ABSTRACT

Title of Thesis: An Extensible Order Promising and Revenue Management Test-bed

Frederick Faber, 2005

Advisor: Michael Ball, Ph.D.
Institute for Systems Research

Over the last quarter century it has become increasingly vital for large-scale businesses to exploit revenue management strategies. This is a consequence of thinning profit margins that have diminished from the effects of several pervasive trends. Among these trends are the globalization of markets, the regulation of particular industries, and, especially in the airline industry, the introduction of heightened security standards. To keep pace with this volatile landscape, revenue management and order promising policies rapidly have become more complex. Correspondingly, testing such policies has become increasingly difficult. This thesis presents an extensible test-bed designed to facilitate testing complex revenue management and order promising policies. The test-bed was modeled and built using an agent-oriented architecture in order to reflect real-life business environments accurately. We describe the design and construction of the test-bed in this thesis. We then present case studies to highlight its capabilities.

An Extensible Order Processing and Revenue Mangement Testbed

by

Frederick Faber

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Masters of Science
2005

Advisory Committee:

Dr. Michael Ball, Chair
Dr. Mark Austin
Dr. Subramanian Raghavan

ACKNOWLEDGMENTS

I thank Michael Ball, Ph.D. for his generosity in accepting me as an advisee. I thank him also for providing guidance and flexibility to me throughout our research.

I thank Zhenying Zhao, Ph.D. for taking the time to explain thoroughly the underlying research concepts related to this thesis. Without his patience during our frequent discussions I surely would have strayed miles from our research objectives.

I thank Mark Austin, Ph.D. and S. Raghavan, Ph.D. for generously serving on my thesis committee. I genuinely enjoyed working with and learning from both during my time at the University of Maryland.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES

Chapter 1

Introduction

Most early developments in revenue management are attributable to research performed in the airline industry. This research stems from as far back as the early 1960's. For several decades thereafter, revenue management remained a science almost exclusively concentrated in the airline business. Eventually, other industries took notice of this work and began to adopt revenue management practices. Some of these industries had more complicated product models than simple airline tickets. For these industries, implementing a successful revenue management strategy included optimizing the order fulfillment process. This spawned the birth of the order promising field, which remains very closely coupled with the field of revenue management.

As interest in revenue management and order promising strategies grew, so did the complexity of the policies involved. The increased complexity made testing new policies a difficult task to perform analytically. As a result, simulation became a popular means of evaluating new policies. Simulation has remained the primary means of testing new policies to the current day. The advantages of this technique are irrefutable. Among these are the ability to product unpredictable demand, the ability to test dynamic policies, and the ability to test policies quickly. Despite such advantages, though, there is one major determinant to using simulation: the burden of creating a new simulation test-bed to test each new policy. This is the problem that this thesis seeks to ameliorate. Specifically, this thesis presents an extensible, agent-oriented test-bed that may be used to evaluate revenue management and order promising policies from a myriad of different industries.

Our solution integrates work from two major fields: revenue management strategies and agent-oriented technologies. An introduction to both fields is given in the following two sections.

1.1 Revenue Management

The revenue management problem is defined succinctly as "selling the right product to the right customer at the right time" [47]. This description captures the following elemental trade-off: A seller has limited and perishable inventory. Once it expires, the value of this inventory drops to zero. A seller also has several classes of customers, each of which pay a different price for the same product. Further, higher paying customers may place orders after the seller receives less valuable orders. Therefore, a seller must trade-off the revenue benefits realized by maintaining a reservation inventory for higher paying customers (while in the process denying lower paying customers), versus the risk of owning this inventory when it expires. This, in simplified terms, is the goal of an effective revenue management policy [12].

The problem of order promising under uncertainty is essentially a sub-set of revenue management. Specifically, it describes how customer orders are processed and fulfilled. Both problems have been studied in a wide breadth of industries, but the pioneering movement originated primarily from the airline industry. As far back as the early 1960's revenue management policies were introduced for use in airline reservation systems. Of particular significance was the introduction of Super Saver fares from American Airlines in the spring of 1977; this not only represented the launch of an enormously successful revenue management campaign, but effectively advertised the

irrefutable worth of investing in revenue management research to all industries. [1]

For the revenue management challenge to be present in a business environment, several requisite conditions must hold. Among the more critical elements are the following [12]:

(i) Perishable Inventory - The value of inventory disappears after an expiration date (e.g. the value of a ticket to a sporting event, after that event concludes).

(ii) Stochastic Demand - Demand is not entirely predictable, and may be sensitive both to exogenous factors (e.g. seasons) and endogenous factors (e.g. price).

(iii) Variable Pricing Structure - Different classes customers pay different prices for the same product.

(iv) Reservation Booking in Short-Selling Horizons - Orders may be taken for a product that will be delivered in the future (e.g. booking an airline ticket).

While these conditions may seem limiting, there is a large number of industries that satisfy them. Several examples include the following: consumer transportation (airlines, railways, rental cars), hotels, restaurants, clothing retailers, and Internet bandwidth providers. In this sample set alone, it is obvious that each industry has a different set of characteristics. This has led to the creation of a vast number of unique revenue management policies, most of which are specific to a particular industry. This has also led to the development of several new testing frameworks, many of which are created to test only one policy. This issue, the uneconomical creation of redundant test-beds, is exactly what this thesis addresses.

The effort of creating a unique simulation framework to test a new revenue management policy traditionally has been little more than experimental overhead. This work, however, is necessary to predict the potential economic impact of deploying a revenue management policy. Further, in environments with many significant uncertainties, such as stochastic demand, simulation approaches become increasingly valuable versus analytic predictions. Moreover, computing advances

[1] American Airlines had launched revenue management initiatives previously, but these dealt primarily with overbooking, a concept specific to the airline and hotel industries. Super Saver fares introduced customer segmentation into revenue management, which is a concept that can be applied to a much larger scope of industries [47].

in recent years have enabled the feasible creation of revenue management policies that actually are *developed* through the use of simulation (see [8]). Hence, for both evaluation and calibration of revenue management policies, there is a substantial need to employ simulation.

1.2 Agent-Oriented Software Design

In the field of Software Engineering, there are many varied and disharmonious views of the definition of the term, "Software Agent." This thesis does not present an exhaustive list, or attempt to argue the pros and cons of each definition. Instead, a working definition is established below for use throughout.

In the context of this thesis, an agent is defined as popularized by Russel and Norvig to be [45]: *"... anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors"*. To focus this definition specifically to software agents, additional context must be applied. Firstly, the "environment" must be limited to the simulation system; this is fairly intuitive. Secondly, it is necessary to impose a condition that a software entity should possess at least one of several distinct attributes to be labeled an agent. These attributes are:

(i) Adaptiveness - The agent may change its behavior based on acquired knowledge or experience.

