of Invalid Order
8: InvalidOrder()
Customer corrects Order
9: CorrectedCustomerOrder()

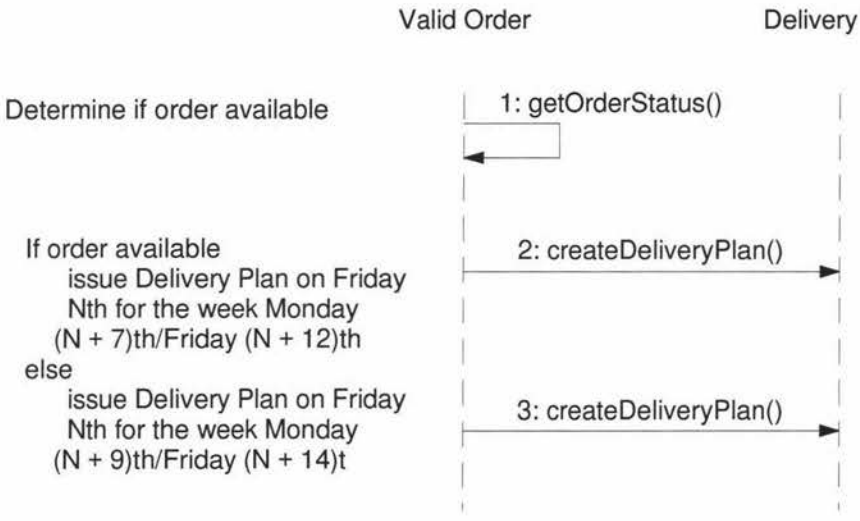

1: getOrder()
2: get Product()
:Order
Valid Order
:Product
4: OrderValid()
6: OrderValid()
5: getCreditLimit()
Credit Limit
9: CorrectedCustomer
3: OrderNotValid()
7: OrderNotValid()
:Customer
Invalid Order
8: InvalidOrder()

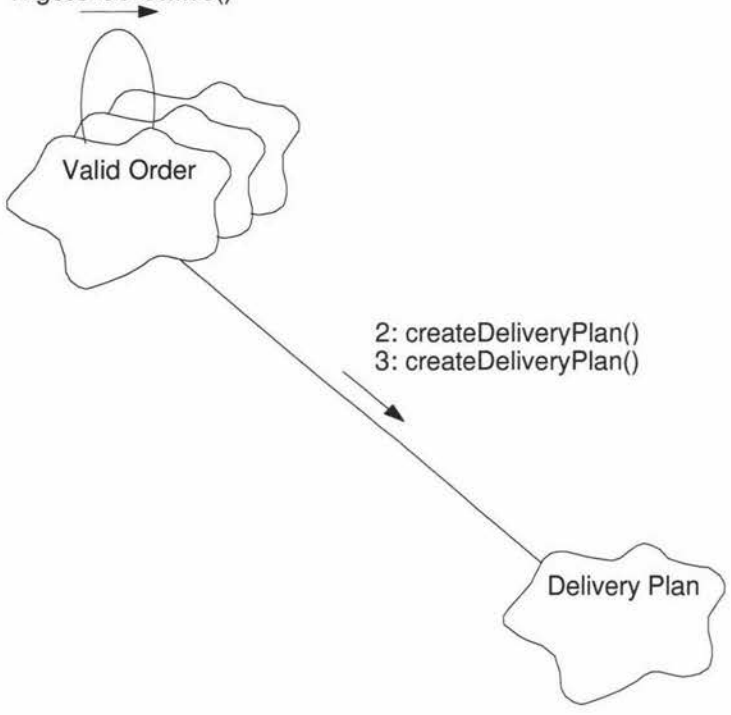

Valid Order
:Order Document Set
Sales Copy
Accounts Copy
Van Copy
Despatch Instruction
Despatch Note
Order Acknowledgement
:Customer
Retrieve order details
1: getOrderDetails()
Use order details to create copies of order
2: createOrderDocumentSet()
3: createSalesCopy()
Create Sales copy
4: create AccountsCopy()
Create Accounts Copy
5: createVanCopy()
Create Van Copy
6: createDespatchInstruction()
Create Despatch Instruction
7: createDespatchNote()
Create Despatch Note
Create Order Acknowledgement
8: createOrderAcknowledgement()
9: sendOrderAcknowledgement()
Inform Customer that Order has been received and is being processed


1: getOrderDetails()
Valid Order
Sales Copy
3: createSalesCopy()
2: createOrderDocument
Accounts Copy
4: create AccountsCopy()
:Order Document
5: createVanCopy()
Van Copy
8: createOrder
6: createDespatch
Order Acknowledgement
7: createDespatchNote()
Despatch Instruction
Despatch Note
9: sendOrder
:Customer


Valid Order
:Order Summary
Daily Summary
Weekly Summary
Use a batch of orders to generate an order summary
1: OrderBatch()
Generate a daily summary of orders
2: createDailySummary()
If end of week
Generate a weekly summary of orders
3: createWeeklySummary()


Daily
Summary
2: createDailySummary()
Valid Order
1: OrderBatch()
:Order
Summary
3: createWeeklySummary()
Weekly
Summary


Valid Order
Delivery Plan
Determine if order available
1: getOrderStatus()
If order available
issue Delivery Plan on Friday
Nth for the week Monday
(N + 7)th/Friday (N + 12)th
else
issue Delivery Plan on Friday
Nth for the week Monday
(N + 9)th/Friday (N + 14)t
2: createDeliveryPlan()
3: createDeliveryPlan()


1: getOrderStatus()
Valid Order
2: createDeliveryPlan()
3: createDeliveryPlan()
Delivery Plan

Class name:

Accounts Copy

Category: Order Processing System
Documentation:
Accounts Copy is a copy of the original order which is kept in the Accounts department for billing purposes.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Credit Limit

Category: Order Processing System
Documentation:
The credit limit determines if a customer is able to pay for products that they have ordered. The credit limit also shows the customer's outstanding debt and credit limit.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:
Customer

Operations:
getCreditLimit (Customer ID, Credit Limit)
OrderNotValid (Customer ID, Order Number)
OrderValid (Customer ID, Order Number)

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Customer

Category: Order Processing System
Documentation:
Customer is the person or organisation who places an order or requests information from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Associations:

Order Information•
Pack requests
A customer may request an order information pack from the Trusty Furniture Company in order to receive a product brochure and an order form.

Order Form places
A customer may place an order with the Trusty Furniture Company by using an order form.

Phone Order places
An existing customer may place an order with the Trusty Furniture Company over the telephone.

Public Interface:
Operations:

CorrectedCustomerOrder (Order Details, Order Quantity)
createCustomerID (Customer First Name, Customer Last Name, Customer Phone Number, C
CustomerOrder (Order Details, Order Quantity)
getCustomerType (Customer Type)
LogCustomerDetails (Customer First Name, Customer Last Name, Customer Address, Custo
OrderRequest (Customer First Name, Customer Last Name, Customer Address, Customer P

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Daily Summary

Category: Order Processing System
Documentation:
Daily Summary is a summary of the orders received by the Trusty Furniture Company that is produced on a daily basis. The summary is analysed by product.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Summary
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Delivery Plan

Category: Order Processing System
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Uses:

Valid Order

State machine: No

Concurrency: Sequential
Persistence: Persistent

Class name:

Despatch Instruction

Category: Order Processing System
Documentation:

Despatch Instruction is a copy of the order which is used to check which items are actually loaded at the time of despatch. The items which are loaded copy through to the Despatch Note and the Van Copy. Despatch Instruction is initially sent to the Despatch Manager.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Despatch Note

Category: Order Processing System
Documentation:

Despatch Note is sent to the Sales department by the Despatch manager. Despatch Note notes any shortages from the original order. At this point the Despatch Note is forwarded to the Customer notifiying them of any shortages. Despatch Note also uses the Despatch Instruction to note which items were loaded and sent to the Customer at despatch time.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
Associations:

Customer delivered to
A Despatch note is delivered to a Customer to notify them if there are any shortages.

Public Uses:

Despatch Instruction

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Existing Customer

Category: Order Processing System

Documentation:
This is a customer who has previously purchased goods from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Customer
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Invalid Order

Category: Order Processing System
Documentation:
Invalid Order is a list of items for purchase by a Customer that has not been approved. Reasons for declining an order include: an order form that has been incorrectly filled out, failure to obtain credit approval, and orders for products that have been discontinued/are out of stock or have had a change in price.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order
Associations:

Customer notifies
Notifies the Customer if there is a problem with their order.

Public Interface:
Operations:
InvalidOrder (Customer ID, Reason for Invalid Order, Order Number)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
New Customer

Category: Order Processing System
Documentation:
This is a person who has not ordered from the Trusty Furniture Company before.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Customer
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Order

Category: Order Processing System
Documentation:
Order is a list of items selected by the customer for purchase.

Export Control: Public
Cardinality: 1
Hierarchy:
Superclasses: none
Public Uses:
Credit Limit
Order Mode

Public Interface:
Has-A Relationships:
Product

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Order Acknowledgement

Category: Order Processing System
Documentation:
Order Acknowledgement is a copy of the order which is sent to the customer to notify them that their order has been recieved, approved and is currently being processed.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
Associations:

Customer sent to
An order acknowledgement form is sent to all customers who have a valid order. The order acknowledgement form show that the order has been received, validate, and is currently being processed.

Public Interface:
Operations:
sendOrderAcknowledgement (Customer ID, Customer First Name, Customer Last Name, Cus

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Order Document Set

Category: Order Processing System
Documentation:
Order Document Set is a collection of copies of the original order which are made after the order has been approved.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Valid Order
Public Interface:
Operations:
createAccountsCopy (Order Number, Customer ID, Product Quantity, Product Description)
createDespatchInstruction (Customer ID, Order Number, Product Description, Product Quantit
createDespatchNote (Customer ID, Order Number, Product Description, Product Quantity)
createOrderAcknowledgement (Customer ID, Order Number, Product Description, Product Qu
createSalesCopy (Customer ID, Order Number, Product Description, Product Quantity)
createVanCopy (Customer ID, Order Number, Product Quantity, Product Description)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Order Form

Category: Order Processing System
Documentation:
A form used by the customer to detail their order.

Export Control: Public
Cardinality: 1
Hierarchy:
Superclasses: Order Mode
Public Interface:
Operations:
RecordOrder (Customer First Name, Customer Last Name, Customer Address, Customer Pho

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Order Information• Pack

Category: Order Processing System
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:
Product Brochure
Order Form

Operations:

GetBrochure ( )
GetOrderForm ( )
SendOrderPack ( )

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Order Mode

Category: Order Processing System
Documentation:

Order Mode is used to determine the means by which an order has been placed with the Trusty Furniture Company. An order may be placed by Order Form or by Phone if the Customer has already purchased goods from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Order Summary

Category: Order Processing System
Documentation:

Order Summary is a summary of the orders recieved by the Trusty Furniture Company. The summary is analysed by product.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Uses:

Valid Order

Public Interface:
Operations:

createDailySummary (Order Number, Product Description, Product Quantity, Date)
createWeeklySummary (Order Number, Product Description, Product Quantity, Date)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Phone Order

Category: Order Processing System
Documentation:
Phone Order is an order for products placed by a customer. Only existing customers of the Trusty Furniture Company may place an order by phone, all other customers must place their order using the supplied order form.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Mode
Public Interface:
Operations:
RecordPhoneOrder (Customer Phone Number, Customer First Name, Customer Last Name,

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Product

Category: Order Processing System
Documentation:
Product is an item produced by the Trusty Furniture Company for sale.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
getProduct (Product Name, Product Quantity, Products in Stock)
OrderNotValid (Products in Stock, Credit Limit)
OrderValid (Products in Stock, Credit Limit)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Product Brochure

Category: Order Processing System
Documentation:
Is a catalogue of the Trusty Furniture Company's product line plus the prices of each product.

