MCI Systemhouse
Applied Technology Research

UML Meets CORBA
Design of a fault-tolerant distributed system

Hernán Astudillo, Dan Uhlar, Andy Trice and Jan Pachl

Copyright Notice
Copyright © 1997 MCI Systemhouse Inc.
Executive Summary

This paper presents the development of a distributed fault-tolerant system, in a CORBA environment, using the recently proposed Unified Modeling Language (UML). The paper is useful as (1) an example of actual modeling using the UML and (2) a non-trivial example of modeling distributed fault-tolerant object distributed systems for a CORBA environment.

The example application is part of a commodity trading environment, supporting individual traders in a commodity trading environment, and focuses on two functions: the monitoring and gathering of market information, and the completion of a commodity trade. The application has significant requirements regarding distribution and fault-tolerance. The modeling building and presentation is organized into three layers: a domain analysis model, a design architecture, and a packaging and deployment model. The emphasis of the modeling exercise is on the architectural design layer, and the emphasis of the design is on the representation of services in a CORBA system.

We use seven types of UML diagrams in our example: use case diagrams, class diagrams, sequence diagrams, state transition diagrams, collaboration diagrams, component diagrams, and deployment diagrams.
# Table of Contents

1. Introduction ............................................................................................................................................. 6
   1.1. Purpose ............................................................................................................................................ 6
   1.2. Audience ....................................................................................................................................... 6
   1.3. Scope ............................................................................................................................................ 6
   1.4. Document Structure .................................................................................................................. 7

2. Approach .................................................................................................................................................. 7
   2.1. Problem overview ....................................................................................................................... 7
   2.2. Implementation restrictions ....................................................................................................... 7
   2.3. The flow of the example ............................................................................................................ 8
   2.4. UML diagrams .......................................................................................................................... 8
   2.5. Stereotypes and extensions to UML .......................................................................................... 10
   2.6. A note on UML usage ............................................................................................................... 10
   2.7. A note on writing style .............................................................................................................. 10

3. Building the Application Domain Model ......................................................................................... 10
   3.1. Problem description ..................................................................................................................... 11
   3.2. Identifying domain entities ......................................................................................................... 11
      3.2.1. Alternative representations for use cases ........................................................................... 12
   3.3. Refining the domain classes ....................................................................................................... 15
      3.3.1. Defining class states ........................................................................................................... 15
      3.3.2. Structure of the domain entities ......................................................................................... 16
   3.4. Additional requirements ............................................................................................................ 17

4. Building the Architectural Model ....................................................................................................... 18
   4.1. Representing the CORBA entities ............................................................................................... 18
      4.1.1. IDL-generated classes ......................................................................................................... 18
      4.1.2. Relating CORBA types and classes: BOA and GTIE ....................................................... 19
   4.2. Incorporating fault-tolerance with Orbix+Isis .......................................................................... 20
   4.3. Services as design abstraction ................................................................................................... 20
   4.4. Identifying the system services .................................................................................................. 21
   4.5. Representing a service: Market Information .............................................................................. 23
      4.5.1. Modeling the service participants ...................................................................................... 23
      4.5.2. Modeling the service connector ......................................................................................... 25
   4.6. Design patterns (EventStream) .................................................................................................. 26
      4.6.1. An alternative implementation of EventStream .............................................................. 27

5. Defining layers and distribution units ............................................................................................... 28
   5.1. Software layers ........................................................................................................................... 28
   5.2. Compilation dependencies and program deployment .............................................................. 30
      5.2.1. Components ........................................................................................................................ 30
      5.2.2. Deployment .......................................................................................................................... 31
      5.2.3. Replicated hardware .......................................................................................................... 32

6. Issues ..................................................................................................................................................... 33
   6.1. Usefulness of tools and techniques ............................................................................................ 33
   6.2. Appropriateness of UML to model object models .................................................................... 34
   6.3. Appropriateness of OO models to represent domains ............................................................... 34

7. Future work .......................................................................................................................................... 34

8. Conclusions .......................................................................................................................................... 35

9. Acknowledgments .............................................................................................................................. 35

10. References ......................................................................................................................................... 35

11. Appendix A:Enumeration of features in each figure ................................................................. 37
    11.1. Figure 2: Use case diagram for the problem domain ............................................................... 37
    11.2. Figure 4: Sequence diagram for Trade Commodities ........................................................... 37
Table of Figures and Tables

Figure 1: Structure of this document ............................................................... 9
Figure 2: Use case diagram for the problem domain ..................................... 12
Figure 3: Use case specification for Trade Commodities ............................ 13
Figure 4: Sequence diagram for Trade Commodities .................................. 14
Figure 5: Collaboration diagram for Trade Commodities ............................ 15
Figure 6: State transition diagram for Trade class ...................................... 16
Figure 7: Class diagram for (a portion of) the domain ................................. 17
Figure 8: A service with a replicated server .................................................. 21
Figure 9: Collaboration diagram for services to be provided ........................ 22
Figure 10: Market Information service (design for the connected entities) .... 24
Figure 11: Market Information service (design for the connector) ................. 25
Figure 12: Context of EventStream pattern (implemented via BOAImpl CORBA inheritance) and Figure 13: Context of EventStream pattern (implemented via GTIE CORBA delegation) ................................................................. 28
Figure 14: Component Diagram (top level) ................................................ 29
Figure 15: Component diagram (Application Tier) ....................................... 30
Figure 16: Component diagram (compilation dependencies) ........................ 31
Figure 17: Deployment diagram (top-level) ................................................ 32
Figure 18: Deployment diagram (node instances) ......................................... 33
Table 1: Features shown in Figure 2 ............................................................ 37
Table 2: Features shown in Figure 4 ............................................................ 38
Table 3: Features shown in Figure 5 ............................................................ 39
Table 4: Features shown in Figure 6 ............................................................ 40
Table 5: Features shown in Figure 7 ............................................................ 41
Table 6: Features shown in Figure 9 ............................................................ 42
Table 7: Features shown in Figure 10 and Figure 11 ..................................... 42
Table 8: Features shown in Figure 12 and Figure 13 .................................... 43
Table 9: Features shown in Figure 14 and Figure 15 .................................... 43
Table 10: Features shown in Figure 16 ....................................................... 44
Table 11: Features shown in Figure 17 ........................................................ 44
Table 12: Features shown in Figure 18 ........................................................ 45
1. Introduction

The Unified Modeling Language (UML) [UML 97] is a language for specifying and visualizing the artifacts of a software system. It adapts and extends the work of Grady Booch [Booch94], Jim Rumbaugh [Rumbaugh 91] and Ivar Jacobson [Jacobson 92; Jacobson 95]. 1 UML has been proposed to the Object Management Group as a standard notation for object-oriented modeling [OMG 96].

As part of the MCI Systemhouse contribution to the definition of UML, we used the proposed representation and modeling concepts to describe a fault-tolerant, distributed object application. This work was carried out concurrently with the development of UML itself, allowing us to:

- stress-test the UML as a modeling vehicle for distributed systems.
- recommend extensions to the UML for the modeling of fault-tolerant solutions.
- build UML models of selected CORBA services.

1.1. Purpose

This paper presents the UML model for a fault-tolerant, distributed object application. The construction of the model is explained in some detail, and we emphasize the architectural design, in particular the representation of services. We give special attention to using a significant portion of the UML’s concepts and graphical descriptions.

1.2. Audience

This paper is addressed to object-oriented modelers interested in:

- familiarizing themselves with the UML via an actual modeling example,
- studying the modeling of fault-tolerant, distributed object systems in CORBA environments,
- considering the representation of services in terms of UML collaborations and patterns.

We assume that the reader has a working knowledge of object-oriented analysis and design concepts; basic knowledge of distributed objects; basic knowledge of CORBA (especially object services) 2; and basic understanding of fault-tolerance issues 3.

1.3. Scope

This paper is presented as a running example, showcasing the application of the UML to a domain problem that has fault-tolerant requirements. We model several services provided to individual traders in commodity trading environments, with special attention to using a significant portion of the UML’s concepts and graphical descriptions. Also, we describe how to map our modeling concepts to a C++ CORBA implementation.

This paper is not:

- an exhaustive presentation of the UML’s modeling concepts and representation.
- an example of a specific analysis and design methodology. We to present of diagrams and connections among them independently of the methodology used to construct them 4.
- a tutorial on the UML, CORBA or fault-tolerance.

---

1 UML has been developed by Rational Software in collaboration with representatives of a number of companies, including: Digital Equipment, Hewlett-Packard, i-Logix, ICON Computing, James Odell Associates, MCI Systemhouse, Microsoft, Oracle, Texas Instruments and Unisys.

2 Helpful references are [Orfali 96] (for distribution middleware) and [Mowbray 95] (for CORBA specifics).

3 Helpful references are [Tannenbaum 96] (for distributed systems) and [Christian 91] and [Birman 95] (for fault-tolerance).

4 Although we recognize that our presentation of the model and identification of connections may suggest a methodology, as yet not clearly formulated.
1.4. Document Structure
The reminder of this paper is organized into 8 sections, plus references and one appendix.

Section 2 (Approach) presents an overview and a framework for understanding the example model and describes the relationship between the various diagrams presented.

The model is developed in the remainder of the document, organized in a series of sections, each of them highlighting a kind a diagram. Detailed explanations of the UML concepts and notation employed in each diagram can be found at the end of this paper (see Appendix A, Section 11).