(ii) Autonomy - The agent may exercise control over its own actions.

(iii) Communication - The agent may establish discourse with other agents.

(iv) Goal Orientation - The agent acts to achieve a particular objective.

(v) Mobility - The agent may transport itself across different physical systems.

(vi) Pro-activeness - The agent may initiate actions on its own, without receiving directives to do so.

(vii) Reactiveness - The agent may respond to stimuli in its environment.

(viii) Understanding (Abstractness) - The agent interprets instructions at the *knowledge* level versus the step-by-step declarative level.

This set is an aggregation of attributes that are commonly found in software agent literature (see [21] and [28]). It is important to recognize that a software entity does not necessarily need to possess each of these attributes to be labeled as an agent. For example, a software agent may be reactive and adaptive, but immobile. The unique combination of attributes embodied by an agent provide a means for further classification into agent taxonomies. This is described in detail by Franklin and Graesser (1996) [21].

Having established a working definition of an agent, the larger picture of agent-oriented design needs to be addressed. This thesis relies heavily on this concept, and so a detailed introduction is provided below.

Through the evolution of the field of Computer Science, several different programming paradigms have been introduced. Examples of such approaches include monolithic programming, modular programming, functional programming, and object-oriented programming. Each new approach has several defining characteristics that identify it from its ancestors. Theoretically, these characteristics engender a level of productivity that improves on established programming techniques. This argument seems sensible under the assumption of progressive evolution. That is, the claim that as software engineering techniques evolve, the total the yield of productivity from each technique will improve correspondingly. This, unfortunately, is not the reality in practice. Increasingly exotic and complicated problem domains, along with other non-trivial phenomena, instead have asymptotically limited the productivity gains of software engineering innovation. [2]

Despite these hindering factors, innovation in software design is not for naught. This is because it is possible to realize substantial gains in efficiency by applying an appropriate solution architecture to a problem domain. Moreover, greater levels of abstraction in solution methodologies help to flatten the steep engineering learning curves. This effect also contributes to more efficient solutions. Hence, the development of new engineering techniques constantly is broadening the

[2] There is a clear reference here to Brook's renowned work [14].

arsenal of solution designs available, and in turn, empowering engineers to solve different and more complex problems.

Agent-oriented software design, an adolescent practice in software engineering, is no exception to this rule. As such, there are distinct classes of business models to which applying agent-oriented solutions is decidedly advantageous. Further, there are engineering design issues that should be considered when selecting a modeling approach. Much like defining a software agent, though, it is difficult to identify a set of criteria that is universally accepted. In lieu of attempting to do this, a list of the primary reasons an agent-oriented design is used in this thesis is given below.

(i) Complexity of Business Scenario - The business scenarios simulated in this thesis represent complex systems. Briefly, these are systems that are comprised of many interacting units, exhibit top-level behavior that is not necessarily deducible by the behavior of its components, and are often decomposable into sub-complex systems ([5], [53]). Agent-oriented design has been shown to be an especially advantageous design technique to model such systems; the particular reasons are outlined elegantly in Jennings's 1999 work (see [28].

(ii) Scalability of Simulation Model - Scalability is a principal requirement in the design and construction test-bed presented in this thesis. Agent-oriented systems tend to provide a large measure of scalability.

(iii) Representation of Logic at the Knowledge Level - A primary design goal of our test-bed is to allow for complex logic to be implanted into its agents components at an abstract level. As illustrated by Russell and Norvig in [45], such installation of top-level knowledge into dynamic software components is readily realized through the use of software agents.

For the reasons outlined above, an agent-oriented design is an appropriate methodology for the design of our test-bed.

1.3 Research Motivation & Thesis Objectives

The overarching goal of this thesis is to implement an extensible revenue management and order promising test-bed to simulate wide array of business scenarios. This test-bed, in turn, is to aid in the creation and evaluation of new revenue management policies, which also include order promising under uncertainty. In order to meet these ends, several prerequisite objectives must be met.

The first objective is to develop a conceptual software model to represent the main components of a business that are involved in its revenue management policy. This is done to capture the business requirements of the test-bed. A sub-goal of this objective is to ensure this software model is as robust and general as possible. Doing this will allow the test-bed to evaluate revenue management policies in many different business scenarios.

The second objective is to implement this software model using an agent-oriented architecture. This includes the entire development process, from designing and planning, to implementing and testing.

The third objective is to use the test-bed to evaluate several order promising and revenue management policies. This involves executing new experiments, as well as duplicating published experiments. This is done to exhibit the robustness and flexibility of the test-bed. This is also done to the illustrate the business insight that may be gained by analyzing experimental results.

1.4 Contributions

This thesis proposes a new agent-based simulation infrastructure to support policy decision design and analysis in revenue management and order promising. This thesis also presents the design of several new business agents. Additionally, this thesis implements the agent-oriented design it establishes using the JAVA^{TM} based simulation package, RePast [42]. A series of experimental results are also presented to evaluate the effectiveness of different order promising and revenue management policies.

In practice, this software has been used to support various research activities in the eMarkets

lab at the University of Maryland. This includes efforts in order promising techniques and electronic auctions. Further, several custom packages from the software have been extracted and shared with other researchers for use in other projects.

1.5 Thesis Overview

In Chapter 1, the main objectives of the thesis are outlined. Chapter 2 provides a review of literature relevant to the content of the thesis. An explanation of the design and implementation of the test-bed is given in Chapter 3. A series of case studies, and their results, are described in Chapter 4. Finally, Chapter 5 completes the thesis by offering concluding remarks and potential future extensions.

Chapter 2

Literature Review

This chapter presents an overview of selected literature that highlights work related to this thesis. This consists of three sections: 1) literature illustrating the trend of the increased flexibility of recent order promising policies; 2) literature highlighting the breadth of industries across which revenue management policies are utilized; and 3) literature describing successful implementations of agent-based simulation.

2.1 Flexibility in Order Promising

Recent literature on order promising policies highlights the emerging trend of increasing the level of flexibility given to the customer when placing orders, and to the manufacturer when fulfilling orders. This trend is identified and emphasized in this thesis as it illustrates the need for robustness within a generic test-bed. Traditionally, this flexibility is decomposed into the following two dimensions: quantity and configuration. Quantity flexibility refers to the ability of a manufacturer to fulfill an order only partially, within some given lower bound. This type of flexibility is often present in supply contract models, as is illustrated by the in-depth review authored by Tsay, Nahmias, and Agrawal [48]. The second type of flexibility, configuration flexibility, refers to combination of "demand substitution" and "reactive substitution", as defined by Balakrishnan and Geunes [3]. The order promising models that incorporate this flexibility allow customers to select more than one supplier for at least one entry on the Bill of Materials (BOM) of the order. This allows the manufacturer to select the supplier to use based on availability at fulfillment time, hence the term "reactive substitution."