Export Control: Public
Cardinality: 1
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:
Product

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Prospective Customer

Category: Order Processing System
Documentation:
This is a customer who has expressed an interested in the range of products sold by the Trusty Furniture Company. In order to find out more information about the products available for sale they have requested an order information pack. A prospective customer may or may not purchase goods from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Customer
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Sales Copy

Category: Order Processing System
Documentation:
Sales Copy is a copy of the order form that is sent to the salesperson who made the sale via the sales department.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Valid Order

Category: Order Processing System
Documentation:
Valid Order is a list of items for purchase by a Customer that has been approved.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order
Public Interface:
Operations:
createDeliveryPlan (Customer ID, Order Number)

createOrderDocumentSet (Customer ID, Product Description, Product Quantity, Order Numbe
getOrderDetails (Product Description, Product Quantity, Customer ID, Order Number)
getOrderStatus (Customer ID, Order Number)
OrderBatch (Date)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Van Copy

Category: Order Processing System
Documentation:

Van Copy is a copy of the order which goes with the driver to obtain the customers acknowledgement of receipt. Van copy also uses the Despatch Instruction to note which items were loaded and sent to the Customer at despatch time.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
Public Uses:
Despatch Instruction

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Weekly Summary

Category: Order Processing System
Documentation:

Weekly Summary is a summary of the orders received by the Trusty Furniture Company that is produced on a weekly basis. The summary is analysed by product.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Summary
State machine: No
Concurrency: Sequential
Persistence: Persistent

APPENDIX F: Outputs from Object-Oriented Model After Changes

Appendix F contains the following object-oriented output for the Trusty Furniture Company's Order Processing System:

- One System Function Statement
- One set of Class Diagrams (2 Levels)
- One set of Interaction Diagrams (7 Diagrams)
- One set of Object Diagrams (7 Diagrams)
- Selected Data Dictionary Output

Trusty Furniture Company Order Processing System

System Function Statement (After Changes)

1. Customer requests Order Information Pack

2. Customer places Order

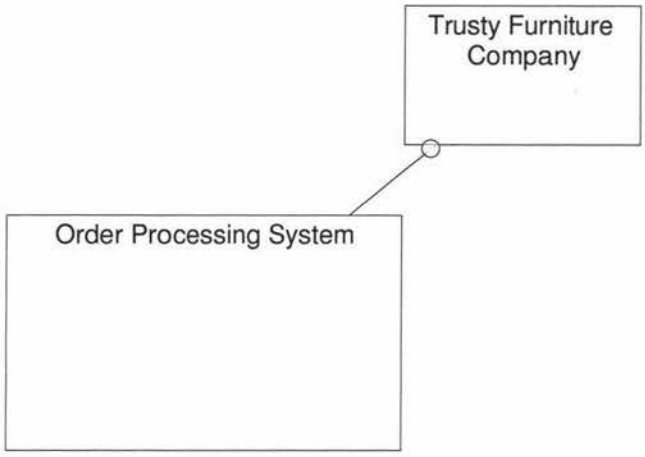
3. Customer pays for goods at time of purchase