Section 3 (Building the Application Domain Model) describes the functional elements of the domain model and showcases the various types of diagrams used during the initial project phase.

Section 4 (Building the Architectural Model) refines the application model to address architectural requirements such as reliability, availability, and performance. The representation of services is addressed at length.

Section 5 (Defining layers and distribution units) discusses the packaging of software artifacts into distribution and deployment units. A graphical representation of compilation dependencies is also presented and discussed.

Section 6 (Issues) presents a several issues and problems raised during the development of the model, which reflect on the overall usefulness of UML for modeling.

Section 7 (Future work) suggests some ways in which the authors feel this work could be extended.

Section 8 (Conclusions) summarizes the paper.

Appendix A: Enumeration of features in each figure presents (in tabular form) the UML features used in each figure throughout the paper.

2. Approach
This section describes briefly the application, some restrictions, and the flow of our presentation.

2.1. Problem overview
The example application supports services to individual traders in a commodity trading environment. The application has significant requirements regarding distribution and fault-tolerance (other issues, such as security and load-balancing, where not considered due to scope limitations). Our modeling will focus on two functions: the distribution of market information, and the completion of a commodity trade.

2.2. Implementation restrictions
The development of an application to address the above mentioned domain has some empirical restrictions, which influenced heavily our modeling:
1. CORBA: the distribution aspects of the application must be handled through a CORBA system [OMG 96].
2. Orbix+Isis: the fault-tolerance aspects of the solution must be addressed, whenever possible, with the Orbix+Isis [Isis 96] toolkit (features offered by this toolkit are described in Section 4.2).
3. C++: the implementation language must be C++ whenever possible. Some implications of this choice for CORBA-oriented programming are mentioned in Section 4.1.1.
2.3. **The flow of the example**

As we present the various excerpts of the model, we will point out the specific connections between the different diagrams as described in this section. Notice, however, that the set of diagrams presented are not intended to provide complete traceability across layers, as would be ideal.

For each diagram, we describe its purpose, present its content and derivation, and discuss the connection with other diagrams. Additionally, we enumerate (in Appendix A) the individual modeling features that each diagram illustrates.

Throughout this example, we will make a distinction between the domain and the application: the domain is the “real world” being modeled and the application is the software system that will support (and perhaps replace) the domain. Some domain entities may not have a counterparts in the application, and most application entities will be created with no direct counterpart in the domain.

2.4. **UML diagrams**

We divide our modeling task, and the flow of this example, into three activities: “Application Domain Analysis,” “Architectural Design” and “Physical Packaging and Deployment.” These activities are not stages in the software development life-cycle, but different abstraction levels that highlight particular viewpoints of the application. In practice, modelers will iterate between levels, as we did in fact.

UML elements are displayed graphically via diagrams. We will use seven types of diagrams in our exposition: use case diagrams, class diagrams, sequence diagrams, state transition diagrams, collaboration diagrams, component diagrams, and deployment diagrams. Other UML diagram types, namely activity diagrams, object dataflow diagrams, and instance diagrams, are not represented in this example. The main structuring unit in the UML is the package, which provides a means to organize the model elements in groups according to the modeler’s needs. Packages may be nested. For example, a subsystem (with its component classes) and a class (with any internal state diagrams) are good candidates for packages.

Each activity privileges some kinds of diagrams, as shown in Figure 1. This figure also provides a framework for understanding the diagrams in the example model, as well as highlighting the relationships between the various diagrams that we will use.

---

5 For other layering alternatives, see Rational Rose views (Use Case, Logical, Component and Packaging) or Kruchten’s “4+1” views [Kruchten 91].
The “Application Domain Analysis” package shows the diagrams that we used for the application domain analysis. Use case diagrams present a graphical view of the actors and how they interact with the system. Sequence diagrams specify the sequence of messages involved in individual use cases; collaboration diagrams resemble sequence diagrams, but focus more on object relationships and allow for the presentation of dataflows. Class diagrams provide the classes and types needed to realize use cases and state transition diagrams describe the states of particular classes. Finally, textual description of use cases
are of vital importance, though they are not proper diagrams; they can be attached to a sequence diagram or left as a separate document (as we do in this example).

The “Architectural Design” package shows the diagrams used for the architectural design. In general, all diagram kinds used during analysis can be used for the design; however, our example includes only collaboration and class diagrams. Diagrams related to dynamic behavior were left out in order to allow a longer discussion of CORBA, services and fault-tolerance issues.

The “Physical Packaging and Deployment” package shows the component diagrams and deployment diagrams that we used to describe and allocate the development units of the system.

2.5. Stereotypes and extensions to UML

UML can be extended via stereotypes, which can be defined for a specific project or corporate setting. Of course, the flexibility of this approach must be tempered with the awareness that its overuse may lead to a proliferation of mutually incompatible UML dialects. We decided that the specialized nature of our project’s domain (distributed CORBA systems) was well served if we defined a few additional stereotypes, and we have used «type», «connector», «IDL-generated» and «delegation» throughout this design.

2.6. A note on UML usage

This example was developed concurrently with, and as part of, an effort to enhance the UML. Utmost care has been taken to track the notation and naming changes, but some inconsistencies may remain between the usage in this example and the latest version of the UML.

The UML is evolving and this white paper presents only highlights of its use for actual modeling, omitting many details. For the latest information, see the object technology section of Rational Software Corporation’s Web site (http://www.rational.com).

2.7. A note on writing style

To simplify our presentation for didactic purposes, many modeling decisions (e.g., how to best represent a service, or what kind of replication we used for a server) are not documented in the main text. When deemed useful, we have added footnotes pointing to our reasons for some choices or to provide references for those interested in further treatment of the issue(s).

We found that formatting names in ad-hoc forms facilitates following the discussion. Hence, names and labels in the text of this paper are formatted as follows:

- Important concepts are written in italics.
- New words or concepts are introduced in italics.
- Names of “packages” are written in bold letters, surrounded by quotes.
- Names of use cases are written in bold letters.
- Names of services are written in bold letters.
- Names of classes and types are written in italics.
- Kinds of diagrams are written in italics.

3. Building the Application Domain Model

The application domain model introduces the entities and requirements in the problem domain that are relevant to the construction of this application. This model is essential to ensure understanding between

---

6 At the time of this writing, inconsistencies remain between the “semantics” and “notation” working documents; whenever possible, inconsistencies between these core documents were resolved by direct interaction with their authors.
customers and developers, and hence emphasizes the behavior of the domain entities (as perceived by the
users) rather than structural issues (which will remain to be solved during the design).

The application domain analysis is primarily concerned with the specifications required to satisfy the
application’s functional requirements. For instance, for a commodity trading system, traders must be able
to read news and execute trades. The application domain analysis for the system identifies usage scenarios in
use case diagrams, defines classes such as news, trade, and contract in class diagrams, and specifies
interactions between instances of these classes using sequence diagrams and interaction diagrams.

The application domain analysis also yields a description of non-functional requirements for the application;
in our case, some fault-tolerance properties (in Section 3.4).

3.1. Problem description

The main users of the system are the traders, who perform individual trades from their desks, mostly over
the phone. Traders are supported by personal workstations, connected to larger corporate servers. Traders
rely on many sources of market information, some informal and some formal. The workstation is used to
convey some of this market information, which is provided by the system of interest in two forms: pricing
information and market news, from both wire services and financial information providers. Pricing
information includes current prices of specific commodities (e.g., oil) for different time periods. Market news
includes press releases, demand and supply forecasts, wire stories, etc. The system must collect, format,
and disseminate this news to traders.

A trader executes a trade after completing a negotiation with another trader (called a counterparty in trading
parlance). To complete the trade, a formal contract must be generated specifying the terms of the deal (e.g.,
volume, price, quantity, delivery date, mode of transportation). In addition, the trader’s position in the
commodity being traded must be updated appropriately. The position is captured in a schedule often
known in the industry as a slate, which gives a summary of all agreed to receipts and deliveries for a given
combination of commodity, movement location and movement period, as defined by all contracts pertaining
to that combination. Receipts and deliveries of a trade are called its legs.8

3.2. Identifying domain entities

Our first step is to describe several business functions, some of which will be supported by the application.9
Each business function is represented by a use case9 or usage scenario. Use case diagrams are the main
communication tool between users and analysts, but also convey information to object modelers (who need
to implement the analysis result) and test-script writers. Figure 2 provides an overall picture of the use
cases executed by the trading system, using the classic notation introduced by Jacobson [Jacobson 95].
Use cases are denoted by ovals, and actors external to the system are denoted by “stick men”.

---

7 [Fowler 96] uses the term contract to describe a trade. In the industry, trade, contract and deal are often used synonymously. We preserve the use of “contract” to identify the formal, printed document, and use “trade” to capture the concept of the agreement.
8 [Fowler 96] uses the terms long and short to represent a buy and a sell, respectively. Our domain experts were more comfortable with the terms “receipt” and “delivery.” This is principally due to the company’s downstream requirement to schedule the movement (receipt and delivery) of the actual commodity. In addition, our domain experts felt it was important to make a distinction between the trading of paper and the trading/handling of the physical commodity. Thus, Fowler’s use of the term instrument was replaced by “leg.”
9 But they don’t have to be; a model of non-automated functions can be used for Business Process Reengineering [Jacobson 95].
10 Some methods prefer to describe a function with several alternative-action use cases, but our domain experts preferred to build the alternatives inside a single use case per function.
3.2.1. Alternative representations for use cases

Individual use cases (ovals) are specified in more detail using a combination of text, sequence diagrams, and/or collaboration diagrams. The rest of this section presents all three descriptive techniques for the Trade Commodities use case,
USE CASE: trade commodities

The trade commodities use case describes how trades are concluded, introduced and processed within the trading company.