In addition to these two traditional dimensions of flexibility, a new, third dimension has been introduced recently by Zhao, Ball, and Kotake: This dimension is flexibility relative to the due date of the order [57]. This allows customers to specify an entire range of acceptable due dates, versus the traditional method of supplying only one.

Also involved in the order promising challenge is the decision of how frequently to process orders. This essentially is equivalent to the challenge of deciding how many orders to process at a time. One extreme is to process each order as it arrives; this is essentially real-time processing. The other extreme is to process orders only once each day. The advantage of the latter strategy is that it allows the manufacturer to prioritize orders from its higher paying customers. The trade-off of this batching is, of course, response time to the customer. This consideration, and the effects thereof, are addressed extensively in recent works by Chen et al. (see [16] and [17]). These studies provide a comprehensive analysis of the Available-to-Promise problem, and are suggested as valuable references to the interested reader.

2.2 Breadth and Simulation in Revenue Management

The large quantity of literature addressing applications of revenue management policies illustrates the enormous range of industries across which revenue management is studied. A sample set of this literature is given in this section. A brief survey of the importance of simulation in revenue management is also provided. The objective of providing overviews of both subjects is to emphasize that a generic revenue management simulation test-bed is indeed a valuable tool.

The science of revenue management has its roots in the transportation industry, particularly the airline industry. Breakthrough papers painting its evolution include the pioneering work on overbooking by Rothstein in 1971 [43]; the introduction of the Expected Marginal Seat Revenue (EMSR) model by Belobaba in 1989 [6]; and the classic American Airlines study authored by Smith et al. in 1992 [47]. Additional revenue management works covering the transportation industry are surveyed in McGill and van Ryzin's comprehensive taxonomy, presented in their 1999 work [39].

Current research in revenue management has expanded to many other, non-traditional,

industries. Included among these are the following: the catalog retail industry (e.g. Bitran and Gilbert [10]); the store-front retail industry (e.g. Bitran and Monschein [11], and Coulter [19]); the restaurant industry (e.g. Bertsimas [7], Kimes et al. [31], [32], and [33]); the hotel industry (e.g. Bitran [9]), the broadcast industry (e.g. Bollapragada et al. [13]), the vehicle rental industry (e.g. Geraghty and Johnson [23]), and even the recreational golfing industry [34]). Such a wide breadth of research has spawned many unique revenue management policies; consequently, a test-bed that is designed to accommodate all such industries must possess a great deal of flexibility.

Current research also highlights the widespread use of simulation in the revenue management field. In a 2002 paper by Weatherford simulation is used to produce evaluate revenue management policies over a range of scenarios in which realistic fare data was used [52]. This is in contrast to the typical evaluation process, in which necessarily a hierarchal structure of monotonic increasing fares are used (i.e. the fare for a higher-priority fare class is always greater than or equal to the fare of all lower-priority fare classes). In this instance, the use of a parameterized simulation provided the means by which realistic fare data was generated.

Oliveria uses simulation to study the potential consequences of several airlines applying a revenue management to one particular route [41]. The outstanding value of this simulation is its allowance to include customer preference into the behavioral algorithms of the model. As the author writes, "(the model) permitted the development of crucial details, mainly on the issue of the rationality of the passengers." The simulation also simplified the inclusion of additional factors into the revenue management evaluation, such as probabilistic reservation booking, and customer demand segmentation.

Kimes presents a study performed in 2004 in which a simulation program is used to enumerate different restaurant layouts in order to redesign a popular Mexican restaurant [33]. In this instance, the simulation itself was used to develop a new revenue management strategy. Results of this procedure were successful; a 5.1% increase in yearly revenue was measured following its implementation.

In Healy's 1991 work, a simulation is used to test a revenue management model in order

to determine optimal production lot sizes [25]. Several simulation runs were used to generate a performance function, which was then optimized using analytical techniques.

Skugge uses revenue management simulation to perform a slightly different function: It is used as a feedback mechanism to help to train managers to perform manual adjustments to revenue management policies [46]. This helped expose trainees to the capricious nature of the real-world environment of their business. As Skugge concludes, performing such a revenue management training program also will help "to reduce the black-box mentality surrounding revenue management," which in turn will promote a greater business understanding among employees of the key variables in a firm's revenue performance.

In Ruiz-Torres et al.'s 1998 research, real-time simulation is employed to assign due dates on logistic-manufacturing networks (note that he speaks more specifically to the order promising concern than to the top-level revenue management problem) [44]. The authors conclude that the use of simulation will be beneficial to their client for two major reasons: 1) the performance of due date assignment will improve by using real-time information about endogenous system resources and exogenous system variables; and 2) the simulation system will provide due date options to customers, which in turn will introduce priority pricing into the manufacturing networks.

Umeda and Jones simulate an entire supply chain, including the different business process involved, material flows, and information flows [49]. Doing this created an illustration of the complex relationships among these sub-systems. Exploring such relationships is an important initial step in identifying the driving variables behind a firm's revenue, and consequently in implementing a revenue management policy.

An exciting application of simulation is introduced by Bertsimas and de Boer in their 2003 work, in which simulation is used to derive policy parameters for an airline revenue management strategy [8]. The crux of the innovation in this work is the use of simulation to consider both network effects of multiple-leg flights, as well as key environment variables, such as demand uncertainty and nested fare classes. Traditionally these two factors were examined separately; this is a result of the complexity of considering the two simultaneously. By use of simulation, however, the

combined problem became considerably more tractable. A follow-up effort that addresses several of the pitfalls of the initial proposal is documented by van Ryzin and Vulcano [51]. In its conclusion, the authors laud the simulation solution approach for several reasons, among which are the following: 1) The simulation approach incorporates the true network effects of the system (versus relying on approximations); 2) The simulation method allows for completely general demand processes (versus approximations of the demand distribution); and, 3) The simulation provides "promising results" for large, real-world networks.