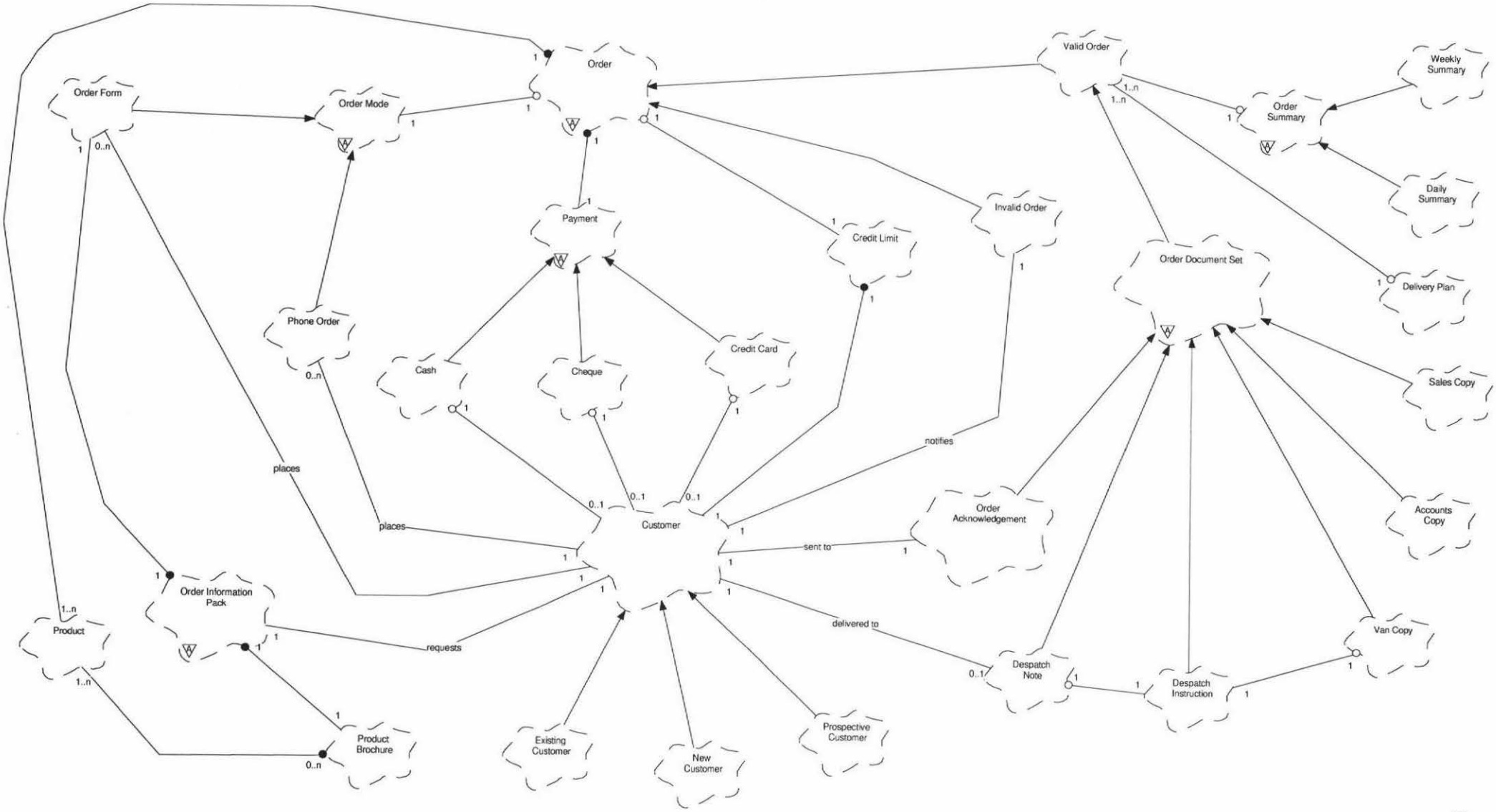
4. Order validated

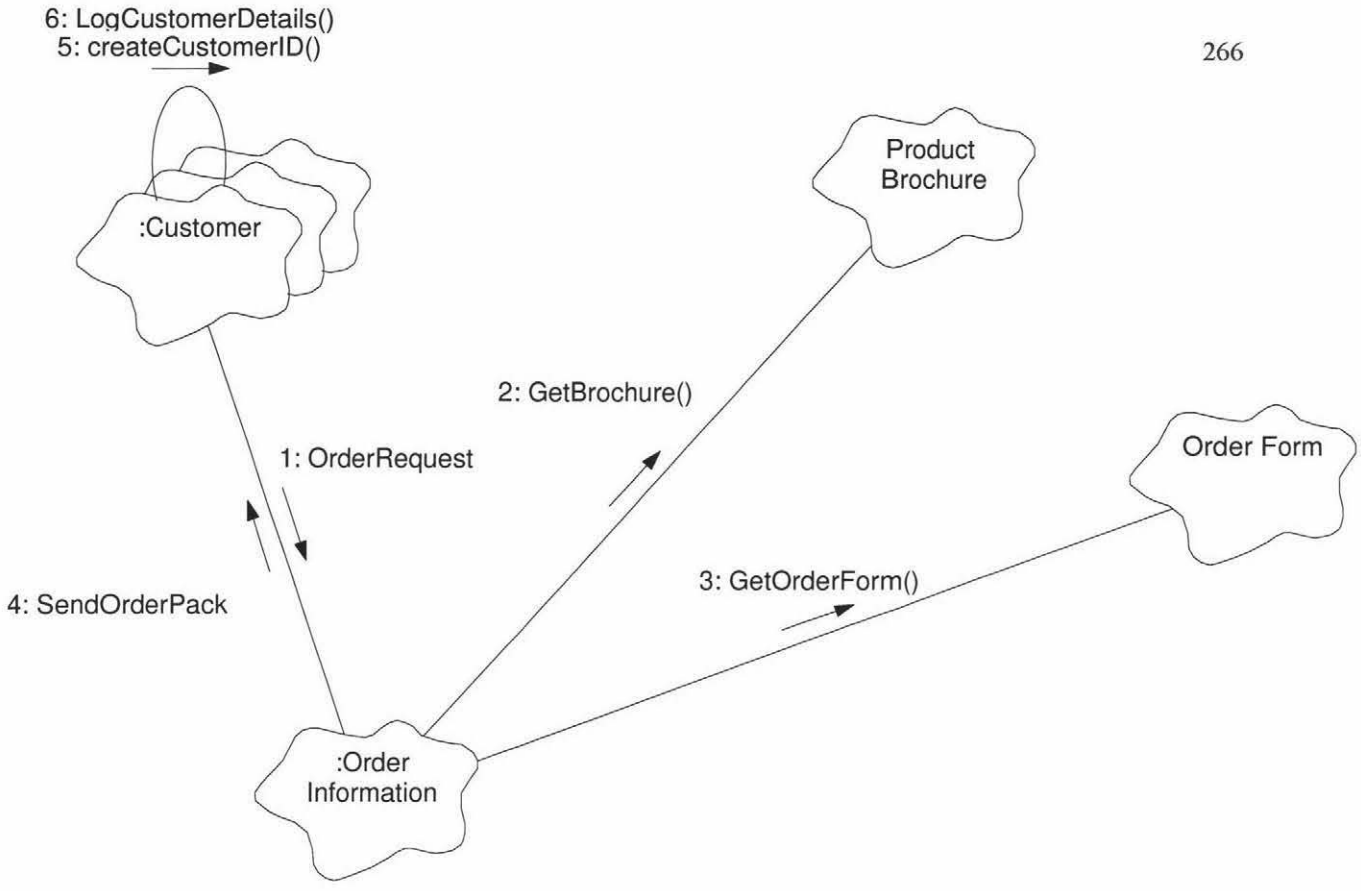
5. Copies of Order made

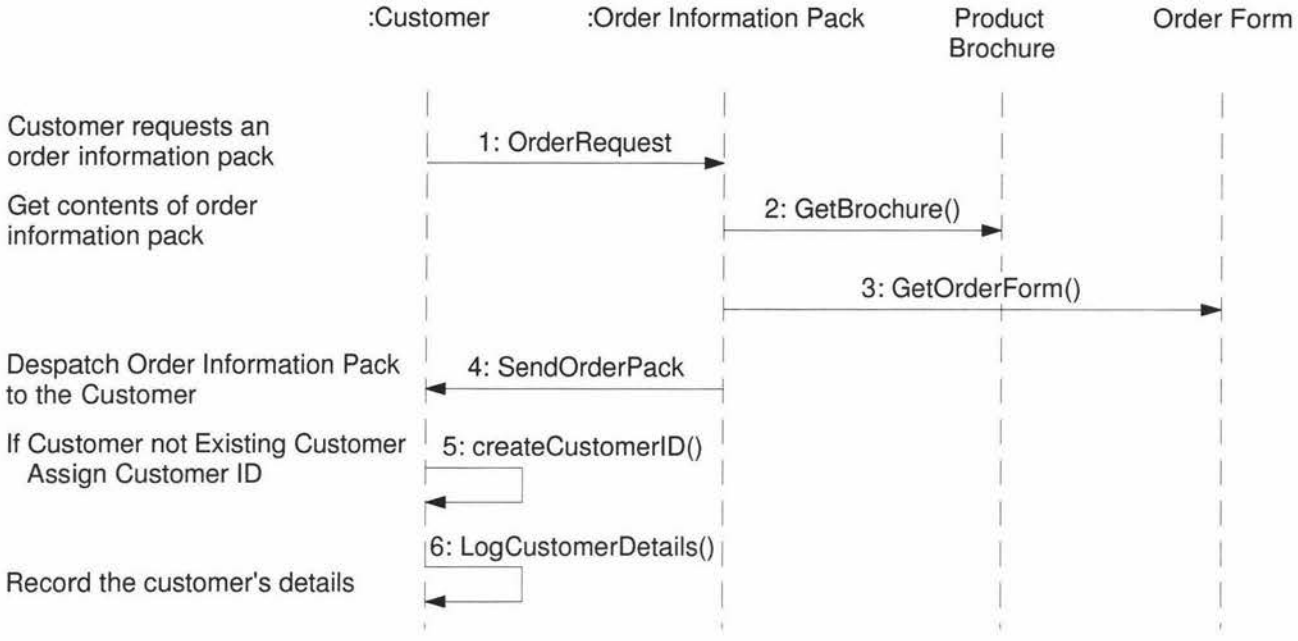
6. Order Summaries generated

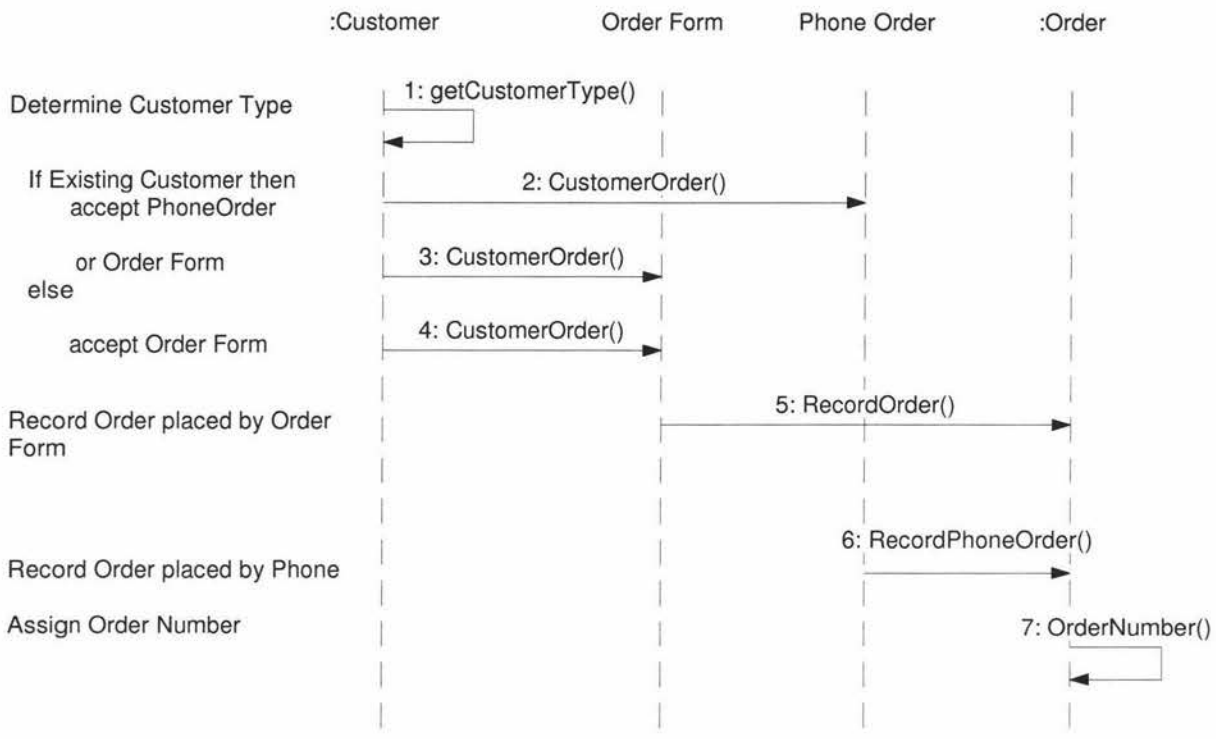
7. Delivery Plan generated

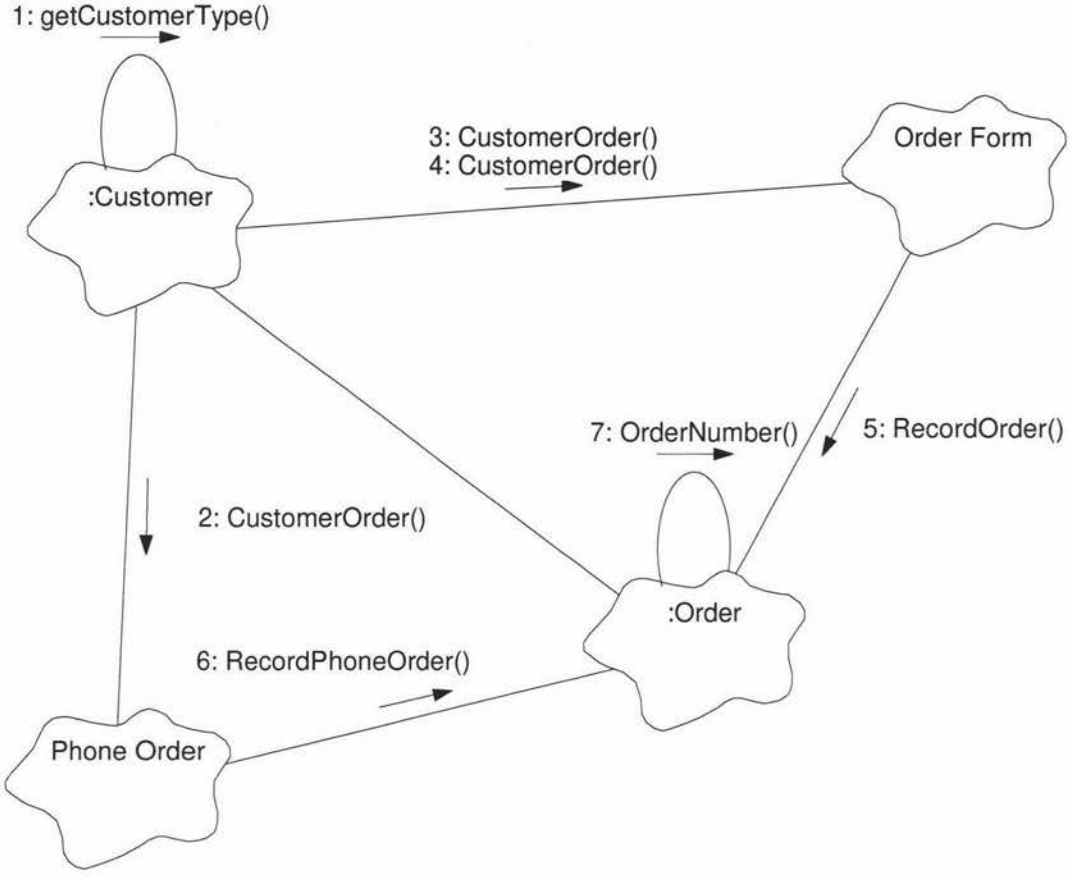

Trusty Furniture Company
Order Processing System

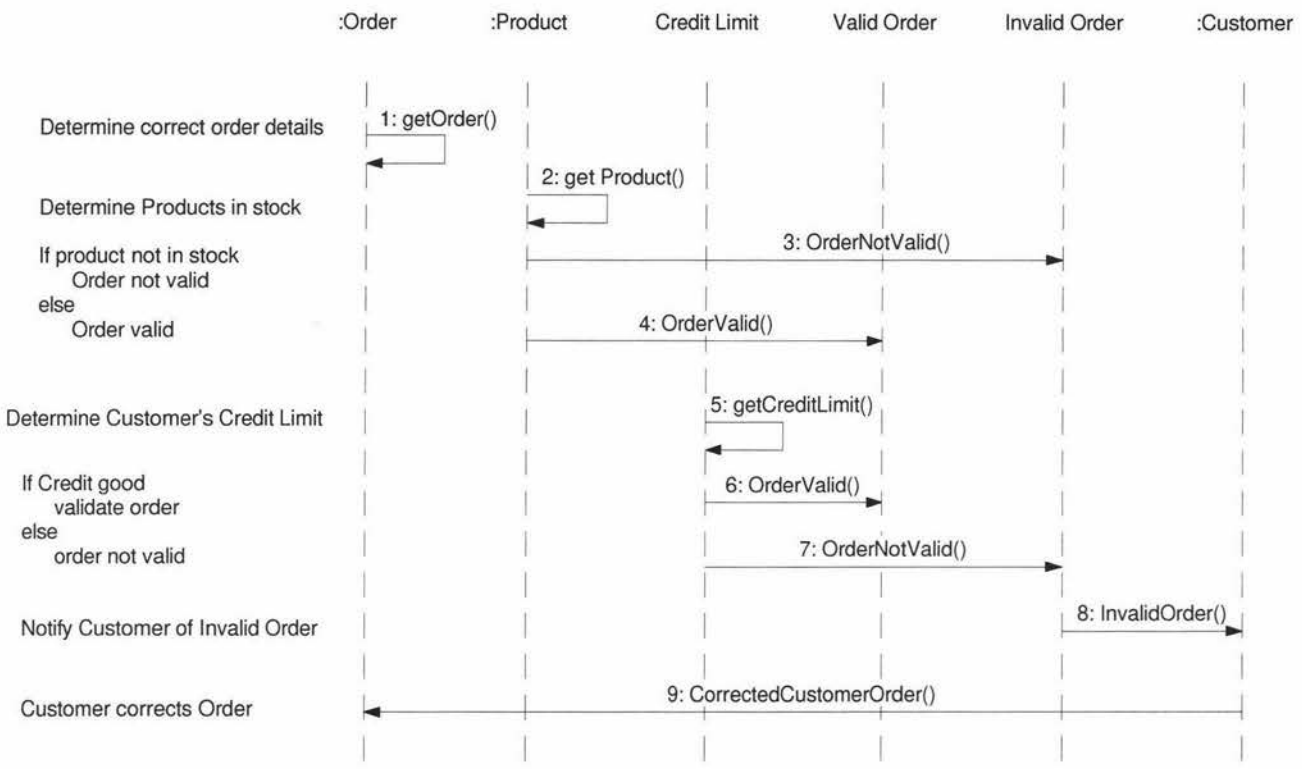

Order
Order Form
Order Mode
Valid Order
Weekly Summary
Order Summary
Daily Summary
Payment
Invalid Order
Credit Limit
Order Document Set
Delivery Plan
Phone Order
Cash
Cheque
Credit Card
Sales Copy
notifies
places
places
Customer
Order Acknowledgement
Accounts Copy
sent to
Order Information Pack
Product
delivered to
Van Copy
requests
Despatch Note
Despatch Instruction
Product Brochure
Existing Customer
New Customer
Prospective Customer