BUSINESS MOTIVATION:

Trading commodities is the central business function of the company. The process of trading a commodity is central to the system.

When negotiating a trade, the company trader must take many things into account. The dynamics of the market in which the commodity is traded is of utmost importance. The company position and internal strategy relative to the commodity will also determine whether an offer to trade is accepted or rejected. Internal and external news surrounding the commodity can also determine the way in which a trader approaches negotiations. The credit history of the counterparty and credit at risk are also taken into account. And, of course, current commodity price information is essential to the terms of the deal.

Upon agreement to trade, details of the agreement must be captured in a secure manner, the company’s position in the commodity must be updated and news of the trade must be communicated to other traders as well as to middle and back personnel. A formal, printed contract must also be generated and transmitted to the counterparty.

ACTORS:

Company trader (trader)
Counterparty trader (traderContact)

PRECONDITIONS:

Credit agreement(s) must be in place between the Company and the Counterparty
Contract template(s) must be in place between the Company and the Counterparty
Counterparty account information must be complete
Counterparty trader must be authorized to trade
Company trader must be authorized to trade
Market data provider(s) agreement(s) and service must be in place

PROCESS DESCRIPTION:

When a counterparty trader wishes to trade a commodity a telephone call is made to a company trader and the request to trade the commodity is announced. The company trader must analyze the offer to trade. For a detailed discussion of this phase of the process see the “analyze potential trade” use case. The terms and conditions of the deal are then discussed and negotiated with the company trader.

On acceptance of the deal, the details of the trade are registered with the Trade Administrator and the company trader focuses attention on the next trade. The Trade Administrator makes a permanent, persistent record of the trade details, notifies the Position Administrator and the Market Data Service (this is a distribution mechanism internal to the company and should not be confused with an external market data provider, such as Reuters) of the trade, requests a formal printed contract from the Contract Administrator and then turns attention to the next trade.

If no previous trades for the same commodity, movement location and movement period have been made, the Position Administrator creates a new slate. The slate is an internal mechanism used to group like components of trades together for the purpose of computing position for a commodity, at a location, for a particular movement period (usually a calendar month). If the trade falls within an existing slate, that slate is simply updated with the details of the new trade. In both cases a permanent, persistent record of the slate state is made.

For a detailed discussion of the notification activities of the Market Data Service see the “distribute trade news” use case. And for a detailed discussion of the execution activities of the Contract Administrator see the “generate contract” use case.

POST CONDITIONS

Details of the trade are recorded for use by various departments (e.g., credit, scheduling) within the company
The company’s position with respect to the commodity is updated
Interested traders and other company personnel are notified that a trade has been completed
A printed contract is distributed to the Counterparty

RELATED USE CASES:

analyze potential trade (uses)
distribute trade news (uses)
generate contract (uses)

Figure 3: Use case specification for Trade Commodities

Figure 3 presents the textual format of a use case specification. The UML does not prescribe a specific format, but does allow text to be attached to any and all diagrams. Due to the complexity of this use case and for reasons of presentation, we have chosen not to attach this specification to the sequence or
collaboration diagrams that follow. It is included, in particular, for the benefit of those who wish a deeper understanding of use case being modeled in this section.

At this point, it is important to notice the occurrence of administrators in the use case. Administrators are abstractions that conceal certain well-defined functionality in the domain (e.g. the TradeAdministrator keeps track of concluded trades and the ContractAdministrator generates actual formal contracts).

The trade commodities use case can also be described graphically, as is illustrated with the sequence and collaboration diagrams that follow (Figure 4 and Figure 5, respectively). As can be seen, each diagram type allows the modeler to present different aspects of the use case.

![Sequence diagram for Trade Commodities](image)

Figure 4: Sequence diagram for Trade Commodities

Figure 4 is a sequence diagram for the TradeCommodities use case supported by the trading system. This diagram is a time ordered representation of the object interactions that occur when completing a commodity trade. It contains no information on business rationale, but allows an easier grasping of the message exchange among domain entities that are required to perform this function.

Notice also that the analyst has incorporated some suggestions for the designer (in the form of notes); these ideas may or may not be used later.
We can also specify the Trade Commodities use case with a collaboration diagram, as shown in Figure 5. This diagram presents much of the same information as the preceding sequence diagram along with dataflows (message parameters). It also shows the possibility of multiple instances of some objects.

### 3.3. Refining the domain classes

The use cases in previous sections showed several partial views of the domain, each of them built around specific business functions. Each use case introduced some domain entities, performing certain activities in the domain. In this section we show how to describe in UML the behavior of each class in detail and how to combine all classes into a single structural model.

#### 3.3.1. Defining class states

The details of a class’ behavior can be shown with a state transition diagram. A state transition diagram is a directed graph, showing the state space of a given class, the events that cause a transition from one state to another, and the actions that result from a state change. Each state transition diagram is associated with one class or with a higher-level state diagram.
A state transition diagram allows analysts to communicate to users and developers their understanding of the detailed behavior of a class. Not all classes need to be described in detail using state transition diagrams. Figure 6 illustrates the state transitions of the Trade class from the domain class model.

UML allows for the presentation of concurrent substates (separated by a dashed line). In our model, the receipt legs can be actualized separate from the delivery legs and the trade itself is not actualized until all receipt and delivery legs are actualized.

3.3.2. Structure of the domain entities

The previous use cases showed several partial views of the domain, each of them built around specific business functions. Eventually, these partial views must converge to a single global view of the domain. We have represented this global view in Figure 7 and Figure 9, which show the static (or structural) and dynamic (or behavioral) views, respectively.
The proliferation and multiple usage of these entities leads, eventually, to the convergence of these partial views into a single global view of the domain. We have represented this global view in Figure 7 and Figure 9, which show the static and dynamic views, respectively.

![Class diagram for (a portion of) the domain](image)

**Figure 7: Class diagram for (a portion of) the domain**

We have used a class diagram (Figure 7) to describe the entities in a portion of the problem domain. The global view shown in Figure 7 is a static or structural view of the domain model for the trading application (information on Slates and Counterparties is omitted for space reasons).

### 3.4. Additional requirements

The domain analysis is not complete if only the domain entities are found. Additional requirements upon them, and upon the system as a whole, need to be stated too. We performed this task concurrently with the above modeling, but present it here as a separate product.

The application must exhibit the following properties:

1. **High availability**: Several domain entities are essential, because the trader cannot work at all without them: *pricing* (reflecting market information) and *position* (reflecting the trading company’s current state in a commodity). These elements must be highly available, a requirement that strongly suggests to use replication and distribution techniques to minimize the probability of service breakdown.

2. **Reliability**: The communication channels between traders and critical components, and among these, must be reliable, regarding availability and correctness. This requirement tails in perfectly with our use
of Orbix+Isis, which offers reliable communication channels among CORBA objects (normal CORBA implementations are not reliable in this sense).

3. **Minimal interference**: The system must avoid interference with the trader’s activities, including delays and demanding replies. Some possible solutions are having the trader issue only asynchronous messages, and feed data to the workstation in stream mode (rather than upon request).

4. **Coverage**: News providers also report prices; all providers share one source for the prices (namely, the exchange), but have complementary areas of strength for news. A trader must draw from several news services to obtain a complete picture of the market. This suggests some degree of replication.

5. **Real time**: The trading company’s position is reflected in the slate; because an accurate and timely view of the position is vital information to the trader, the slate must reflect dynamically the trader’s position, and propagate it so.

6. **Integrity**: The system must keep **safe** and **reliable** records of the completed trades. This is the usual concept of integrity (e.g., in databases), involving consistency maintenance, access control and loss recovery. This suggests that a DBMS will be necessary for some of the entities.

### 4. Building the Architectural Model

The architectural design is primarily concerned with refining the application domain model to incorporate support for the required functions and properties. For instance, if the distribution of news and prices to traders requires a high degree of availability, a complete architectural design must define classes realizing a reliable communication channel and a replicated distribution service. The structural relationships among classes are described in a class diagram, and the interaction between instances of these classes in specific scenarios (e.g., failure) can be described in a sequence or collaboration diagram.

The architectural model identifies specific design choices taken to satisfy the requirements. We will proceed by first describing CORBA and fault-tolerance modeling issues (prerequisites to the design), then identifying services among the domain entities, and finally building a design model for one specific service. Our design must satisfy the service requirements (e.g., having reliable links) and conform to the implementation restrictions (e.g., using Orbix+Isis).

Noteworthy features of this section are: we introduce some UML stereotypes to denote services and some CORBA peculiarities, and a UML representation for design patterns.

#### 4.1. Representing the CORBA entities

An important restriction for our design is that distribution aspects must be supported with a CORBA implementation. We will represent some CORBA concepts in UML, to be able to use them later in our models.