Goren investigates the impact of employing revenue management in the presence of low-fare carriers using simulation [24]. An existing airline revenue management simulator, the Passenger Origin-Destination Simulator (PODS), is used in this study. [1] The results of this study illustrate the detrimental effects on the revenue of larger airlines' caused by the entry of low-fare carriers into the market. Eguchi et al. (2004) extend PODS in order to examine the effects of installing revenue management onto an entire market [20]. Specifically, PODS was modified to incorporate group passenger demand and booking; its original simulated only booking and demand of individual passengers. The authors use their modified PODS to investigate the impact of revenue management on airlines in the domestic market of Japan. The use of simulation provided the authors with clear insights into the benefits of using revenue management methods in a complex, realistic environment. They are, in fact, emphatic in their conclusions that, in the presence of fundamental optimization tools (such as simulation), airlines should no longer consider the complexity of their business environment as a significant deterrent to implementing revenue management policies. In this light, the use of simulation is promoted from a post-implementation evaluator, to a tool capable of reducing the barriers of entry to revenue management. Indeed, this is the general trend that is presented in this section. And it is this espousing of revenue management to simulation that provides the motivation for the development of an extensible and flexible revenue management test-bed.

[1] The Passenger Origin-Destination was originally developed at Boeing.

2.3 Agent-Based Simulation

The literature reviewed in this section offer several examples of successful implementations of agent-based simulation. This is given to instill confidence that the use of an agent-based design in this thesis is an innovative, yet vindicated decision.

Galitsky and Pampathi create software agents to perform deductive and inductive reasoning [22]. [2] The agents are applied to the challenge of processing a collection of customer complaints for a particular company. To address this challenge, the agents predict the possible actions a company may undertake in order to resolve its complaints. This work provides a clear illustration of the ability to use logic and reasoning in multi-agent simulation. This ability is also evidenced by the work of Andronache and Scheutz [1]. They present an architecture framework, "Activated-Processing-Observing-Components" (APOC), that facilitates the development of complex agent architectures. The construction of APOC was motivated by the need to simply the development of systems that utilize a particular agent architecture. An example of such a candidate architecture is the Soar framework, developed by Laird et al. [35]. Soar provides an integrated architecture for the development of knowledge-based agents that have the ability to problem solve, learn, and interact with their environment. Soar is a powerful tool, but development with Soar is often complicated. This is the case with the majority of architectures addressed by APOC, hence the main driver behind its creation. The relevance of APOC to this thesis is important. The motivation to construct APOC provides evidence of the widespread, successful, and continual development of intelligent agent frameworks. This history of others' success in using agent frameworks therefore acts to bolster the justification behind using such a framework in this thesis.

Buccafurri et al. discuss an agent-based hierarchal clustering technique for the use of product recommendation on e-commerce web portals [15]. In this work, agents are used to learn about the customers of an e-commerce site. This knowledge becomes the basis on which the agents recommend different products to customers. This work highlights an application of software agents in which the agents must classify unstructured information by observing their surrounding environ-

[2] Specifically, the inductive reasoning included measures of abductive and analogous reasoning.

ment. A related piece authored by Kehagias et. al in 2004 details an agent-based system in which recommending agents are used to offer product suggestions to customers. In this work, additional agents are used in coordination with the recommending agents. For instance, a separate agent is used to mine historical data in a legacy Enterprise Resource Planning (EAR) system. This data is then fed to other agents in order to infer relationships among customers. The complex coordination among the agents in the architecture presented in this paper illustrate how several disparate agents may cooperate in order to fulfill a singular system goal.

An increasingly popular application of agents is as bidders in electronic auctions. An annual competition, the Trading Agent Competition (TAC), provides a substantial amount of literature on this topic. For example, Cheng et al. describe the internals of their competing agent in their 2003 paper [18]. This is just one reference of the ability of agents to be used in a dynamic trading environment. The interested reader may find several others the TAC home page. [3]

Supply chain management systems often include bidding auctions as a means of procurement. In modern e-commerce systems, the bidding may be performed by software agents. A particular type of bidding agent, a risk-adverse agent, is explored in a paper by Liu et al. [36]. This type of agent executes a utility theory-based supply chain bidding strategy. This work offers a practical example of how agents can be used to model a very human-like algorithm. A related application is the paper by Julka et al. [29]. In this work, an agent-based framework is proposed to model, monitor, and manage a refinery supply chain. This is an illustration of the ability of an agent framework to mimic a large-scale business system.

A simulation tool that is related the test-bed presented in this thesis is described in a Master's Thesis written by Morris in 2001 [40]. Morris presents an agent-based simulator that is used to evaluate dynamic pricing strategies over a range of market conditions. This tool, however, focuses specifically on product pricing and does not provide the ability to test other components of a revenue management strategy, such as order promising and inventory control. To the author's knowledge, this dynamic pricing simulator is the most similar tool to the test-bed presented in this

[3] `http://www.sics.se/tac`

thesis. As there are distinct differences between the two, we are confident that our test-bed is a unique tool, and that its capabilities are not available in existing test-beds.

Chapter 3

Agent-Based Test-bed Design

This chapter describes the design and implementation process of our test-bed. It follows a traditional software development life-cycle sequence: First, the business requirements are defined. Next, a logical system model is created. Following this, detailed behavioral and structural models are established. Implementation of the model follows this step. Finally, the model is tested as a complete system.

3.1 Business Environment Modeling

A primary function of our test-bed is to simulate revenue management and order promising decisions for a wide breadth of business scenarios. We examined the structure of many different industries and extracted a collection of common patterns among each. We used this collection to create a common, generic mold into which each industry fits. This formed the blueprint of our test-bed, which is the basis of its implementation.

3.1.1 Primary Business Actors

The generic business blueprint on which our test-bed is modeled is illustrated in figure 3.1. This model identifies the required actors that must be implemented in our simulation framework. [1]

The first actor included in this generic business model is the Customer. Each Customer belongs to exactly one demand class. [2] The primary function of the Customer is to place orders.

[1] A similar model is presented in Bitran and Caldentey's 2003 work [12], but their focus is more on pricing models than on simulating entire business environments.

[2] See section 5.1 on page 108 for possible relaxations to this constraint.