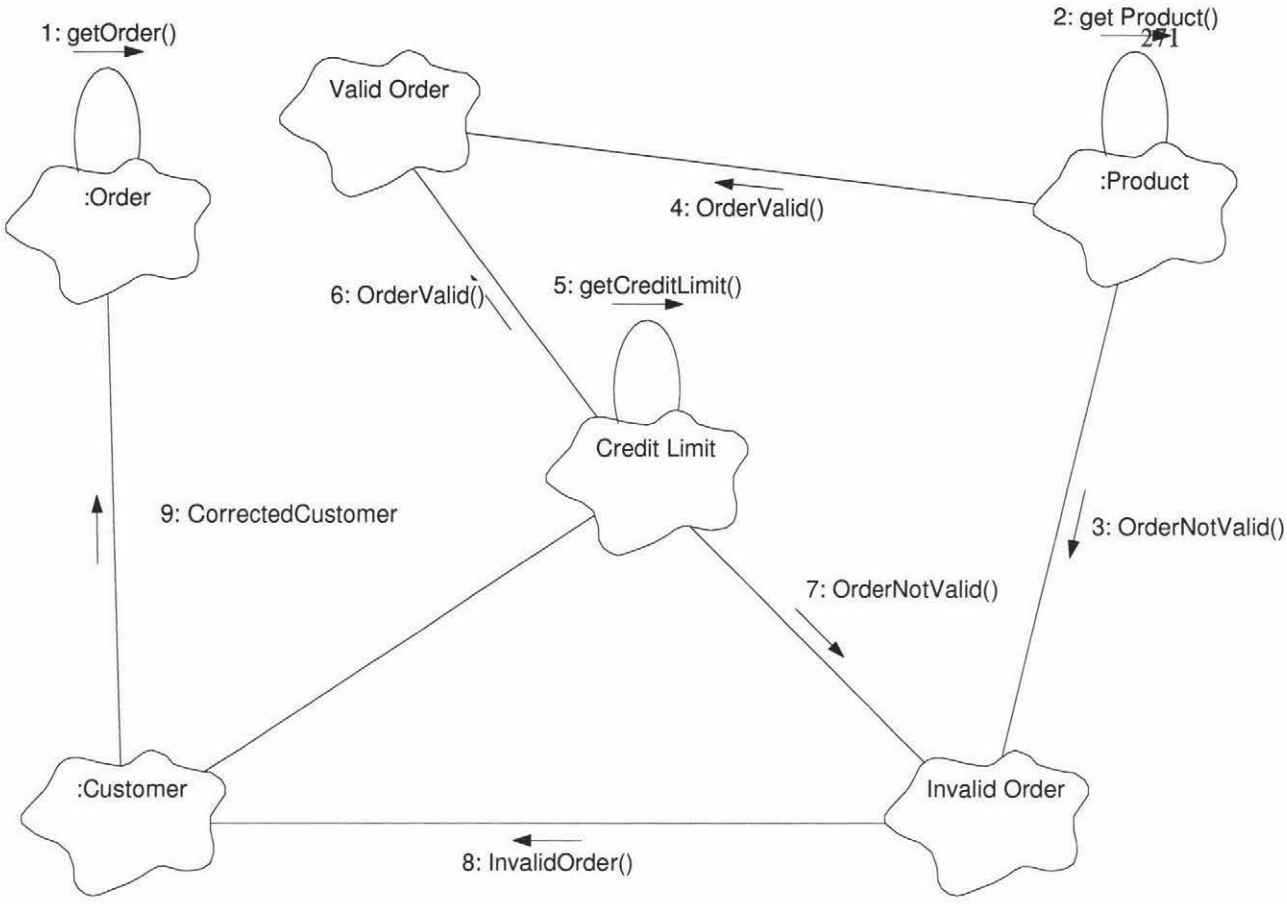

6: LogCustomerDetails()
5: createCustomerID()
:Customer
Product Brochure
2: GetBrochure()
1: OrderRequest
Order Form
3: GetOrderForm()
4: SendOrderPack
:Order Information

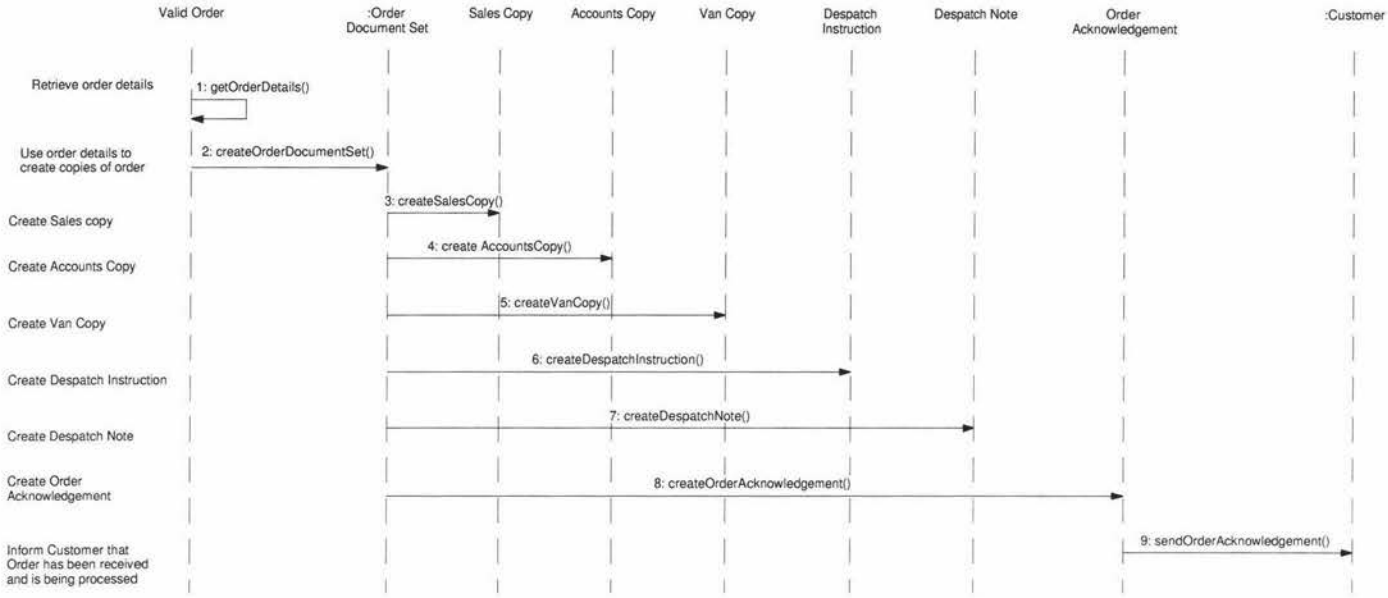

:Customer
:Order Information Pack
Product Brochure
Order Form
Customer requests an order information pack
1: OrderRequest
Get contents of order information pack
2: GetBrochure()
3: GetOrderForm()
Despatch Order Information Pack to the Customer
4: SendOrderPack
If Customer not Existing Customer Assign Customer ID
5: createCustomerID()
6: LogCustomerDetails()
Record the customer's details

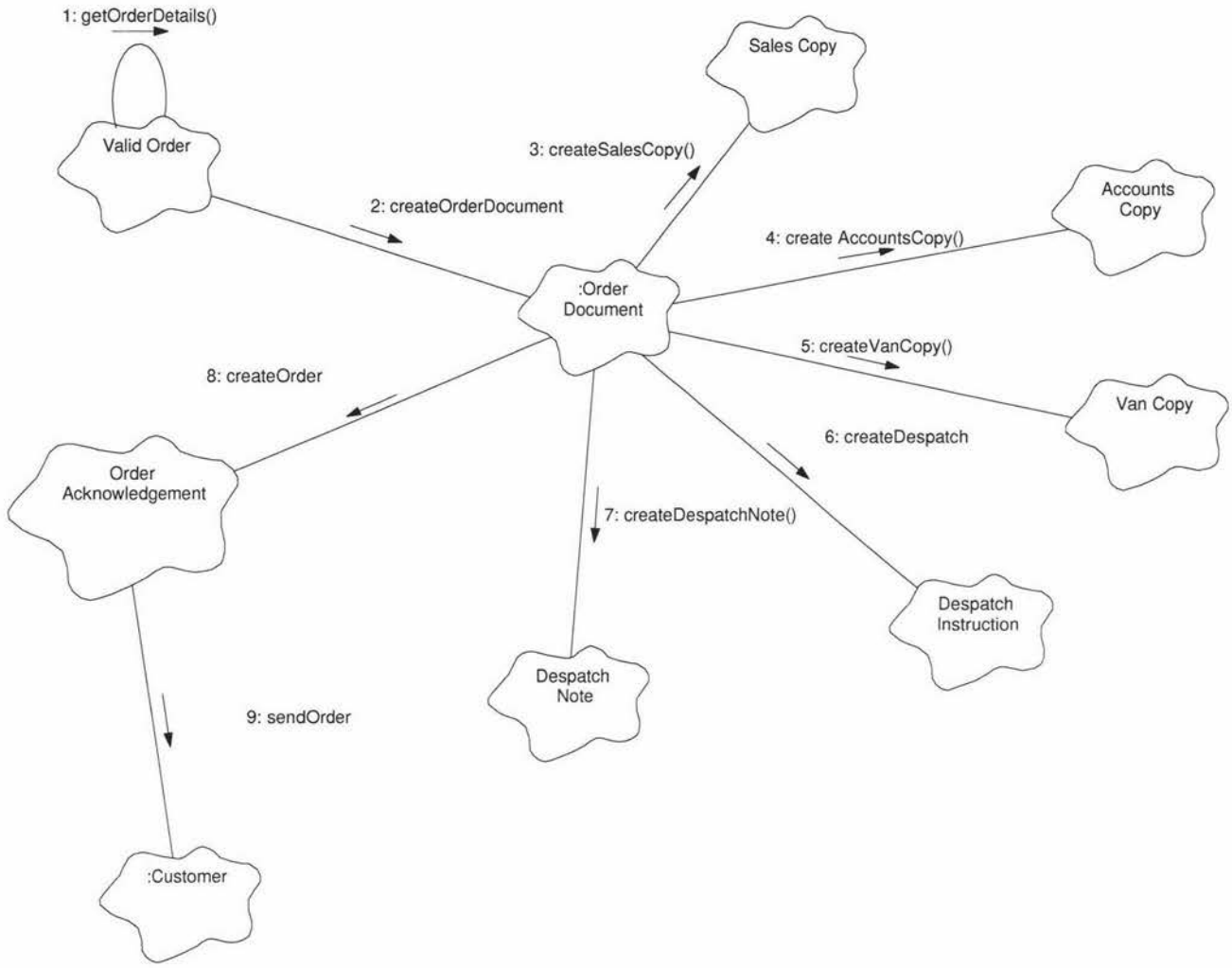

:Customer
Order Form
Phone Order
:Order
Determine Customer Type
1: getCustomerType()
If Existing Customer then
accept PhoneOrder
2: CustomerOrder()
or Order Form
else
3: CustomerOrder()
accept Order Form
4: CustomerOrder()
Record Order placed by Order Form
5: RecordOrder()
6: RecordPhoneOrder()
Record Order placed by Phone
Assign Order Number
7: OrderNumber()