##### 4.1.1. IDL-generated classes

CORBA [OMG 96] distinguishes between “interfaces” and “implementation”, in a similar manner to UML’s types and classes.\(^{11}\) Interfaces describe the operations invocable in an object, without reference to their implementation. The development of a CORBA system usually starts from some “IDL specifications” (interface descriptions written in IDL language; IDL specifications are types in the UML sense). IDL specifications are suited to automatic generation of most distribution-related code (such as initialization of proxies, remote dispatching of operations, or marshaling of parameters). For example (see Figure 10), the type `Employee` can be realized by the class `Employee_i` (the “`_i`” tag is a CORBA convention to indicate “implementation of”). Notice that this corresponds to the UML notion of a class refining a type.

---

\(^{11}\) The notion of conformance among CORBA interfaces and implementations is more elaborate than UML’s notion (e.g., it includes conformance of individual objects to interfaces), but the UML distinction is enough to model all cases where objects are conceived as class instances.
C++ implementations of CORBA\textsuperscript{12} (such as Orbix+Isis) usually process the IDL specifications to generate “macros”, which allow standardized definition of some idiomatic CORBA-isms (e.g., generation of classes as client stubs). These macros can be used to define actual classes, but they do not correspond to any modeling element (they are just text-replacing devices). They also hide the fact that several intermediate and helper classes are built; most of these classes do not belong to software components of their own (such as a separate source file), but the modeler needs to be aware of their existence, since they are part of the global object model. To facilitate our modeling, we have abstracted some widely used macros into \textit{template classes} instead: what would have been a macro instantiation becomes a template instance. We introduce the stereotype «IDL-generated» for these templates.

For an example, consider Figure 10: \textit{GTIE\_Employee} is actually a macro generated by an IDL processor, which abstracts the CORBA “TIE” inheritance mechanism (see next section). We label this template as «IDL-generated» to indicate its special status.

Notice that the stereotype «IDL-generated» denotes classes that are generated directly by IDL processors; these classes are not coded by the programmer (or by a UML-supporting CASE tool). We maintain these classes in the design diagrams to inform the modeler and to avoid spurious errors in model-level validity checking (e.g., due to missing methods that are actually defined by the macros).

\subsection*{4.1.2. Relating CORBA types and classes: BOA and GTIE}

The separation of interfaces and implementations in CORBA allows a great deal of flexibility matching required behavior with available code. The preferred approach to build CORBA code is to generate source code “skeletons” from an IDL specification (providing the overall elements of the needed code), and then fill these skeletons with detailed programmer-written code. There are two main approaches to fill these skeletons: via \textit{inheritance} or via \textit{delegation}.

The inheritance approach is reified by the CORBA “BOA” mechanism, in which the developer sub-classes the original interface and redefines its operations, adding detailed code. It goes like this: an IDL processor generates: (1) an abstract class, representing the target IDL interface, and (2) a CORBA-aware class, sub-class of the previous abstract class; this latter class adds ORB-communication features such as server location and argument marshaling. The developer builds an implementation for the target IDL specification by sub-classing the CORBA-aware class, and by redefining all of its operations (to add detailed code).

The delegation approach is reified by the CORBA “TIE” mechanism (“GTIE” in Orbix+Isis), in which the developer builds a separate class, with similar operations to the original interface, and creates a sub-class of the interface whose non-CORBA operations are made to “delegate” (i.e., forward) their invocation to the same-name operation in the developer-written class. An object may \textit{delegate} to another one the performing of certain operations, either to simplify the implementation or to separate abstraction levels. Because C++ is a statically type-checked language, the generated CORBA-aware class must be a template class, which takes as parameter the “delegated-to” class; otherwise the type of this class cannot be known at compile-time.

Notice that, by virtue of inheritance, the CORBA-aware class is a subtype of the target specification, and the developer must ensure that the detailed classes actually written are also subtypes of the target specification. This is represented in Figure 12: the \textit{Sender} and the \textit{Receiver} of an \textit{EventStream} must be subtypes of the \textit{Events} that specify it; this ensures that they have compatible interfaces.

\textsuperscript{12} Unfortunately, CORBA vendors usually offer C++ implementations, though C++ is not well suited to express several modeling concepts used in CORBA. For example, the following concepts require somewhat contrived, indirect ways to be expressed in C++: objects that do not belong to classes; subtypes that are not subclases; delegation a from “shell” object to the “actual” object; template classes where the members in the interface are defined only at instantiation time.
Delegation is not explicitly recognized in UML, but we consider it a modeling feature distinct from mere referencing. We introduce the stereotyped dependency «delegate» for this purpose. In Figure 10, class GTIE_Employee<Employee_i> (instantiated from an IDL-generated template) delegates its operations to class Employee_i (written by the developer).

Figure 12 and Figure 13 show two implementations of the same set of classes, one of them using “BOA” and the other one using “GTIE” mechanism.

4.2. Incorporating fault-tolerance with Orbix+Isis

A second restriction on our design was that fault-tolerance aspects had to be handled by the Orbix+Isis toolkit. Orbix+Isis is a CORBA-compliant toolkit, which supplements the normal CORBA class suite with some classes enabling some fault-tolerant implementations.

Orbix+Isis offers two main “computational styles”, which provide with fault-tolerant semantics:

1. **Active Replica**: An *ActiveReplica* is an object that has “replicas”, or similar objects. The replicas belong to a “process group”, a set of entities that is an entity by itself and behaves as one. The group members collaborate to maintain the illusion of the group as a single object. *ActiveReplica* is extremely powerful when combined with distribution, since the group may continue functioning even after some network partitioning. There are three “communication styles” in *ActiveReplica*:
   - Multicast: Each request is sent to all the group members, and the replies are “conglomerated” (combined) into a single reply (the conglomeration policy can be modified by the user).
   - Client’s choice: The request sender (i.e., the client) determines a policy to choose which reply from the group members should be sent; this allows some forms of load-balancing (the default chooser is round-robin).
   - Coordinator/cohort: A distinguished group member is nominated as coordinator, and issues replies in behalf of the group; the other group members wait until the coordinator fails. This policy is also called “fail-over” by others.

2. **Event Stream**: An *EventStream* is a persistent message queue with logged, time-ordered, guaranteed-delivery messaging semantics. Several objects may be “sending” and “receiving” events; these events are defined at compile-time, i.e. they must be known in advance. Receivers may register their interest in specific events. *EventStream* is extremely useful to provide location-transparent message delivery.

The abstractions offered by Orbix+Isis take into account failure of individual processes, maintenance of group consistency, and reliable communication among objects, among other fault-tolerant behaviors.

4.3. Services as design abstraction

A key task in the architectural design is determining the elements that need to be added to the original domain model. The new elements should be chosen to support the required properties, so they must be related to specific model elements. The conceptual vehicle for identification and attachment of such properties is the *service*, for our purposes defined as a relationship involving a *server* with a request-receiving role, a *client* formulating requests, and an *interaction channel* carrying requests and replies. In general, the valid requests are defined a priori. Also, note that although each element is logically one, it may denote a set of collaborating entities.
Consider Figure 8: at a conceptual level, the depicted service relates a client and a server, via a connector. The top-level entities of the diagram can be decomposed, where the request-carrying channel and the server have been decomposed to indicate some peculiarities (namely, the server is actually a set of collaborating entities, and the channel is a non-trivial service itself). The conceptual top level allows to present services in an abstract manner, and to identify services among interactions in conventional models. Our modeling will attach fault-tolerance “properties” to interaction identified as services (e.g., activity logging and at-most-once delivery semantics to a channel; and high availability and secure access to a server).

UML has no explicit notion of a service. We will show later how to represent a service (namely, MarketInformation) using a design pattern (Section 4.4). However, identification of services among domain interactions is greatly facilitated if the analysts is provided with a means to indicate that a given connection is rather persistent and conceptually interesting as an entity. We introduce the stereotype «connector» for these cases. (We use «connector» instead of «service» because they provide support for the service, but are not the service itself.)

4.4. Identifying the system services

The design activity starts with the entities identified in the problem domain, plus the additional requirements established a priori (as described in Section 3.4). We can combine both entities and requirements in a diagram, as follows:
Figure 9: Collaboration diagram for services to be provided

Figure 9 presents a graphical overview for the architecture of the application and its services. The application consists of actual objects collaborating via messaging to perform a global function; we use a collaboration diagram, though the sequence numbering is irrelevant (we could have used a class diagram instead, but with loss of the messaging representation).

This stylized use of a collaboration diagram allows application analysts and software architects to describe activities at the boundary of application domain analysis and system architecture (e.g., identifying some relationships among domain entities as services).

By setting the application’s objects as an interaction, we reinforce visually the “links” among them. We will use these links (labeled «connector») to identify services. Most entities in Figure 9 were already introduced in Figure 5; notice that many of them are called “administrator” (e.g., TradeAdministrator or PositionAdministrator); administrators are entities identified during the analysis as controlling some specific resource (e.g., the life-cycle of a contract or the global position, shared by all traders).

---

13 Since UML is process-independent, no specific support for our diagrams exist, though it can be developed via stereotyping (in the context of a project) or extensions to the meta-model (by tool vendors).
In Figure 9, several services can be identified, roughly one per connection. Human traders use the TradeRecording service (provided by TradeAdministrator) to record ongoing and performed trades, the PositionQuerying service (provided by PositionAdministrator) to determine their current position, and the MarketInformation service to obtain price and news information (provided by MarketDataService). Notice though that Hearsay is not a bona fide service, but only an interaction. Also, from a conceptual viewpoint, a particularly interesting example of service is the MarketInformation, because it is not client-driven (i.e., via requests, in a reply-request mode) but server-driven (i.e., by continuous publishing, in a stream mode).