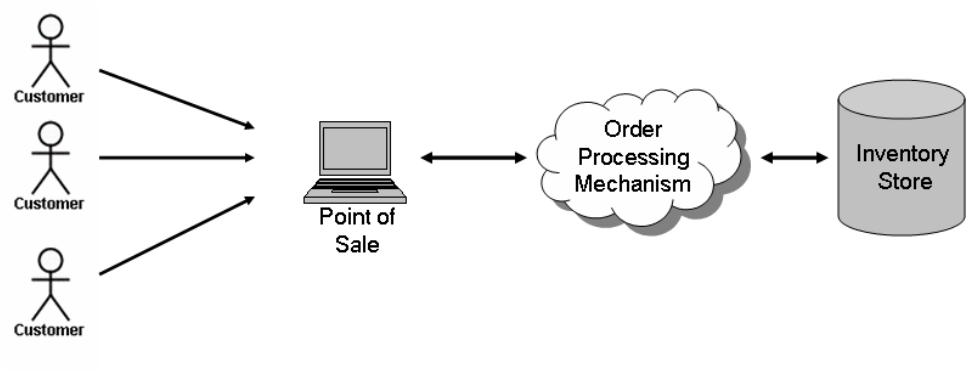

Figure 3.1: Generic Business Model

Not surprisingly, this actor may represent a human customer in a business scenario. A Customer may also, however, represent other purchasing entities, such as an automated bidding agent in an auction scenario.

The second actor in the model is the Point of Sale Device. This represents the entity that receives orders from the customer. This entity also acts as a source of orders to the manufacturer. Effectively, it provides a level of indirection between the customer and the manufacturer. Examples of real-world entities include a central reservation system of an airline, a computer terminal used to submit an online order, and a human reservation agent at a hotel.

The third actor in the model is the Order Processing Mechanism (the Order Processor). This represents the order promising unit of a manufacturer. This mechanism proactively fetches orders from a Point of Sale Device. It also communicates with one or more Inventory Stores to request inventory.

The last actor in the model is the Inventory Store. This is the component that stores and controls the inventory that is promised by the Order Processing Mechanism. The inventory itself may take several different forms. For example, inventory may represent finished products, such as pre-built computers. Inventory may also represent component SKU's, such as individual micro-chips. Airline tickets may also be represented as inventory in this generic model. An Inventory Store, therefore, may also represent several real-life entities, among which are retail stores, warehouses, and electronic databases.

3.1.2 Representation of a Generic Product

A chief feature of our test-bed is its ability to simulate a large number of business scenarios. Many of these scenarios involve the sale of unique products. To avoid multiple, and perhaps redundant, product implementations in our simulation, only one generic product is used. This follows a product model that is extremely simple, yet flexible enough to represent many types of real-world products. Its main components are the following:

(i) Product Name - A human-readable name

(ii) Product Type - An indexable, numeric code

(iii) Product Base Price - A base monetary value (used by retailers to mark-up products)

For example, to accommodate an airline ticket, this model may be populated as follows: Name = "Round-Trip Ticket to LAX", Type ="44", and Base Price = "$800." Very similarly, this model accommodates atomic products (simple SKU's), such as simple electronic components. This model also accommodates recursively defined products, such as assembled electronics. In order to do this, a manufacturer references a lookup-table that indicates which sub-components comprise a super-component. For instance, a product with "Type = 44" may be defined as an assembly of products with types = "40, 41, and 42." For a manufacturer to promise product "44", it first must identify the necessary sub-components, and then proceed with its order promising policy. By providing this flexibility, this product model provides the extensibility required to represent an expansive range of real-life products.

3.1.3 Structure of a Customer's Order

An essential part of our business model was the structure of the customer order. This structure varies widely in different business environments. It therefore was beneficial to create a one generic representation of an order, versus creating many specific implementations.

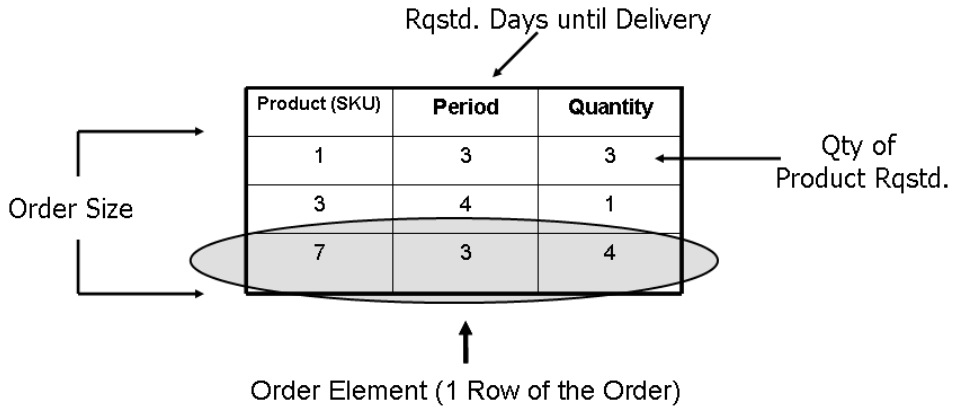

Figure 3.2: Structure of an Order

Figure 3.2 provides the logical representation of the order structure used in our design. The main components to this structure are below:

(i) Product(SKU) - This maps to the Product Type of the generic Product model. Often this simply represents a single SKU.

(ii) Period - The duration within which the product is requested to be delivered. For instance, a period of 3 days indicates the customer requested the product to be delivered within 3 days from the time on which the order was received.

(iii) Quantity - The number of units requested of a particular product.

In this model, one order may contain several rows, or *order elements*. A manufacturer may consider all the order elements collectively; alternatively, a manufacturer may treat each order element individually, almost as if each represented a separate order. This decision refers to the the order promising decision called "splitting," and is described in detail as a case study in section 4.6. Additionally, a customer may request that separate order elements be delivered by different dates (i.e. the value of the period of each order elements may be different). This decision, and the decision to split orders, are both addressed by the a manufacturer's order promising policy.

As a point of reference, the size of an order is defined to be the number of order elements it contains. This is an important attribute of a particular demand distribution.

3.2 Test-bed Infrastructure Design

The next step taken in the implementation of our test-bed was the creation of a system-level design. This section details the software structure and the system behavior of the test-bed.