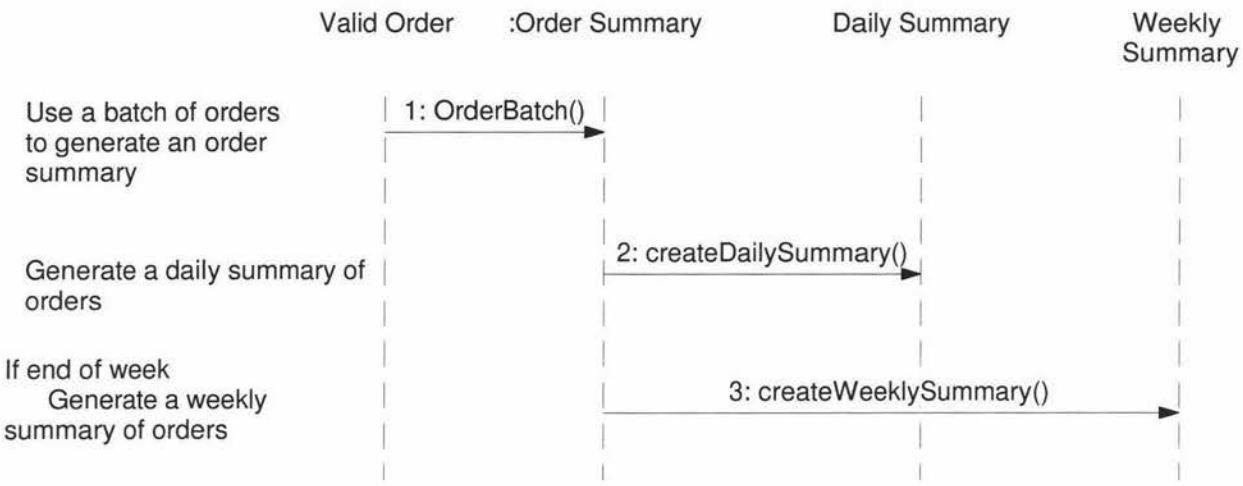

1: getCustomerType()
:Customer
3: CustomerOrder()
4: CustomerOrder()
Order Form
7: OrderNumber()
5: RecordOrder()
2: CustomerOrder()
:Order
6: RecordPhoneOrder()
Phone Order

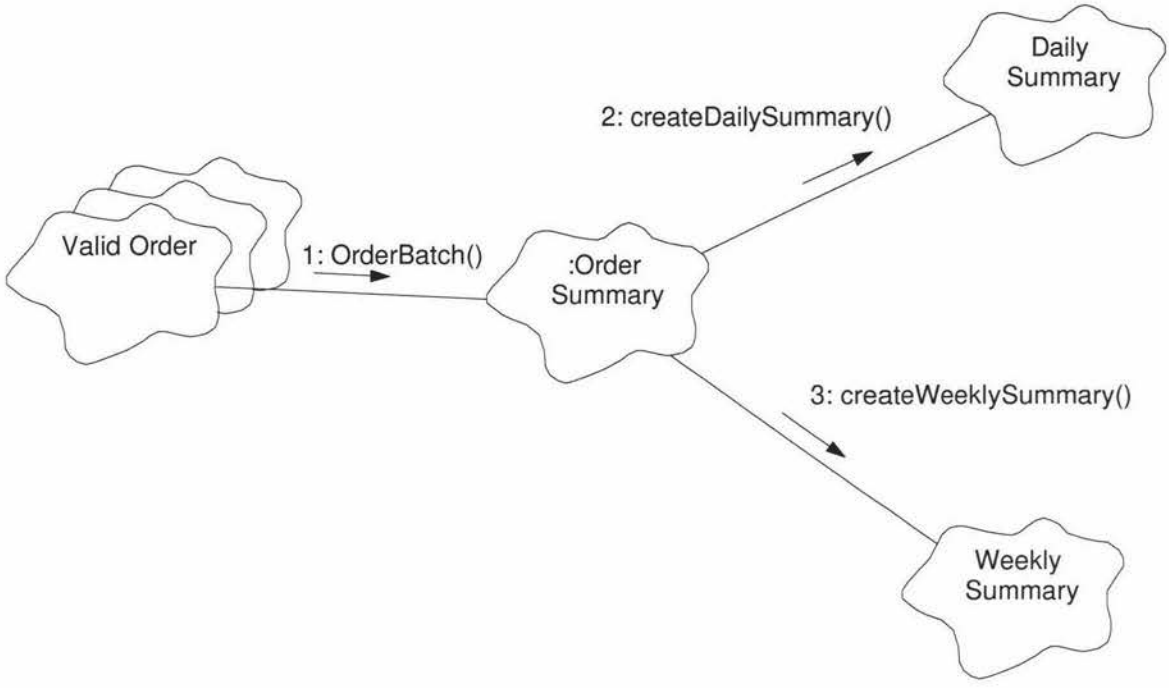

:Order
:Product
Credit Limit
Valid Order
Invalid Order
:Customer
Determine correct order details
1: getOrder()
2: get Product()
Determine Products in stock
3: OrderNotValid()
If product not in stock
Order not valid
else
Order valid
4: OrderValid()
Determine Customer's Credit Limit
5: getCreditLimit()
If Credit good
validate order
else
order not valid
6: OrderValid()
7: OrderNotValid()
Notify Customer of Invalid Order
8: InvalidOrder()
Customer corrects Order
9: CorrectedCustomerOrder()

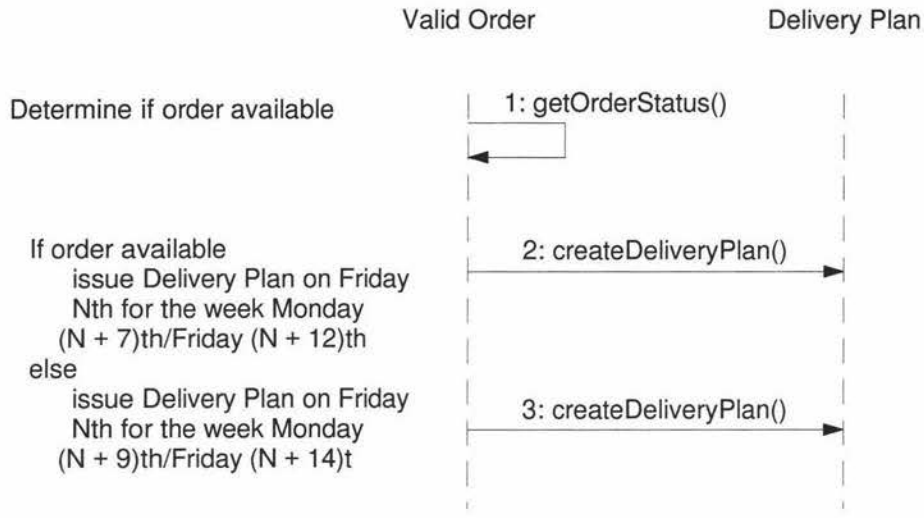

1: getOrder()
2: get Product()
Valid Order
:Order
:Product
4: OrderValid()
6: OrderValid()
5: getCreditLimit()
Credit Limit
9: CorrectedCustomer
3: OrderNotValid()
7: OrderNotValid()
:Customer
Invalid Order
8: InvalidOrder()

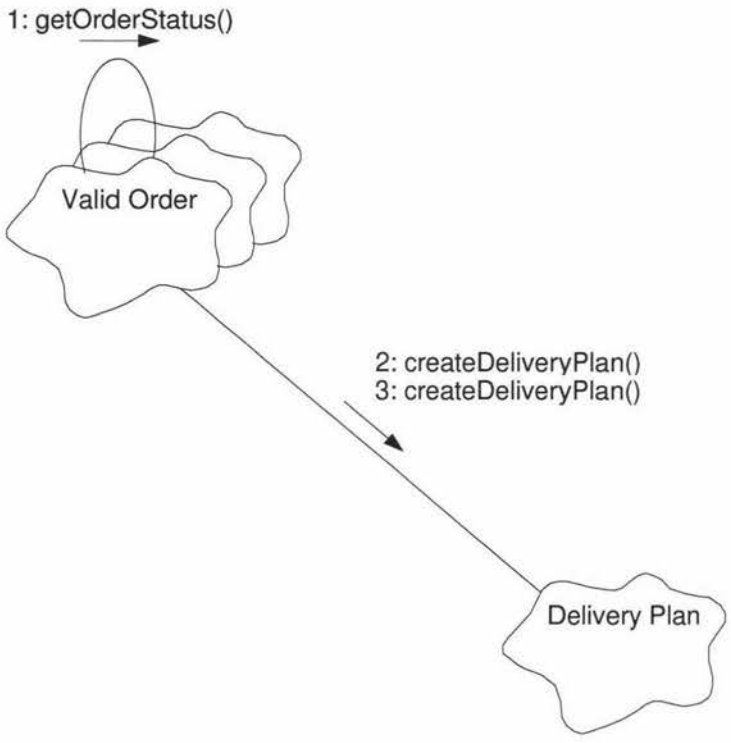

Valid Order
:Order Document Set
Sales Copy
Accounts Copy
Van Copy
Despatch Instruction
Despatch Note
Order Acknowledgement
:Customer
Retrieve order details
1: getOrderDetails()
Use order details to create copies of order
2: createOrderDocumentSet()
3: createSalesCopy()
Create Sales copy
4: create AccountsCopy()
Create Accounts Copy
5: createVanCopy()
Create Van Copy
6: createDespatchInstruction()
Create Despatch Instruction
7: createDespatchNote()
Create Despatch Note
Create Order Acknowledgement
8: createOrderAcknowledgement()
9: sendOrderAcknowledgement()
Inform Customer that Order has been received and is being processed