Note that in Figure 9, the original domain model has been complemented with some purely architectural properties, derived from the non-functional domain needs (as stated in Section 3.4). For example, the communication between a Trader and a TradeAdministrator needs to be “fault-tolerant” (for high availability), “logged” (to enable auditing) and “asynchronous” (to optimize trader time). The properties have been annotated both at the server and at the interaction.

### 4.5. Representing a service: Market Information

We will devote this section to describe the modeling of the Market Information service (shown in Figure 9). This service involves the Trader and MarketDataService entities: the MarketDataService distributes prices and news information to the Traders, not on a per-request basis but as a stream.

UML makes a difference between types and classes. Types correspond to the description of an interface (a set of operations, attributes and constraints) while classes correspond to the description of a possible realization (implementation) of such interface.

The relationship between a type and its realizations is called a refinement dependency: a given class refines the definition of a type, by adding implementation details to it. Incidentally, refinement relationships are also possible among types, among classes, or from a collaboration to either (see below).

During our design activity, the domain-level entities (e.g., MarketDataService and Employee) will be considered as types to be implemented via classes. This allows us to separate the domain information (contained in the types) from the design decisions (reflected in the classes that implement them). The diagrams explaining the design (Figure 10 and Figure 11) show domain entities with the stereotype «type».

Sections 4.5.1 and 4.5.2 present a design to implement the Market Information service using C++ and the Orbix+Isis development and execution environment. We will cover these models in more detail now.

#### 4.5.1. Modeling the service participants

Figure 10 shows the design for the domain entities directly involved in the service (namely Employee and MarketDataService). Employee and MarketDataService are represented as types. These two classes were already identified in the domain model (hence they are the same entity in a unified analysis+design model). All other classes are purely architectural, created to provide the properties attached to these entities in Figure 9.

The MarketDataService is conceptually a single entity, but in practice it denotes several objects, because several incoming price+news streams exist. However, the incoming streams cannot be directly forwarded to the Traders, because: (a) prices are the same, (b) news may be redundant among providers, and (c) traders have specific areas of interest. Hence, our design must include: (1) replicated objects that are conceptually one, and (2) coordination among these replicated objects to ensure a single sender for each news/price item. Our choice is the “Active Replica” computation style, using “Cohort-Coordinator” communication style (explained in Section 4.2). The MarketDataService type is realized by the class MarketDataService_i, which is related to the type by the “BOA” CORBA mechanism (explained in Section 4.1.2).

---

14 In general this is not the case, though some methods treat every single object interaction as an instance of a service.

15 Clients could be annotated too, for example if the service requires specific properties from its clients.
The Trader entity is also a multiple object, because several traders exist\textsuperscript{16}. However, they do not explicitly coordinate their activities, except as already provided by the overall design itself (e.g., by sharing a common slate). Hence, our design must include replicated objects with no extra coordination. Our choice is the “Simple Active Replica” computation style (explained in Section 4.2). The Trader type is realized by the class Employee\textsubscript{i}, which is related to the type by the “GTIE” CORBA mechanism (explained in Section 4.1.2).

\textbf{Figure 10: Market Information service (design for the connected entities)}

Let us look at the implementation of the classes. The domain-level class MarketDataService is implemented via the C++ class MarketDataService\textsubscript{i}, which is defined to account for: (1) its own properties of active replication with primary sender style, and (2) the connector’s property of broadcast mode with publisher-subscriber semantics. Hence, MarketDataService\textsubscript{i} is implemented as an Orbix+Isis Active Replica (see

\textsuperscript{16}More importantly, several traders may interact with the system.
Section 4.2), and relates to the MarketDataService type by the “BOA” CORBA mechanism (see Section 4.1.2).

### 4.5.2. Modeling the service connector

Figure 11 shows the design for the connector supporting the service. The market information connector is the one already identified in the domain model, but it is not directly implemented by a single entity. Instead, we have the enumerations NewsEvents and PriceEvents. These enumerations are implemented (in UML parlance, “refined”) using the EventStream pattern, which we explain separately (see Section 4.3). This pattern provides the fault-tolerance properties attached to the MarketInformation service in Figure 9.

![Diagram of Market Information service (design for the connector)](image)

The fault-tolerance requirements attached to the service (i.e., “broadcast”, “publisher-subscriber” and “at-most-once”) suggest that we should use the EventStream provided by Orbix+Isis (see Section 4.2).

However, we notice now that a stream of price and news events is a heterogeneous stream; designer experience suggests to represent the domain-level service as two separate services: PriceEvents and NewsEvents at design level.

We describe each stream in terms of the (known in advance) events that may flow through it. Since this is a fixed set, it is possible to use a UML “enumeration” (which is a class with the «enumeration» stereotype).
For example, in Figure 11, the *PriceEvents* enumeration indicates the (only allowed) kinds of price-related events that a *MarketDataService* may generate and that a *Employee* may receive.

The refinement of a stream implies the refinement of its enumeration. Since we had decided on an *EventStream*, naturally we want to represent each stream enumeration as being refined by a stream. How is this stream represented? In Section 4.6 we show an implementation of the *EventStream*, as a “design pattern”. At this point we only mention that, in Figure 11, the *PriceEvents* enumeration is refined by *EventStream*, depicted by an oval; the “Sender”, “Receiver” and “Events” dependencies indicate the role that each class plays in the pattern.

An important implementation constraint has influenced this design (see note in the diagram): the Orbix+Isis implementation available to us does not allow a single object to be a receiver of two different streams at a time. Our workaround is to leave *Employee*_i as receiver of *NewsEvents*, and to introduce a new class, *PriceEventsMonitor*, to act as receiver of *PriceEvents*.

### 4.6. Design patterns (EventStream)

Figure 12 and Figure 13 are class diagram representing a pattern, which describes the structure and (optionally) behavior of a set of object implementing a single higher-level concept. Patterns have been widely studied recently [Gamma 95], and their use simplifies greatly the description of how to implement a given design. Patterns are represented in UML using *template collaborations*.\(^\text{17}\)

A design pattern (or pattern for short) specifies a mechanism, consisting of structural elements and behavioral elements. A specific pattern represents a set of *collaborating types and/or classes*, assembled as such because it is a conceptually interesting group.

UML patterns descriptions consist of two aspects: a context (structural composition) and an interaction (dynamic behavior). The pattern context is represented in a class diagram, and includes types and classes for the participating entities, and relationships among them. The pattern interaction is represented with sequence diagrams and/or collaboration diagrams (not shown for this pattern), and includes objects and messages among them.

\(^{17}\) Since use cases are represented using collaborations too (but not templates thereof), UML provides a single unifying abstraction for both concepts.
Figure 12: Context of EventStream pattern (implemented via BOAImpl CORBA inheritance)

Figure 12 shows the context of pattern EventStream, one of the abstractions offered by Orbix+Isis. Since UML does not provide yet a standard iconic representation of pattern contexts, we have used the stereotype «collaboration» on a class template, and group the pattern elements graphically “inside” it. EventStream requires three arguments (all types) to be instantiated: a Sender, a Receiver, and an Events; the collaboration describes how Senders and Receivers are related, and how the Events are sent between them. The Receiver is built by the “BOA” mechanism.

When a pattern is instantiated, we indicate (as we did in Figure 11) which classes will take which roles in the pattern. Notice the restriction that the class used as Sender and Receiver must be sub-types of the class used as Events, and this one must be an enumeration.

The dynamic behavior of EventStream can be described with interaction and sequence diagrams, which are the same tools used to describe use cases. We have omitted these diagrams from our example.

4.6.1. An alternative implementation of EventStream

Figure 13 shows a second possible implementation of the EventStream pattern, this time using a “GTIE” mapping. The pattern itself presents the same interface (i.e., participating types and interaction semantics), but has a completely different internal implementation.
5. Defining layers and distribution units

The architectural model must be completed with implementation details before it becomes available to direct actual implementation. We present the final shape of the software system as a set of layers, and show external systems (such as the used CORBA implementation) as layers too.

The description of these details, as defined in UML, consists of component diagrams (identifying deployable components among the application components) and of deployment diagrams (allocating components to processors).

5.1. Software layers

The development process for an application produces diverse software artifacts, foremost among them programs and coding units. The implementation of any large system can be facilitated by organizing these units into “layers”, which provide specific functionality through specialized interfaces. Layers can use facilities provided by other layers.

UML has no explicit concept of layer, but it provides packages, which can be organized in a layered manner, if so desired. We present in Figure 14 the layers for our application; notice that we display them graphically layered too, for easier understanding.
Component diagrams describe (to developers and system administrators) the structure and dependencies of the source and executable files of the application. Figure 14 shows the top-level physical packaging of the commodity trading application into packages and modules. Notice the several layers (explained in the next paragraph) and existence of an active object (“OrbixD”, drawn with a thicker line); this object will have its own thread of control (a restriction given by its vendor, in fact). All source and executable code needed to implement the system design reside within one of the packages in Figure 14.