3.2.1 High-level System Design

Creating a high-level system diagram was the first step taken in creating a software design. This diagram helps to identify and to organize the major software components that comprise the test-bed. It is shown as figure 3.3. This figure illustrates only the major software components, and does not illustrate all the relationships among them. Its main worth is to show the first step in translating from the business domain to the software domain. As such, the major components it includes, and the business entities they represent, are described below:

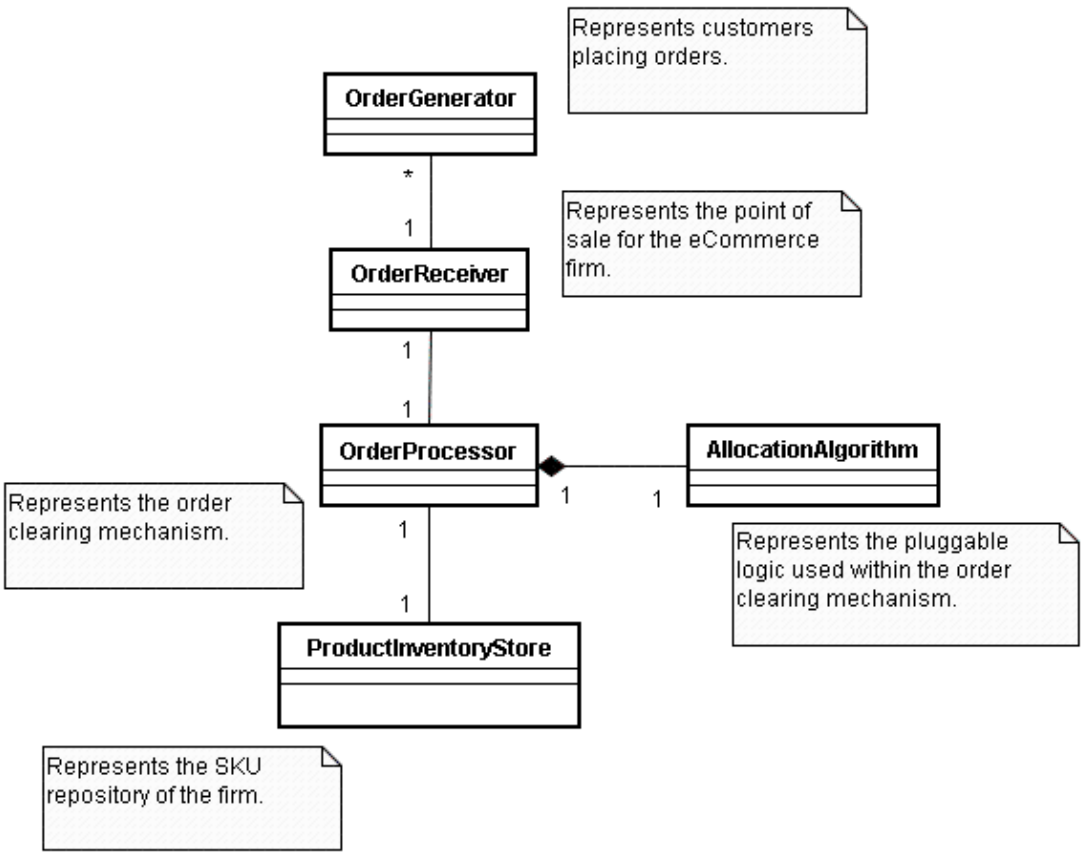

Figure 3.3: Major System Components

(i) `OrderGenerator` - This component represents the Customer. It must generate orders to simulate the demand of different business scenarios. This demand often is modeled after pre-assigned probability distributions, but more uncertain and non-deterministic demand models also may be used.

(ii) `OrderReceiver` - This component represents the Point of Sale Device. It must accept orders from `OrderGenerator`s and make them available to `OrderProcessor`s.

(iii) `OrderProcessor` - This component represents the Order Processing Mechanism. It must decide whether to deny or fulfill customers' orders.

(iv) `AllocationAlgorithm` - This component represents the revenue management policy that is used by a manufacturer to determine how to process customers' orders. This is disconnected from the `OrderProcessor` to allow for a cleaner separation of concerns in the system design.

(v) `InventoryStore` - This component represents the Inventory Store. It must manage and store the inventory that is allocated by an `OrderProcessor`.

3.2.2 Agent Design

We identified three components from the high-level system model to implement as agents. These are the `OrderGenerator`, the `OrderProcessor`, and the `InventoryStore`. Descriptions of each are given below. The goal of providing this is two-fold: 1) to elucidate further the structure of our test-bed ; and 2) to demonstrate that these components have the necessary attributes to be defined as agents.

Design of the `OrderGenerator` Agent:

An `OrderGenerator` agent represents the customer in the business-level blueprint of our test-bed. Its primary role is simple: to place orders. It performs this autonomously. That is, it places orders proactively, and not based on the command of another entity. Also, an `OrderGenerator` agent is able to adapt to its environment. As such, it may alter its behavior rationally to meet a particular

goal (e.g. to maximize its order allocation rate). This behavior is not at all required to be deterministic, and so an `OrderGenerator` agent is not always a predictable entity. Finally, the physical location of an `OrderGenerator` agent need not be static; the interaction of an `OrderGenerator` with other agents in the system is identical whether it be mobile or immobile.

Design of the `InventoryStore` Agent:

An `InventoryStore` agent represents the business entity that manages and restocks the products (inventory) that are ordered by the customers. Its primary roles are to procure this inventory, and to negotiate its assignment with an `OrderProcessor` agent. [3] An `InventoryStore` agent relies on its inventory control policy to determine how to restock its inventory. This policy may be deterministic, such as an Economic Order Quantity (EOQ) model. The policy may also be non-deterministic, and be updated dynamically on-the-fly. This autonomy allows the agent to adjust its behavior to realize its individual goals. For example, an `InventoryStore` agent may lower its safety stock levels if it determines it is incurring a large holding cost penalty.

Design of the `OrderProcessor` Agent:

Of the three, the `OrderProcessor` agent is the most complicated agent in the system. This is because is has two major roles: to promise orders, and to look up its products from an `InventoryStore` agent. As an order promiser, the `OrderProcessor` agent interprets customers' orders, and then decides whether to fill or to reject them. This process is performed proactively, and autonomously. The logic by which this process is performed is internal to the agent itself. An `OrderProcessor` agent relies exclusively on its `AllocationAlgorithm` to determine how to process customer orders. This algorithm may range from entirely predictable, to largely nondeterministic. Consequently, so may the promising behavior of an `OrderProcessor` agent.

After an `OrderProcessor` agent decides to fill an order, it must look up the required inventory from an `InventoryStore` agent. The methods by which this is executed, as well as the deci-

[3] Typically, this negotiation is one-sided: the `OrderProcessor` simply appropriates the inventory it needs. The framework, however, provides the means for an `InventoryStore` agent to deny such requests.

sions made during this execution, are determined solely by the agent's internal `AllocationAlgorithm`. Much like when it promises orders, the agent has complete control over this process.

The goals of an `OrderProcessor` ultimately define its behaviors; both notions are embedded in the the revenue management policy it employs. Typically, the primary goal is to optimize revenue. To realize this end, an `OrderProcessor` agent may adapt dynamically to its environment. For instance, it tactically may change its order promising policy to adjust for spikes in demand. This ability to adjust behavior in pursuit of a goal is an essential agent-like quality of a `OrderProcessor`.