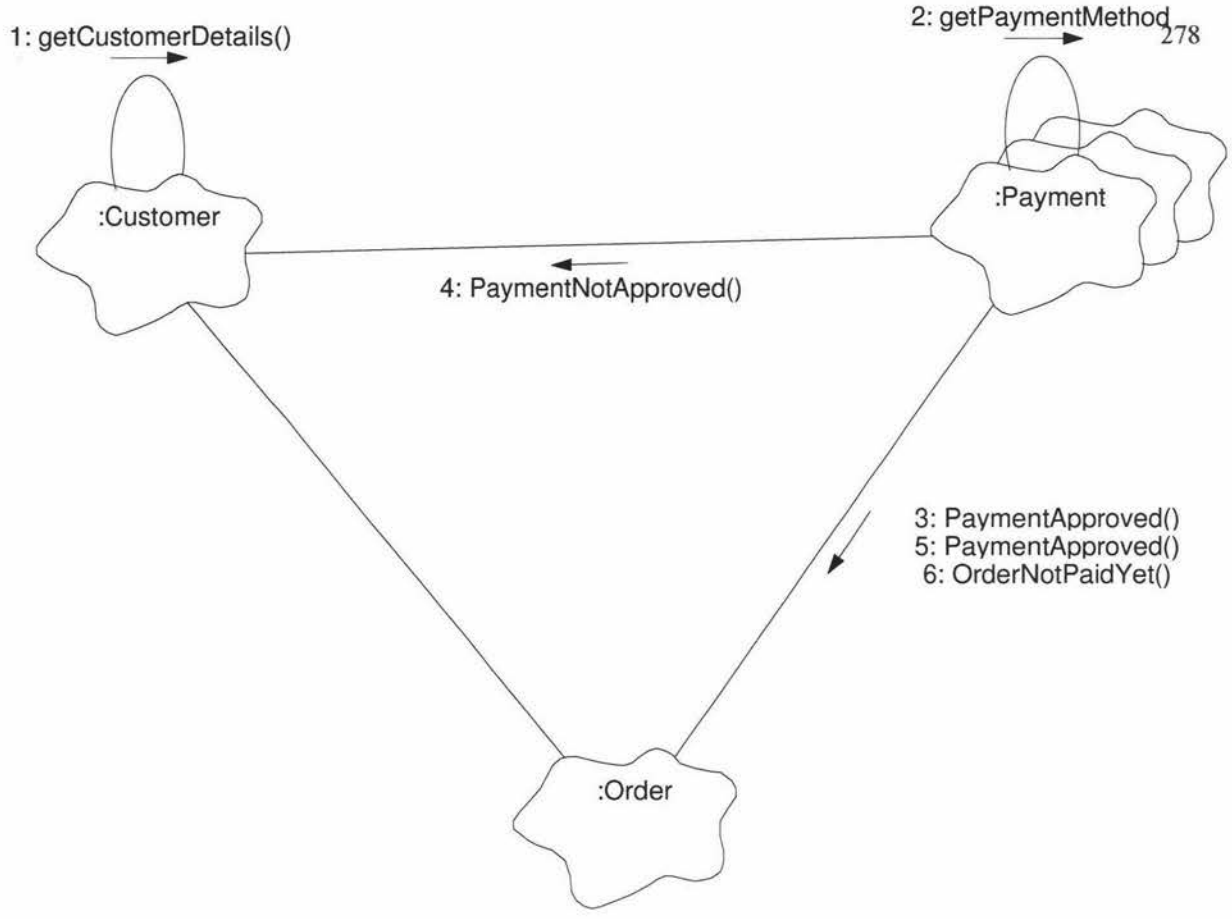

1: getOrderDetails()
Valid Order
Sales Copy
3: createSalesCopy()
2: createOrderDocument
Accounts Copy
4: create AccountsCopy()
:Order Document
5: createVanCopy()
8: createOrder
Van Copy
6: createDespatch
Order Acknowledgement
7: createDespatchNote()
Despatch Instruction
Despatch Note
9: sendOrder
:Customer

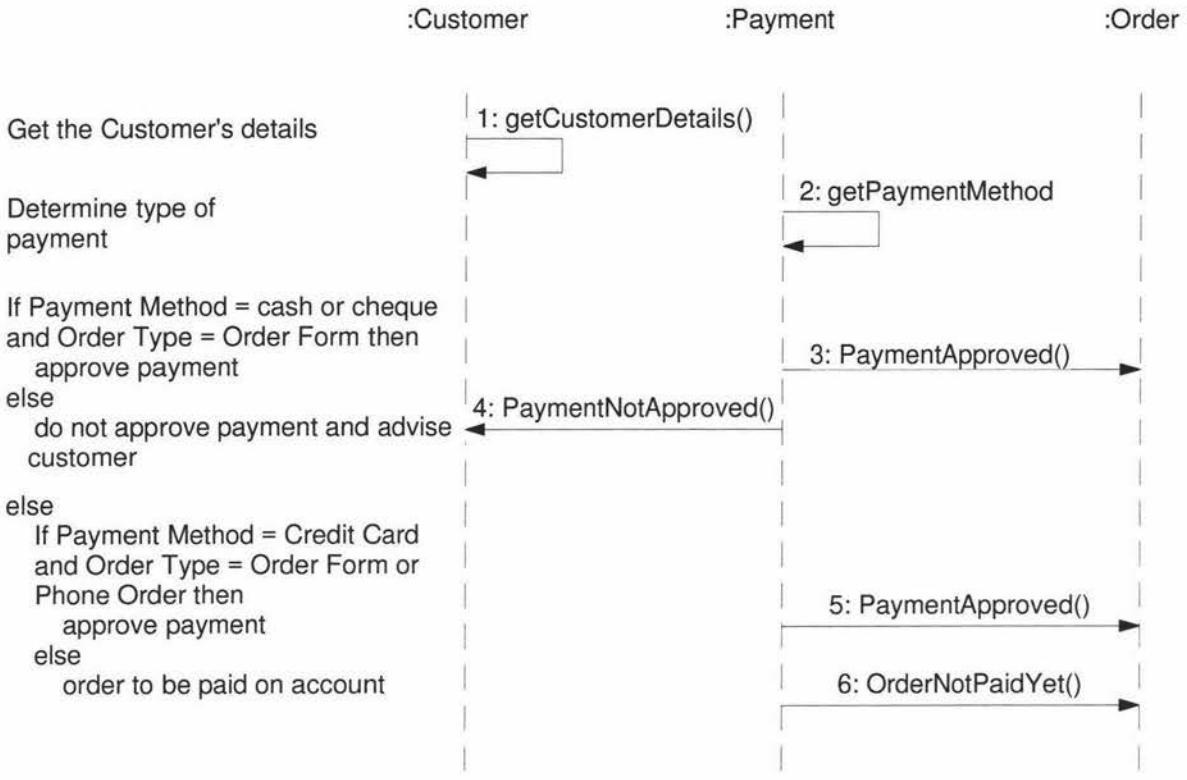

Valid Order
:Order Summary
Daily Summary
Weekly Summary
Use a batch of orders to generate an order summary
1: OrderBatch()
Generate a daily summary of orders
2: createDailySummary()
If end of week
Generate a weekly summary of orders
3: createWeeklySummary()


Daily
Summary
2: createDailySummary()
Valid Order
1: OrderBatch()
:Order
Summary
3: createWeeklySummary()
Weekly
Summary


Valid Order
Delivery Plan
Determine if order available
1: getOrderStatus()
If order available
issue Delivery Plan on Friday
Nth for the week Monday
(N + 7)th/Friday (N + 12)th
else
issue Delivery Plan on Friday
Nth for the week Monday
(N + 9)th/Friday (N + 14)t
2: createDeliveryPlan()
3: createDeliveryPlan()


1: getOrderStatus()
Valid Order
2: createDeliveryPlan()
3: createDeliveryPlan()
Delivery Plan


1: getCustomerDetails()
2: getPaymentMethod
:Customer
:Payment
4: PaymentNotApproved()
3: PaymentApproved()
5: PaymentApproved()
6: OrderNotPaidYet()
:Order


:Customer
:Payment
:Order
Get the Customer's details
1: getCustomerDetails()
Determine type of payment
2: getPaymentMethod
If Payment Method = cash or cheque and Order Type = Order Form then
approve payment
else
do not approve payment and advise customer
3: PaymentApproved()
4: PaymentNotApproved()
else
If Payment Method = Credit Card and Order Type = Order Form or Phone Order then
approve payment
else
order to be paid on account
5: PaymentApproved()
6: OrderNotPaidYet()

Class name:

Accounts Copy

Category: Order Processing System
Documentation:
Accounts Copy is a copy of the original order which is kept in the Accounts department for billing purposes.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Cash

Category: Order Processing System
Documentation:
Cash is a medium that the Customer may choose so that they can pay for their order at the time the order is placed. A customer cannot pay by cash at the time that the order is placed if the order is a phone order.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Payment
Public Uses:
Customer

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Cheque

Category: Order Processing System
Documentation:
Cheque is a medium that the Customer uses to pay for goods at the time that the order is placed. A cheque may be a personal cheque or a bank cheque. A customer cannot pay by cheque at the time the order is placed, if the order is a phone order.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Payment
Public Uses:
Customer

State machine: No
Concurrency: Sequential

Persistence: Transient

Class name:

Credit Card

Category: Order Processing System
Documentation:

Credit Card is a medium that the Customer uses to pay for goods at the time that the order is placed. A valid credit card can be used to pay for both orders placed by phone and with an order form. If the customer pays by credit card then they MUST provide the following details:
1. Formal Name on the Credit Card (i.e. the cardholder's name).
2. The type/brand of credit card (eg: Visa, Mastercard, American Express, Diners Club, etc).
3. The name of the bank that the credit card is issued by.
4. The card number
5. The card's expiry date (this must be current)
6. A signature must be supplied by the customer at some stage in the ordering process, in order to show that they have authorised the transaction. A signature is obtained using the following methods:
6a. A signature on the order form if the order is placed in this manner
6b. A signature on the Van Copy of the order form when the goods are delivered.

Failure by the customer to meet the above conditions will result in the order being invalidated.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Payment
Public Uses:
Customer

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Credit Limit

Category: Order Processing System
Documentation:

The credit limit determines if a customer is able to pay for products that they have ordered. The credit limit also shows the customer's outstanding debt and credit limit.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:
Customer

Operations:
getCreditLimit (Customer ID, Credit Limit)
OrderNotValid (Customer ID, Order Number)
OrderValid (Customer ID, Order Number)

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Customer

Category: Order Processing System
Documentation:
Customer is the person or organisation who places an order or requests information from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Associations:

Order Information•
Pack requests
A customer may request an order information pack from the Trusty Furniture Company in order to receive a product brochure and an order form.

Order Form places
A customer may place an order with the Trusty Furniture Company by using an order form.

Phone Order places
An existing customer may place an order with the Trusty Furniture Company over the telephone.

Public Interface:
Operations:

CorrectedCustomerOrder (Order Details, Order Quantity)
createCustomerID (Customer First Name, Customer Last Name, Customer Phone Number, C
CustomerOrder (Order Details, Order Quantity)
getCustomerDetails (Customer First Name, Customer Last Name, Customer Address, Custor
getCustomerType (Customer Type)
LogCustomerDetails (Customer First Name, Customer Last Name, Customer Address, Custo
OrderRequest (Customer First Name, Customer Last Name, Customer Address, Customer P

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Daily Summary

Category: Order Processing System
Documentation:
Daily Summary is a summary of the orders received by the Trusty Furniture Company that is produced on a daily basis. The summary is analysed by product.