The “Application Tier” encapsulates the functions that we are building, while the “Utilities and External Interfaces” and “Persistence Services” layers provide the lower-level functionality indicated by their names. The “Desktop Tier” contains interface and workstation software, which we have not considered explicitly in our modeling. The desktop software provides the trader with an interface to the automated business functions that we developed.

Packages can be nested: in Figure 15 we refine the “Application Tier” package (introduced in Figure 14), to show the sub-packages inside it, namely “TradeAdministratorServer,” “MarketDataServer,” “PositionServer” and “ContractServer”. This figure also shows the dependencies (both internal and external to the “Application Tier”) among sub-packages, and between sub-packages and other layers.\(^\text{18}\)

\(^{18}\) Griss introduces the term “façade” to describe packages that are actual layer interfaces, and not mere grouping of entities [Jacobson 97].
5.2. Compilation dependencies and program deployment

Once the application components have been defined and organized, the remaining development task that the UML supports is indicating the *compilation dependencies* and the final location of such components. Some dependencies (generally structural ones) can be deduced from relationships among design entities (e.g., from inheritance among classes) but others require specific modeler intervention. For example, it is not obvious a priori whether an object and its stream-receiving components must reside in separate processes, or in a same process with separate threads (and hence shared run-time data).

5.2.1. Components

Figure 16 shows the main software modules needed to build the application as a C++ system using the Orbix+Isis development and execution environment. (Due to its size, this diagram shows only a fragment of the system described in the other diagrams). The diagram presents the software entities used to create two executables: **PositionServer** and **TradeAdministratorServer**. The compilation and linking process will produce these executable files.
Notice that this type of component diagram interests not only software developers, but also architects of the software development environment (not of the application), since they need to understand which entities and tools the environment must support.

5.2.2. Deployment

The executable components that are the final product of the process shown in Figure 16 must be now mapped to specific server processes, which will be the run-time agents actually receiving and processing requests (in a CORBA environment, they will also be registered with an ORB to allow access by client
objects). The deployment (allocation of processes to processors) is shown in the deployment diagram in Figure 17.

![Deployment Diagram](attachment:image)

**Figure 17: Deployment diagram (top-level)**

Software architects use these diagrams to indicate deployment strategies and primary communication paths in the system, and network architects take them as an input to network design and architecture. For example, a “broadcast” path could be implemented via Ethernet and UDP/IP (where messages are sent and received through a shared medium) and a “point-to-point” path via Ethernet and TCP/IP (where messages could be addressed to specific receivers).

Notice that the structure of the nodes and connections in Figure 17 resembles that of the objects and message paths on the collaboration diagrams that we used originally to describe the domain (see Figure 9). However, an additional node (PersistenceServerNode) is included, because we decided to model persistence as an externally available service (shown as a lowest-level layer in the component diagram in Figure 14).

### 5.2.3. Replicated hardware

Besides presenting allocation of processes to processors, Figure 17 contains explicit references to user nodes (e.g., workstations) and shows that all server nodes are replicated, but does not specify exactly how many of each are used at any given time. To convey this information, we use a diagram (Figure 18) which shows groups of node instances rather than mere kinds of nodes.
For further elaboration of characteristics of and connections between specific node instances (e.g., primary vs. backup servers), another level of detail could be shown in a separate diagram.

6. Issues

The elaboration of the model presented in this paper created an opportunity to assess the usefulness of several modeling tools and techniques, to assess the convenience of the UML for describing object models, and indeed of the appropriateness of object models to describe services and fault-tolerant requirements. Let us examine these items briefly.

6.1. Usefulness of tools and techniques

The modeling of distributed object systems would be enhanced with explicit UML support for several design concepts that, while used as conceptual design tool, were only implicitly represented in our design:

- **Layered approach:** The elaboration and the description of the model were much simplified by conceptualizing it in a layered manner. The possibility of mapping concepts at a given abstraction level into a (set of) concepts at a “more concrete” level simplifies the model itself, and allows to discover properties that were hidden by the low-level clutter.

- **Convenience of UML stereotypes:** UML users can extend it via stereotypes. This facility can be misused and lead to a proliferation of mutually incompatible UML dialects, but we found that a judicious use simplifies modeling significantly. However, it is not clear yet how to define modified semantics for a stereotype, so we used them as graphical devices only.
• **CORBA-specific modeling**: CORBA has several concepts that do not map well into traditional OOA&D notations, such as refined notions of conformance, object identity, and classes with alternative implementations (for clients and servers). These concepts require the development of some specialized concepts and/or notation; otherwise they remain as patterns in the modeler’s head.

• **Fault-tolerance modeling**: Several concepts widely used to describe fault-tolerant systems are lacking in UML. This is the case with process groups (more generally, with sets that have identity) and multicast.

### 6.2. Appropriateness of UML to model object models

Our modeling exercise exercised several features of UML, hence highlighting several aspects of its appropriateness to represent object models in general:

• **Used/unused features**: Of course, the basic object-oriented concepts (classes, objects, types, association, aggregation, etc.) we heavily used. A separate notion proved to be even more useful to organize our thinking (and model): the “refinement” dependency, which turned out to be extremely useful for mapping concepts between layers (see discussion above). A third UML aspect that seems extremely useful is the ability to customize UML for specific processes, and in this vein we could have gained much from using OOSE’s process-specific stereotypes, allowing us to organize the model to reflect the modeling process itself (however, due to their late publication, we didn’t incorporate them).

• **Missing concepts**: Some high-level design concepts are missing from the UML. For example, sets (as first-class entities) would have been very useful to model replicated servers that act as a single entity. Also, some object-oriented notions are straight-jacketed into a modeling approach that betrays its C++ roots (e.g., sub-classing is considered to imply sub-typing, despite allowing separate classes and types).

• **Stereotypes**: UML stereotypes provide the meta-modeler with large flexibility, and we used it, to model both design concepts (e.g., service connector) and system-specific concepts (e.g., CORBA’s inheritance). Interestingly enough, the main problem is perhaps the opposite: the temptation of introducing an arbitrary amount and kinds of stereotypes, hampering the portability of the models.

### 6.3. Appropriateness of OO models to represent domains

On the issue of using UML to represent problem domains (rather than designing solutions for them), we missed two kinds of features, but welcomed another:

• **Description of non-entities**: Object models allow to describe domain entities with great power, but seem to fail when the aim is to describe other properties, such as downtime constraints or replication degree. Additional notation must be used to denote such properties.

• **Sets**: As mentioned in the previous sub-section, sets are most useful feature to intensally represent “kinds” (non-types) of objects; such definitions would simplify enormously the analysis of some domains.

• **Types**: Types, as separate from classes, turned out to be extremely useful to separate the domain modeling (using types) from the corresponding solution (as classes “representing” such types). This problem/solution spaces separation is veryt appropriate to study object systems.

### 7. Future work

This document presented a non-trivial example of modeling distributed object systems using the UML. Many research venues are possible at this point, and further effort would allow to:

1. Build a complete CORBA prototype of the modeled system:
   - provide a complete example of UML use, from use-case to architecture and deployment
   - acquire familiarity with CORBA programming, and Orbix in particular
   - acquire familiarity with fault-tolerance programming, and Isis in particular
• demonstrate how the architecture changes when the Persistence Service is implemented with each of the three major types of database solutions (i.e., OODBMS, ORDBMS and an RDBMS).

2. Perform additional architecture modeling of the “infrastructure” (i.e., support for fault-tolerance and load-balancing)
   • building a “catalog” of common design that arise when trying to provide fault-tolerance and load-balancing properties for a given service
   • writing design guidelines for use of infrastructure models
   • provided pre-packaged solutions that can be easily embedded in larger designs

3. Further stress, evaluate and collaborate with Rational Software on their implementation of the UML in the Rational Rose product.

8. Conclusions

This paper has presented a UML model for an application supporting services to individual traders in a commodity trading environment. The application has significant requirements regarding distribution and fault-tolerance. The modeling building and presentation is organized into three layers: a domain analysis model, a design architecture, and a packaging and deployment model. The emphasis of the modeling exercise is the design layer, and the emphasis of the design is the representation of services in a CORBA system.

9. Acknowledgments

The authors wish to thank John Wahl for providing domain expertise, and Grady Booch, Cris Kobryn, Beverly (Bev) MacMaster, and Joaquin Miller for their valuable comments and feedback.