3.2.3 Object-Oriented Motivation

The most critical, and perhaps most equivocating, aspect of the implementation is this: The logical software design is agent-oriented (AO), but the underlying implementation is object-oriented (OO). Three primary reasons explain why this is.

First, there are a plethora of well-tested and effective tools for OO development. This is hardly the case for AO development. Examples of OO tools include mature programming languages, Integrated Development Environments (IDE's), testing frameworks, and performance profiling tools.

Second, efficient OO development techniques have been studied since the late 1960's. [4] Only since the mid 1990's, however, has the study of efficient AO development techniques attained widespread consideration. This is despite the adoption of agents in a handful of esoteric fields, such as Artificial Intelligence (AI) as early as the late 1970's [27]. [5] Examples of established OO development techniques include programming to abstractions, applying design patterns, and refactoring existing code.

Third, there exist several powerful frameworks that facilitate mapping an agent-oriented design onto object-oriented software. Building on top of such a framework allows the developer to exploit the advantages of object-oriented techniques, while enforcing an agent-oriented design. In

[4] This corresponds to the introduction of Simula-67 in 1967.

[5] Esoteric at the time, that is. Currently, many AI studies are much closer to, if not in, the mainstream.

fact, this approach is so advantageous that Wooldbridge and Jennings cite the failure to exploit it several times when they identify their 'Pitfalls of Agent-Oriented Development" [55]. One such pitfall is written as, "(the software developer) spends all (his) time implementing infrastructure." A consequence of not avoiding this pitfall is that time and effort are wasted on design and implementation overhead. This detracts directly from the implementation of the more interesting aspects of the system, such as the agent behavior itself. A second pitfall is given as, "(the software developer doesn't) exploit related technology." This point also speaks to the issue of spending time and effort implementing supporting technology, rather than creating specific agent functionality. It differs slightly from the first in that it refers more to implementation specific technology, but nonetheless is stresses the same principle.

The reasons described in this section highlight the key drivers in our decision to implement our test-bed using an object-oriented development methodology. The next section give details on how this was achieved.

3.2.4 Object-Oriented Design

The high-level business model diagrams illustrate the entities of the business environment that are necessary to include in the test-bed. Effectively this represents a translation from the real-world domain into a conceptual software domain. To implement this conceptual design with a particular software development methodology, another level of decomposition is required. This decomposition translates the conceptual software components into the definitive building blocks that comprise the final system. The important point here is that this translation is different for each different software development methodology. For example, implementing a model using a functional programming methodology requires a different design than does an object-oriented methodology. As discussed in the previous section, we used object-oriented methodology to construct our test-bed. Correspondingly, we translated the conceptual software model into a a series of object-oriented class diagrams. One such diagram is presented in figure 3.4.

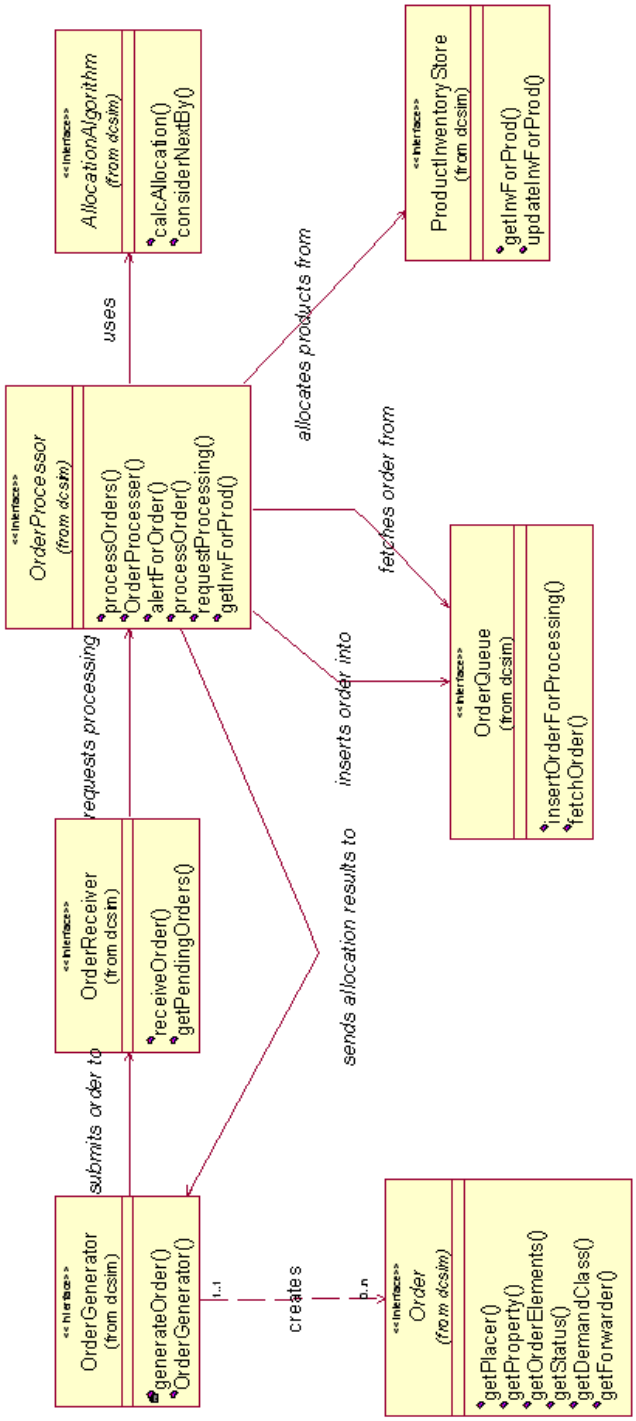

Figure 3.4: Major System Components (Detailed)

One advantage of an object-oriented approach is that it often allows one-to-one mappings from real-world entities to classes. [6] An example of this is the `OrderGenerator` class: This class realizes the `OrderGenerator` component in the conceptual software design. The `OrderGenerator` component then maps, in a one-to-one relationship, to a real-world customer. It is convenient, and intuitive, for humans to operate within this methodology.

A second interesting point is the relationship between an `OrderProcessor` and an `AllocationAlgorithm`. An `OrderProcessor` uses exactly one `AllocationAlgorithm`, yet they are disjointed entities. This effectively separates the revenue management logic from the mechanics of the order promising operation. The benefit of this is substantially increased efficiency: `AllocationAlgorithm`s can be plugged into `OrderProcessor`s with a great deal of ease. Further, to test new revenue management strategies, only the creation of a new `AllocationAlgorithm` is needed. It is not necessary to re-develop the mechanical abilities present in the `OrderProcessor`, such as communication and scheduling. In software engineering, this technique is called "separating concerns."[7] It is mentioned here because of its large influence our system design.