Export Control: Public

Cardinality: n
Hierarchy:
Superclasses: Order Summary
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Delivery Plan

Category: Order Processing System
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Uses:
Valid Order

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Despatch Instruction

Category: Order Processing System
Documentation:
Despatch Instruction is a copy of the order which is used to check which items are actually loaded at the time of despatch. The items which are loaded copy through to the Despatch Note and the Van Copy. Despatch Instruction is initially sent to the Despatch Manager.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Despatch Note

Category: Order Processing System
Documentation:
Despatch Note is sent to the Sales department by the Despatch manager. Despatch Note notes any shortages from the original order. At this point the Despatch Note is forwarded to the Customer notifiying them of any shortages. Despatch Note also uses the Despatch Instruction to note which items were loaded and sent to the Customer at despatch time.

Export Control: Public
Cardinality: n
Hierarchy:

Superclasses: Order Document Set
Associations:

Customer delivered to
A Despatch note is delivered to a Customer to notify them if there are any shortages.

Public Uses:

Despatch Instruction

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Existing Customer

Category: Order Processing System
Documentation:
This is a customer who has previously purchased goods from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Customer
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Invalid Order

Category: Order Processing System
Documentation:
Invalid Order is a list of items for purchase by a Customer that has not been approved. Reasons for declining an order include: an order form that has been incorrectly filled out, failure to obtain credit approval, and orders for products that have been discontinued/are out of stock or have had a change in price.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order
Associations:

Customer notifies
Notifies the Customer if there is a problem with their order.

Public Interface:
Operations:

InvalidOrder (Customer ID, Reason for Invalid Order, Order Number)

State machine: No

Concurrency: Sequential
Persistence: Transient

Class name:
New Customer

Category: Order Processing System
Documentation:
This is a person who has not ordered from the Trusty Furniture Company before.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Customer
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Order

Category: Order Processing System
Documentation:
Order is a list of items selected by the customer for purchase.

Export Control: Public
Cardinality: 1
Hierarchy:
Superclasses: none
Public Uses:
Credit Limit
Order Mode

Public Interface:
Has-A Relationships:
Payment
Product

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Order Acknowledgement

Category: Order Processing System
Documentation:
Order Acknowledgement is a copy of the order which is sent to the customer to notify them that their order has been recieved, approved and is currently being processed.

Export Control: Public
Cardinality: n

Hierarchy:
Superclasses: Order Document Set
Associations:

Customer sent to
An order acknowledgement form is sent to all customers who have a valid order. The order acknowledgement form show that the order has been received, validate, and is currently being processed.

Public Interface:
Operations:

sendOrderAcknowledgement (Customer ID, Customer First Name, Customer Last Name, Cus

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Order Document Set

Category: Order Processing System
Documentation:
Order Document Set is a collection of copies of the original order which are made after the order has been approved.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Valid Order
Public Interface:
Operations:

createAccountsCopy (Order Number, Customer ID, Product Quantity, Product Description)
createDespatchInstruction (Customer ID, Order Number, Product Description, Product Quantit
createDespatchNote (Customer ID, Order Number, Product Description, Product Quantity)
createOrderAcknowledgement (Customer ID, Order Number, Product Description, Product Qu
createSalesCopy (Customer ID, Order Number, Product Description, Product Quantity)
createVanCopy (Customer ID, Order Number, Product Quantity, Product Description)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Order Form

Category: Order Processing System
Documentation:
A form used by the customer to detail their order.

Export Control: Public
Cardinality: 1
Hierarchy:
Superclasses: Order Mode
Public Interface:

Operations:

RecordOrder (Customer First Name, Customer Last Name, Customer Address, Customer Pho

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Order Information• Pack

Category: Order Processing System
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:

Product Brochure
Order Form

Operations:

GetBrochure ( )
GetOrderForm ( )
SendOrderPack ( )

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Order Mode

Category: Order Processing System
Documentation:

Order Mode is used to determine the means by which an order has been placed with the Trusty Furniture Company. An order may be placed by Order Form or by Phone if the Customer has already purchased goods from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Order Summary

Category: Order Processing System
Documentation:

Order Summary is a summary of the orders recieved by the Trusty Furniture Company. The

summary is analysed by product.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Uses:
Valid Order

Public Interface:
Operations:
createDailySummary (Order Number, Product Description, Product Quantity, Date)
createWeeklySummary (Order Number, Product Description, Product Quantity, Date)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Payment

Category: Order Processing System
Documentation:
Payment determines how a customer is going to pay for their order if they decide to pay at the time that they place their order.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
getPaymentMethod (Type of Payment)
PaymentApproved (Customer ID, Payment OK)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Phone Order

Category: Order Processing System
Documentation:
Phone Order is an order for products placed by a customer. Only existing customers of the Trusty Furniture Company may place an order by phone, all other customers must place their order using the supplied order form.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Mode
Public Interface:
Operations:

RecordPhoneOrder (Customer Phone Number, Customer First Name, Customer Last Name,

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Product

Category: Order Processing System
Documentation:
Product is an item produced by the Trusty Furniture Company for sale.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
getProduct (Product Name, Product Quantity, Products in Stock)
OrderNotValid (Products in Stock, Credit Limit)
OrderValid (Products in Stock, Credit Limit)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Product Brochure

Category: Order Processing System
Documentation:
Is a catalogue of the Trusty Furniture Company's product line plus the prices of each product.

Export Control: Public
Cardinality: 1
Hierarchy:
Superclasses: none
Public Interface:
Has-A Relationships:
Product

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Prospective Customer

Category: Order Processing System
Documentation:
This is a customer who has expressed an interested in the range of products sold by the Trusty Furniture Company. In order to find out more information about the products available for sale they

have requested an order information pack. A prospective customer may or may not purchase goods from the Trusty Furniture Company.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Customer
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Sales Copy

Category: Order Processing System
Documentation:
Sales Copy is a copy of the order form that is sent to the salesperson who made the sale via the sales department.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Valid Order

Category: Order Processing System
Documentation:
Valid Order is a list of items for purchase by a Customer that has been approved.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order
Public Interface:
Operations:
createDeliveryPlan (Customer ID, Order Number)
createOrderDocumentSet (Customer ID, Product Description, Product Quantity, Order Numbe
getOrderDetails (Product Description, Product Quantity, Customer ID, Order Number)
getOrderStatus (Customer ID, Order Number)
OrderBatch (Date)

State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Van Copy

Category: Order Processing System
Documentation:
Van Copy is a copy of the order which goes with the driver to obtain the customers acknowledgement of receipt. Van copy also uses the Despatch Instruction to note which items were loaded and sent to the Customer at despatch time.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Document Set
Public Uses:
Despatch Instruction

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Weekly Summary

Category: Order Processing System
Documentation:
Weekly Summary is a summary of the orders received by the Trusty Furniture Company that is produced on a weekly basis. The summary is analysed by product.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Order Summary
State machine: No
Concurrency: Sequential
Persistence: Persistent

APPENDIX G: Object-Oriented Model Metric Results

Appendix G contains the following object-oriented metric results from the Trusty Furniture Company's Order Processing System:

- One table showing the MOOSE results before the changes were made

- One table showing the MOOSE results after the changes were made

MOOSE Results Before Changes

	WMC	DIT	NOC	CBO	RFC	LCOM
Customer	6	0	3	7	10	4
Prospective Customer	0	1	0	0	0	0
New Customer	0	1	0	0	0	0
Existing Customer	0	1	0	0	0	0
Order Information Pack	3	0	0	3	4	0
Product Brochure	0	0	0	2	0	0
Product	3	0	0	2	4	3
Order Form	1	1	0	2	2	1
Order Mode	0	0	2	1	0	0
Phone Order	1	1	0	1	1	1
Order	0	0	2	3	0	0
Invalid Order	1	1	0	1	2	1
Valid Order	5	1	1	2	13	3
Credit Limit	3	0	0	2	4	3
Order Summary	2	0	2	1	2	2
Daily Summary	0	1	0	0	0	0
Weekly Summary	0	1	0	0	0	0
Delivery Plan	0	0	0	1	0	0
Order Document Set	6	2	6	0	7	6
Sales Copy	0	3	0	0	0	0
Accounts Copy	0	3	0	0	0	0
Van Copy	0	3	0	1	0	0
Despatch Instruction	0	3	0	2	0	0
Despatch Note	0	3	0	2	0	0
Order Acknowledgement	1	3	0	1	0	1
Total	**32**	**29**	**16**	**34**	**49**	**25**

MOOSE Results After Changes

	WMC	DIT	NOC	CBO	RFC	LCOM
Customer	7	0	3	10	11	5
Prospective Customer	0	1	0	0	0	0
New Customer	0	1	0	0	0	0
Existing Customer	0	1	0	0	0	0
Order Information Pack	3	0	0	3	4	0
Product Brochure	0	0	0	2	0	0
Product	3	0	0	2	4	3
Order Form	1	1	0	2	2	1
Order Mode	0	0	2	1	0	0
Phone Order	1	1	0	1	1	1
Order	0	0	2	4	0	0
Invalid Order	1	1	0	1	2	1
Valid Order	5	1	1	2	13	3
Credit Limit	3	0	0	2	4	3
Order Summary	2	0	2	1	2	2
Daily Summary	0	1	0	0	0	0
Weekly Summary	0	1	0	0	0	0
Delivery Plan	0	0	0	1	0	0
Order Document Set	6	2	6	0	7	6
Sales Copy	0	3	0	0	0	0
Accounts Copy	0	3	0	0	0	0
Van Copy	0	3	0	1	0	0
Despatch Instruction	0	3	0	2	0	0
Despatch Note	0	3	0	2	0	0
Order Acknowledgement	1	3	0	1	0	1
Payment	4	0	3	1	3	2
Cash	0	1	0	1	0	0
Cheque	0	1	0	1	0	0
Credit Card	0	1	0	1	0	0
Total	**37**	**32**	**19**	**42**	**53**	**28**

APPENDIX H: Object-Oriented Model Validation Results

Appendix H contains the following object-oriented validation results from the Trusty Furniture Company's Order Processing System:

- One table showing the MOOSE results from Booch's (1994) Ordering System

- One table showing a summary of the classes used in this study and in Booch (1994)

MOOSE Results from Booch's (1994) Ordering System

	WMC	DIT	NOC	CBO	RFC	LCOM
Customer	2	0	0	2	5	
CustomerRecord	0	0	0	2	0	
Order	0	0	0	5	0	
OrderAgent	12	0	0	2	15	Not
PackingOrder	2	0	0	2	3	Able
ProductRecord	0	0	0	2	0	To Be
StockPerson	5	0	0	2	5	Calculated
Supplier	0	0	0	2	0	
SupplierRecord	0	0	0	1	0	
Total	**21**	**0**	**0**	**20**	**28**	

Summary of the Classes Used in this Study and in Booch (1994, p.390)