10. References


11. Appendix A: Enumeration of features in each figure

11.1. Figure 2: Use case diagram for the problem domain

The use case diagram in Figure 2 captures the following modeling concepts and notation:

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram(^\text{19})</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor</td>
<td>“stick man” figure</td>
<td>trader, marketDataProv</td>
</tr>
<tr>
<td>use case</td>
<td>oval with label</td>
<td>generate contract, trade commodities</td>
</tr>
<tr>
<td>system boundary</td>
<td>rectangle enclosing a set of use cases</td>
<td>large rectangle enclosing all use cases in figure</td>
</tr>
<tr>
<td>stereotype</td>
<td>name of stereotype enclosed by guillemets adjacent to a model element</td>
<td>«extends»</td>
</tr>
<tr>
<td>Relationship: participates</td>
<td>solid line connecting actor and use case</td>
<td>trader participates in the “trade commodities” use case</td>
</tr>
<tr>
<td>Relationship: extends</td>
<td>«extends» stereotype added to inheritance notation (triangle on the base use case end of a relationship)</td>
<td>“distribute trade news” use case extends the “distribute news” use case.</td>
</tr>
<tr>
<td>Relationship: uses</td>
<td>«uses» stereotype on a dashed (dependency) arrow. Source use case includes behavior of target use case</td>
<td>“trade commodities” use case uses the “generate contract” use case</td>
</tr>
</tbody>
</table>

Table 1: Features shown in Figure 2

11.2. Figure 4: Sequence diagram for Trade Commodities

The sequence diagram in Figure 4 captures the following modeling concepts and notation:

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>rectangle with labels of the form (name: class) across top of diagram, attached to a vertical dotted line</td>
<td>tradeAdministrator : TradeAdministrator object</td>
</tr>
<tr>
<td>message</td>
<td>labeled arrow between objects, arrowhead on the target object</td>
<td>“request commodity” message (#1)</td>
</tr>
<tr>
<td>activation (focus of control)</td>
<td>double vertical line on object</td>
<td>steps 1,3,4 on object traderContract</td>
</tr>
<tr>
<td>time sequence of messages</td>
<td>physical placement and sequential numbering of messages</td>
<td>Message “request commodity” (#1) occurs before “analyze potential trade” (#2).</td>
</tr>
<tr>
<td>Message synchronization: balking</td>
<td>message arrow returning on itself (indicates that sender abandons the operation if receiver is not immediately ready)</td>
<td>message #1, “request commodity”</td>
</tr>
</tbody>
</table>

\(^{19}\) Example(s) shown in this column are not necessarily exhaustive, only representative.
### Table 2: Features shown in Figure 4

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>rectangle with labels of the form (name: class)</td>
<td>tradeAdministrator : tradeAdministrator object</td>
</tr>
<tr>
<td>multi-object</td>
<td>multiple offset rectangles (label has the same form)</td>
<td>TradeAdministrator: Trade</td>
</tr>
<tr>
<td>message</td>
<td>labeled arrow between objects</td>
<td>“request commodity” message (#1)</td>
</tr>
<tr>
<td>time sequence of messages</td>
<td>sequential numbering of messages</td>
<td>Message “request commodity” (#1) occurs before “analyze potential trade” (#2).</td>
</tr>
<tr>
<td>Participation of objects</td>
<td>objects exchanging messages</td>
<td>tradeAdministrator, contractAdministrator, and Persistence Service participate in the use case.</td>
</tr>
<tr>
<td>Participant (object) interaction</td>
<td>messages between system objects; arrowhead on the target object</td>
<td>message #7, “save trade.”</td>
</tr>
<tr>
<td>Message synchronization:</td>
<td>message arrow returning on itself</td>
<td>message #1, “request commodity”</td>
</tr>
</tbody>
</table>

**11.3. Figure 5: Collaboration diagram for Trade Commodities**

The collaboration diagram in Figure 5 captures the following (note that there is significant overlap with the representation for the preceding sequence diagram in Figure 4):

- **Message synchronization:**
  - Synchronous: “X” near arrowhead of message
  - Asynchronous: One-sided arrowhead on message
  - Time-out: small circle attached to message

- **Constraint (conditional execution of messages):**
  - Text within braces adjacent to messages
  - Messages with bracketed text:
    - Messages #9-10, “update […]” & #11-12 “new […]”
    - Indicates that an existing slate will be updated OR a new slate will be created and stored, depending on whether the slate already exists.

- **Uses relationship:**
  - Labeling message with name of use case

- **Timing marks:**
  - Free form expression enclosed by braces
  - Example: \(8 + 9 + 10 < 1 \text{ sec.}\) (see left-hand margin)

- **Note:**
  - Rectangle with bent corner (attached to messages or objects with dotted lines), or free form text in margins
  - Notes on event stream implementation; states of use case and description in left hand margin
balking

<table>
<thead>
<tr>
<th>Message synchronization: synchronous communication</th>
<th>“X” near arrowhead of message</th>
<th>message #12, “store slate”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message synchronization: asynchronous communication</td>
<td>One-sided arrowhead on message</td>
<td>message #6, “register trade”</td>
</tr>
<tr>
<td>Message synchronization: time-out</td>
<td>small circle attached to message</td>
<td>message #8, “trade notification.” Indicates that tradeAdministrator will abandon the message (and raise an error) if positionAdministrator cannot handle the message within a specified amount of time.</td>
</tr>
<tr>
<td>Dataflows/ passing of parameters</td>
<td>arrow with circle at the tail, or method name and parameter(s)</td>
<td>message #1, “generate contract” (trade object is passed to contractAdministrator) Alternative would be generateContract(Trade)).</td>
</tr>
<tr>
<td>note</td>
<td>rectangle with bent corner (attached to messages or objects with dotted lines), or free form text in margins</td>
<td>Notes on event stream implementation</td>
</tr>
</tbody>
</table>

| **Table 3: Features shown in Figure 5** |

11.4. **Figure 7: Class diagram for (a portion of) the domain**

The class diagram in Figure 7 captures the following modeling concepts and notation:

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class</strong></td>
<td>box with three compartments, separated by horizontal lines, showing class name (in bold), class attributes, and class operations (last two are optional)</td>
<td>PositionDetail</td>
</tr>
<tr>
<td><strong>visibility of attributes and operations</strong></td>
<td>“-” for private, “+” for public, “#” for protected</td>
<td>attributes and operations of PositionDetail (protected visibility does not apply to PositionDetail)</td>
</tr>
<tr>
<td><strong>abstract class</strong></td>
<td>«abstract» stereotype (using guillemets) in the name compartment of the class; name of class is italicized</td>
<td>Administrator</td>
</tr>
<tr>
<td><strong>association</strong></td>
<td>solid line between classes</td>
<td>TradeLeg is associated with a Trade (Trade has TradeLeg(s))</td>
</tr>
<tr>
<td><strong>association role</strong></td>
<td>text string on the relevant end of the line</td>
<td>Receipt, Delivery roles of TradeLeg on association between Trade and TradeLeg</td>
</tr>
<tr>
<td><strong>multiplicity</strong></td>
<td>numeric range(s) adjacent to the relevant class</td>
<td>Trade may have zero or more Receipt TradeLeg(s) and may have zero or more Delivery TradeLeg(s).</td>
</tr>
<tr>
<td><strong>Relationship: aggregation</strong></td>
<td>Hollow diamond on “whole”, or aggregate, end of the association. (Weaker association than</td>
<td>A TradeLeg is part of a Trade. (Conversely a Trade has TradeLeg(s))</td>
</tr>
<tr>
<td>Composition</td>
<td></td>
<td>Position is composed of PositionDetail(s). (A PositionDetail is not meaningful or accessible outside the context of a Position, but reverse is not true)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Relationship: composition</td>
<td>Solid diamond on “whole”, or aggregate (Stronger association than Aggregation)</td>
<td>A TradeAdministrator is a subclass of Administrator</td>
</tr>
<tr>
<td>Relationship: generalization</td>
<td>large hollow triangle at the end of the association attached to the more general element</td>
<td>TradeLeg is dependent on Price (for computation of its value). Price is dependent on MarketDataService (for instantiation).</td>
</tr>
<tr>
<td>Relationship: dependency</td>
<td>broken line associations with an arrowhead attached to the independent element.</td>
<td></td>
</tr>
<tr>
<td>note</td>
<td>rectangle with bent corner (attached to classes with dotted lines), or free form text in margins</td>
<td>note attached to the Administrator class indicating a requirement for fault-tolerance and load-balancing strategies</td>
</tr>
</tbody>
</table>

### Table 4: Features shown in Figure 6

#### 11.5. Figure 6: State transition diagram for Trade class

The state transition diagram in Figure 6 captures the following modeling concepts and notation:

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>rounded rectangle</td>
<td>States of Trade: Estimated, Actualized, Canceled</td>
</tr>
<tr>
<td>sub-state</td>
<td>physical enclosure of states within another state</td>
<td>Delivery Estimated/Actualized and Receipt Estimated/Actualized sub-states within Estimated state</td>
</tr>
<tr>
<td>concurrent sub-states</td>
<td>physical positioning of substates (Alternative not shown; separate substates by broken lines within the “superstate”)</td>
<td>the substates mentioned above (both deliveries and receipts must be actualized before entire trade is actualized)</td>
</tr>
<tr>
<td>sequential substates</td>
<td>physical enclosure of substates, positioned sequentially with events between them</td>
<td>(N/A)</td>
</tr>
<tr>
<td>pseudo-state (initial)</td>
<td>small solid filled circle</td>
<td>see top of diagram</td>
</tr>
<tr>
<td>pseudo-state (final)</td>
<td>circle surrounding small solid filled circle (“bull’s eye”)</td>
<td>see bottom of diagram</td>
</tr>
<tr>
<td>events (triggered by simple conditions)</td>
<td>arrow between states, with conditions specified in text</td>
<td>“deliveryEstimates=0” (moving from Delivery Estimated to Delivery Actualized because there are no more estimated tradeLeg’s)</td>
</tr>
</tbody>
</table>
| events (triggered by receipt of messages) | event-name(parameter list) | tradeLegActualized(tradeLeg) and tradeLegEstimate(tradeLeg) events on the Estimated state (trade object is notified that a
specific trade leg has been estimated or actualized; could result in changes to substates within Estimated state)