Trusty Furniture Company (Before)	*Trusty Furniture Company (After)*	*Booch (1994)*
Customer	Customer	Customer
Prospective Customer	Prospective Customer	CustomerRecord
New Customer	New Customer	Order
Existing Customer	Existing Customer	OrderAgent
Order Information Pack	Order Information Pack	PackingOrder
Product Brochure	Product Brochure	ProductRecord
Product	Product	StockPerson
Order Form	Order Form	Supplier
Order Mode	Order Mode	SupplierRecord
Phone Order	Phone Order	
Order	Order	
Invalid Order	Invalid Order	
Valid Order	Valid Order	
Credit Limit	Credit Limit	
Order Summary	Order Summary	
Daily Summary	Daily Summary	
Weekly Summary	Weekly Summary	
Delivery Plan	Delivery Plan	
Order Document Set	Order Document Set	
Sales Copy	Sales Copy	
Accounts Copy	Accounts Copy	
Van Copy	Van Copy	
Despatch Instruction	Despatch Instruction	
Despatch Note	Despatch Note	
Order Acknowledgement	Order Acknowledgement	
	Payment	
	Cash	
	Cheque	
	Credit Card	

APPENDIX I: Comparison of CBO Results on an Individual Basis

Appendix I contains the following result from the Trusty Furniture Company's Order Processing System:

- One table showing a comparison between the CBO results of the structured system and the object-oriented system using equivalent entities/classes

Comparison of CBO Results on an Individual Basis

	CBO Structured (Before)	CBO Structured (After)	CBO OO (Before)	CBO OO (After)
Customer	1	1	7	10
Order	2	3	2	3
Payment		1		1
Product	2	2	2	2
Product Brochure	1	1	2	2
Total	**6**	**8**	**13**	**18**

BIBLIOGRAPHY

Arthur, L.J. (1988). Software Evolution: The Software Maintenance Challenge. New York: John Wiley & Sons, Inc.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D, and Zdonik, S. (1992). The object-oriented database system manifesto. In F. Bancilhon, C. Delobel, and P. Kanellankis (eds.), Building an object-oriented database system: The story of O_2 (pp. 3-20). San Mateo: Morgan Kaufmann Publishers, Inc.

Basili, V.R., and Rombach, H.D. (June, 1988). The TAME project: Towards improvement-oriented software environments. IEEE Transactions on Software Engineering. 14, (6). 759-773.

Bingham, J., and Davies, G. (1992). Systems Analysis. Houndmills: The Macmillan Press Ltd.

Booch, G. (1994). Object-oriented analysis and design with applications. (2nd ed.) Redwood City: The Benjamin/Cummings Publishing Company, Inc.

Booch, G. (February, 1986). Object-oriented development. IEEE Transactions on Software Engineering. SE-12, (2). 211-221.

Brinkworth, J.W.O. (1992). Software Quality Management: a pro-active approach. Hemel Hempstead: Prentice Hall International (UK) Ltd.

Burch, J.G. (1992). Systems analysis, design, and implementation. Boston: boyd & fraser publishing company.

Card, D.N., and Glass, R.L. (1990). Measuring software design quality. Englewood Cliffs: Prentice-Hall, Inc. 005. Car

Cardelli, L. (1990). A semantics of multiple inheritance. In S.B. Zdonik and D. Maier (eds.), Readings in object-oriented database systems (pp. 59-83). San Mateo: Morgan Kaufmann Publishers, Inc.

Cattell, R. (Ed.). (1994). The object database standard: ODMG-93 Release 1.1. San Francisco: Morgan Kaufmann Publishers, Inc.

Cattell, R.G.G. (1991). Object data management: Object-oriented and extended relational database systems. Reading: Addison-Wesley Publishing Company.

Chen, J-Y., and Lu, J-F. (April, 1993). A new metric for object-oriented design. Information and Software Technology. 35, (4). 232-240.

Chidamber, S.R., and Kemerer, C.F. (June, 1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering. 20, (6). 476-493.

Chidamber, S.R., and Kemerer, C.F. (1991). Towards a metrics suite for object oriented design. In Proceedings 6th ACM Conference of Object Oriented Programming, Systems, Language, and Applications (OOPSLA) (pp. 197-211). Phoenix: ACM.

Churcher, N.I., and Shepperd, M.J. (April, 1995). Towards a conceptual framework for object oriented software metrics. ACM SIGSOFT Software Engineering Notes. 20, (2). 69-76.

Clapp, J.A., Cerino, D.A., Dziegiel, R.J. Jr., Peng, W.W., Stanten, S.F., and Wallace, D.R. (1995). Software quality control, error analysis, and testing. Park Ridge: Noyes Data Corporation.

Coad, P., & Yourdon, E. (1991). Object-oriented analysis. (2nd ed.) Englewood Cliffs: Prentice-Hall, Inc.

DeMarco, T. (1979). Structured Analysis and System Specification. Englewood Cliffs: Prentice-Hall, Inc.

Deubler, H-H., and Koestler, M. (November, 1994). Introducing object orientation into large and complex systems. IEEE Transactions on Software Engineering. 20, (11). 840-848.

Eid, M., and Rose, E. (January, 1996) A comparison of software development methodologies with respect to maintainability. Palmerston North: Massey University.

Fertuck, L. (1995). System Analysis & Design with Modern Methods. Dubuque: Business and Educational Technologies.

Fichman, R.G., and Kemerer, C.F. (October, 1992). Object-oriented and conventional analysis and design methodologies: Comparison and critique. IEEE Computer. 22-39.

Fournier, R. (1991). Practical Guide To Structured System Development And Maintenance. Englewood Cliffs: Prentice-Hall, Inc.

Gane, C., and Sarson, T. (1979). Structured Systems Analysis: Tools and Techniques. Englewood Cliffs: Prentice-Hall, Inc.

Gillies, A.C. (1992). Software Quality: Theory and Management. London: Chapman & Hall.

Grady, R.B., and Caswell, D.L. (1987). Software Metrics: Establishing a Company-Wide Program. Englewood Cliffs: Prentice-Hall, Inc.

Harrison, W., Magel, K., Kluczny, R., and DeKock, A. (September, 1982). Applying software complexity metrics to program maintenance. IEEE Computer. 15, (9). 65-79.

Henry, S., and Humphrey, M. (1993). Comparison of an object oriented programming language to a procedural programming language for effectiveness in program maintenance. Journal of Object Oriented Programming. 6, (3). 41-49.

Henry, S., and Kafura, D. (1981). Software structure metrics based on information flow. IEEE Transactions on Software Engineering. 7, (5). 510-518.

Jackson, B. (1996). 57.221 Information Systems Analysis: Project Guidelines and Scenario. Palmerston North: Massey University.

Kim, W. (1991). Introduction to object-oriented databases. Cambridge: The MIT Press.

Korson, T., & McGregor, J.D. (1990). Understanding object-oriented: A unifying paradigm. Communications of the ACM, 33 (9), 40-60.

Levitin, A.V. (1986). How to measure software size, and how not to. In Proceedings COMPSAC '86 (pp. 314-318). Chicago: IEEE.

Lientz, B.P. and Swanson, E.B. (1980). Software Maintenance Management. Reading: Addison-Wesley Publishing Company.

Low, G.C., Henderson-Sellers, B., and Han, D. (1995). Comparison of object-oriented and traditional systems development issues in distributed environments. Information & Management. 28, (5). 327-340.

Martin, J., and McClure, C. (1983). Software Maintenance: The Problem and its Solutions. Englewood Cliffs: Prentice Hall, Inc.

Mason, D., and Willcocks, L. (1994). Systems Analysis, Systems Design. Orchards: Alfred Waller Ltd, Publishers.

McCabe, T.J. (December, 1976) A complexity measure. IEEE Transactions On Software Engineering. SE-2, (4). 308-320.

McCabe, T.J., and Schulmeyer, G.G. (1983). System testing aided by structured analysis (A practical experience). In T.J. McCabe (ed.), Structured Testing (pp. 51-56). Silver Spring: IEEE Computer Society Press.

McMenamin, S., and Palmer, J. (1984). Essential Systems Analysis. New York: YOURDON Press.

Meyer, B. (1981). Towards a two dimensional programming environment. In Readings in artificial intelligence (p.178). Palo Alto: Tioga.

Möller, K.H., and Paulish, D.J. (1993). Software Metrics: A practitioner's guide to improved product development. London: Chapman & Hall.

Nierstrasz, O. (1989). A survey of object-oriented concepts. In W. Kim and F.H. Lochovsky (eds.), Object-Oriented Concepts, Databases, and Applications (pp. 3-21). New York: ACM Press.

Perry, W.E. (1981). Managing systems maintenance. Wellesley: Q.E.D. Information Sciences, Inc.

Pickard, M.M., and Carter, B.D. (July, 1993). Maintainability: What is it and how do we measure it? ACM SIGSOFT Software Engineering Notes. 18, (3). A36-A39.

Rose, E. (1995). 57.494 Lecture Notes: OO vs Structured Methods. Palmerston North: Massey University.

Sallis, P.J., Tate, G., and MacDonell, S.G. (1995). Software engineering: practice, management, improvement. Sydney: Addison-Wesley Publishing Company.

Schneidewind, N.F. (March, 1987). The state of software maintenance. IEEE Transactions on Software Engineering. SE-13, (3). 303-310.

Shafer, D., and Taylor, D. (1993). Transforming the enterprise through COOPERATION: an object-oriented solution. Englewood Cliffs: Prentice Hall, Inc.

Sharble, R.C., and Cohen, S.S. (April, 1993). The object-oriented brewery: A comparison of two object-oriented development methods. ACM SIGSOFT Software Engineering Notes. 18, (2). 60-73.

Sharon, D. (January, 1996). Meeting the challenge of software maintenance. IEEE Software. 13, (1). 122-125.

Shlaer, S., and Mellor, S.J., (1989). An object-oriented approach to domain analysis. ACM SIGSOFT, 14 (5), 66-77.

Stark, M. (1993). Impacts of object-oriented technologies: Seven years of SEL studies. ACM SIGPLAN Notices, 28 (10), 365-373.

Swanson, E.B. (1976). The dimensions of maintenance. In B.P. Lientz and E.B. Swanson, (1980). Software maintenance management. Reading: Addison-Wesley Publishing Company, Inc.

Teague, L., and Pidgeon, C. (1985). Structured Analysis Methods for Computer Information Systems. Chicago: Science Research Associates.

Whiddett, D., Dasari, S., and Woodfield, T. (1995). Comparisons of development methodologies: A study of the use of object-oriented and structured analysis techniques. New Zealand Journal of Computing, 6 (1A), 107-114.

White, I. (1994). Using the Booch method: A Rational approach. Redwood City: The Benjamin/Cummings Publishing Company, Inc.

van Genuchten, M., Brethouwer, van den Boomen, and Heemstra, F. (August, 1992). Empirical study of software maintenance. Information and Software Technology. 34, (8). 507-512.

Wilde, N., and Huitt, R. (December, 1992). Maintenance support for object-oriented programs. IEEE Transactions on Software Engineering. 18, (12). 1038-1044.

Yau, S.S., and Collofello, J.S. (September, 1985). Design stability measures for software maintenance. IEEE Transactions on Software Engineering. SE-11, (9). 849-856.

Zdonik, S.B., & Maier, D. (1990). Fundamentals of object-oriented databases. In S.B. Zdonik and D. Maier (eds.), Readings in object-oriented database systems (pp. 1-32). San Mateo: Morgan Kaufmann Publishers, Inc.