<table>
<thead>
<tr>
<th>action clauses on events</th>
<th>event ‘/’ action-clause</th>
<th>estimateDelivery(...), actualizeDelivery(...), estimateReceipt(...), actualizeReceipt(...) events on the Estimated state (on these events, the action is to increment or decrement the number of delivery or receipt estimates using the appropriate private method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>transition string</td>
<td>event-signature [ guard-condition ] / action-clause ^ send-clause</td>
<td>cancel[All Receipts and Deliveries are estimates] ^ tradeLeg.cancel (indicates that trade cannot be canceled unless all receipts and deliveries are estimates; when this holds, the action is to cancel the individual trade legs as well)</td>
</tr>
<tr>
<td>simple transition</td>
<td>event causing a transition from one state or substate to another</td>
<td>transition between the DeliveryActualized and DeliveryEstimated states labeled with the transition string “estimateDelivery(tradeLeg) [isDeliveryLeg(tradeLeg)] / deliveryEstimates + 1”</td>
</tr>
<tr>
<td>internal transition</td>
<td>Event not causing a transition from one state or substate to another (event arrow)</td>
<td>tradeLegEstimate(tradeLeg)… (attached to a single state of Estimated)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi-object</td>
<td>multiple offset rectangles</td>
<td>tradeAdministrator; TradeAdministrator</td>
</tr>
<tr>
<td>link</td>
<td>line between objects</td>
<td>link between MarketDataService and Trader</td>
</tr>
<tr>
<td>(specific) message</td>
<td>labeled arrow along links</td>
<td>“hearsay” and “news” messages along association from Trader to MarketDataService</td>
</tr>
<tr>
<td>note</td>
<td>rectangle with bent corner (attached to objects or links with dotted lines), or free form text in margins</td>
<td>Notes on requirements for fault tolerance and reliable communication channels</td>
</tr>
<tr>
<td>note stereotype</td>
<td>guillemets above text of note</td>
<td>«requirement» and «constraint» stereotypes</td>
</tr>
</tbody>
</table>

**Table 5: Features shown in Figure 7**

**11.6. Figure 9: Collaboration diagram for services to be provided**

The collaboration diagram in Figure 9 captures the following:
### Table 6: Features shown in Figure 9

#### 11.7. Figure 10: Market Information service (design for the connected entities) and Figure 11: Market Information service (design for the connector)

The class diagrams in Figure 10 and Figure 11 capture the following (some redundant features from previous class diagram are not shown):

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>rectangle with type name and «type» stereotype</td>
<td>Employee, MarketDataService (variants of domain classes)</td>
</tr>
<tr>
<td>parameterized class or template</td>
<td>Small dashed rectangle representing parameters superimposed on upper right-hand corner of the class rectangle</td>
<td>GTIE_Employee (implm)</td>
</tr>
<tr>
<td>parameter for template class</td>
<td>text in small dashed rectangle, of form “name: type” Type is optional</td>
<td>“implm” parameter (omits type)</td>
</tr>
<tr>
<td>refinement</td>
<td>«refines» stereotype on dependency arrow</td>
<td>“represents” dependency from GTIE_Employee&lt;Employee_i+&gt; to Employee (the first is a refinement of the second for implementation purposes. “_i” is a CORBA naming convention)</td>
</tr>
<tr>
<td>binding, bound class</td>
<td>«bind» stereotype on dependency arrow and/or duplicated class name with parameter</td>
<td>dependency from GTIE_Employee&lt;Employee_i+&gt; to GTIE_Employee; the first class is bound to the second.</td>
</tr>
<tr>
<td>pattern</td>
<td>oval with pattern name</td>
<td>EventStream</td>
</tr>
<tr>
<td>implementation-specific stereotypes</td>
<td>user-defined extensions to UML «IDL-generated» and «delegate» stereotypes</td>
<td></td>
</tr>
<tr>
<td>note</td>
<td>rectangle with bent corner (attached to classes with dotted lines), or free form text in margins</td>
<td>Notes on requirements attached to Employee and MarketDataService</td>
</tr>
</tbody>
</table>

### Table 7: Features shown in Figure 10 and Figure 11

#### 11.8. Figure 12: Context of EventStream pattern (implemented via BOAImpl CORBA inheritance) and Figure 13: Context of EventStream pattern (implemented via GTIE CORBA delegation)

The class diagrams in Figure 12 and Figure 13 capture the following (some redundant features from previous class diagrams are not shown):

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>collaboration</td>
<td>class with «collaboration» stereotype</td>
<td>EventStream (large rectangle enclosing others)</td>
</tr>
<tr>
<td>pattern</td>
<td>template collaboration</td>
<td>EventStream</td>
</tr>
<tr>
<td>context</td>
<td>nesting of entities within</td>
<td>All other classes, etc. are inside</td>
</tr>
<tr>
<td></td>
<td>dominant class</td>
<td>EventStream</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>parameterized or template (collaboration)</td>
<td>Small dashed rectangle representing parameters superimposed on upper right-hand corner of the class rectangle</td>
<td>EventStream class (parameters Sender, Receiver, Events)</td>
</tr>
<tr>
<td>conformance to interface</td>
<td>&lt;subtype&gt; stereotype on dependency</td>
<td>Sender must be a subtype of (i.e., conform to) Events</td>
</tr>
</tbody>
</table>

**Table 8: Features shown in Figure 12 and Figure 13**

11.9. **Figure 14: Component Diagram (top level) and Figure 15: Component diagram (Application Tier)**

The component diagrams in Figure 14 and Figure 15 capture the following modeling concepts and notation:

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td>File folder</td>
<td>Desktop Tier package</td>
</tr>
<tr>
<td>active object</td>
<td>rectangle with darkened border with name of object underlined</td>
<td>OrbixD</td>
</tr>
<tr>
<td>physical composition/nesting of packages and modules</td>
<td>Graphical enclosure</td>
<td>The TraderUI and CorporateServer packages nested within the Desktop Tier package; the OrbixD active object nested within the Orbix + ISIS Support package, which in turn is nested within the Utilities and External Interfaces package.</td>
</tr>
<tr>
<td>dependency</td>
<td>dashed arrow from “client” to “supplier” package or module</td>
<td>Desktop Tier depends on Application Tier</td>
</tr>
<tr>
<td>layers (of software)</td>
<td>Dependencies coupled with physical placement of packages (higher packages at higher up on drawing are at a higher level of abstraction than lower packages)</td>
<td>Desktop Tier is at higher level of abstraction than Application Tier</td>
</tr>
<tr>
<td>note</td>
<td>rectangle with bent corner (attached to packages or modules with dotted lines), or free form text in margins</td>
<td>Notes on contents of Application Tier</td>
</tr>
</tbody>
</table>

**Table 9: Features shown in Figure 14 and Figure 15**

11.10. **Figure 16: Component diagram (compilation dependencies)**

The component diagram in Figure 16 captures the following (asterisk indicates extension of notation done for this diagram):

<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>source file</td>
<td>Vertical, white rectangle with compartment for name (C++ source), white parallelogram with overlapping smaller rectangles (IDL source)</td>
<td>PositionAdministrator and TradeAdministrator (C++); IPositionAdministrator and ItradeAdministrator (IDL) h-IpositionAdministrator and h-ItradeAdministrator (C++ header)</td>
</tr>
<tr>
<td>Modeling Concept</td>
<td>Representation</td>
<td>Example(s) in diagram</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>node</td>
<td>3-dimensional cube</td>
<td>UserNode</td>
</tr>
<tr>
<td>connection (a.k.a. communication association)</td>
<td>line between nodes</td>
<td>connection between UserNode and TradeAdministrator node</td>
</tr>
<tr>
<td>process (a.k.a. distribution unit) running on a node</td>
<td>Graphical enclosure of process (shown here as text representing a package or active object; alternative is enclose the package or active object itself)</td>
<td>MarketDataServerNode runs the MarketDataServer and OrbixD distribution units (package and module, respectively)</td>
</tr>
<tr>
<td>node stereotype</td>
<td>guillemets enclosed within node cube</td>
<td>«client» and «server» stereotypes</td>
</tr>
<tr>
<td>connection stereotype</td>
<td>guillemets adjacent to connection</td>
<td>«broadcast» connection between UserNode and MarketDataServerNode (indicates that a single MarketDataServer node can broadcast the same news and prices to many UserNodes at the same time); «point-to-point» connection between UserNode and PersistenceServer (e.g. a UserNode sends messages directly to a Persistence Server node).</td>
</tr>
</tbody>
</table>

Note: rectangle with bent corner (optionally attached to nodes with dotted lines), or free form text in margins

Notes on server replication

Table 10: Features shown in Figure 16

11.11. Figure 17: Deployment diagram (top-level)

This deployment diagram in Figure 17 captures the following:

Table 11: Features shown in Figure 17

11.12. Figure 18: Deployment diagram (node instances)

The deployment diagram in Figure 18 captures the following modeling concepts and notation:
<table>
<thead>
<tr>
<th>Modeling Concept</th>
<th>Representation</th>
<th>Example(s) in diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicity of nodes</td>
<td>number within 3-dimensional cube</td>
<td>6 UserNodes, 3 MarketData and TradeAdministrator Server nodes, 2 Contract and Position Server nodes</td>
</tr>
<tr>
<td>multi-object nodes</td>
<td>multi-object stereotype</td>
<td>See TradeAdministrator node</td>
</tr>
</tbody>
</table>

Table 12: Features shown in Figure 18