A second type of object diagram is given in figure 3.5. This is a sequence diagram that provides a high-level illustration of how an order is placed and processed. This is slightly equivocating because it implies this process is a deterministic sequence. In actuality, the autonomous properties of the agents involved make it difficult to predict the exact timing of each activity. For the sake of simplicity, though, this collection of activities are described sequentially below:

(i) An `OrderGenerator` decides to place an order. It then populates this order (this activity is described in more detail below).

[6] In the object-oriented methodology, a class essentially is a template that defines the behavior and structure of a particular type. Instances of this class are called objects. For example, there may be two instances of the `OrderGenerator` class in our system: Customer1, and Customer2. These are unique, separate, and "living" entities. Their behavior and structure, however, are defined commonly by the contents of the `OrderGenerator` class. A useful analogy is to consider a class definition as a cookie-cutter; it is used to stamp many individual instances of a cookie that are all shaped from the same mold.

[7] In this example, one concern is the order promising logic. The other is the mechanic ability needed to execute this logic.

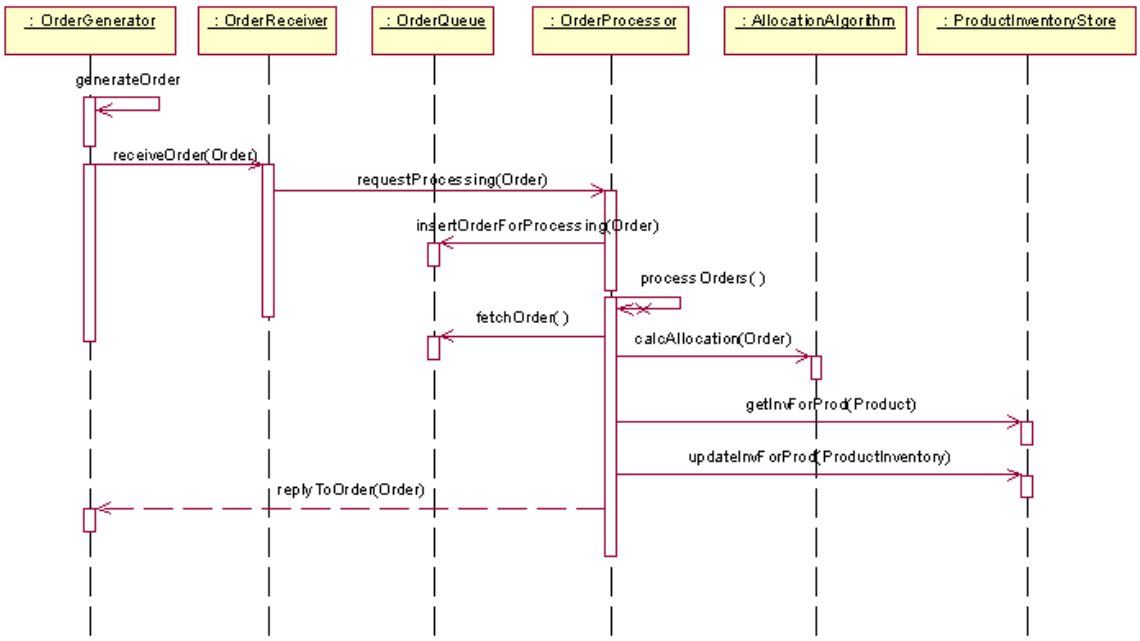

Figure 3.5: Order Fulfillment Submission Sequence

(ii) The `OrderGenerator` sends its order to an `OrderReceiver`.

(iii) The `OrderProcessor` decides to process one or more orders. It then asynchronously retrieves an order from its `OrderReceiver`.

(iv) The `OrderProcessor` relies on its `AllocationAlgorithm` to decide the extent to which to fill the order.

(v) The `OrderProcessor` decides to fill an order. It requests inventory from the `InventoryStore`.

(vi) The `OrderProcessor` delivers the order to the original `OrderGenerator`.

The way in which an `OrderGenerator` places its orders is central to our test-bed. As previously described, an `OrderGenerator` places orders autonomously and proactively. In order to simulate particular business scenarios, though, it is necessary to generate predictable aggregate demand that fits pre-determined statistical data. Typically, such aggregate demand is assumed to follow a standard random distribution, such as the normal or uniform distributions. Often the aggregate demand distribution varies for each demand class. Therefore, to simulate business

scenarios accurately, the test-bed must be able to re-create these per-demand class aggregate distributions.

To accomplish this, each instance of an `OrderGenerator` is assigned to one demand class. An aggregate demand distribution for each demand class is also defined. This serves as a guide for each `OrderGenerator` when it places orders. Additionally, the test-bed allows for an unequal amount of `OrderGenerator`s to be assigned to each demand class. As this imbalance is present in several real-life business scenarios, the test-bed must accommodate it to conduct some simulations accurately.

Specifically, there are four dimensions over which demand distributions are defined. These are defined for each demand class independently. Along with their abbreviations, they are given in the following list:

(vi) Number of Generators (Number Gens) - The total number of `OrderGenerator`s in the demand class

(vi) Orders per Step (Ords/Step) - The number of orders placed per day by each `OrderGenerator`

(vi) Order Elements per Order (OE/Ord) - The number of order elements in each order (i.e. the size of the order). [8]

(vi) Quantity per Product (Qty/P) - The quantity of a particular product to request (per order element).

(vi) Requested Due Date (DD) - The period of time within which an order element is requested to be delivered.

To define a random number distribution that describes one of these dimensions, four key parameters must be defined. Their definitions, and the abbreviations used to represent them are as follows:

(i) μ - The mean of the distribution

[8] See Section 3.1.3 on page 20 for a description of an order element.

(ii) Min - The minimum value of the distribution

(iii) Max - The maximum value of the distribution

(iv) σ - The standard deviation of the distribution

Using these parameters, the test-bed is able to re-create segmented demand that mimics historical patterns. This is a key element of its ability to model different business environments. Detailed examples of this are given in the the case studies in Chapter 4.

3.2.5 Agent Framework Selection

A final step in the design process was the selection of an object-oriented agent framework on top of which the test-bed is built. To accomplish this, a three-stage evaluation technique was used. In the first stage, about 10 dozen frameworks were evaluated briefly on the basis of the following criteria: 1) availability of support; 2) vitality of current development; 3) cost of use; and 4) flexibility of development. This initial screen pared down the options to about 20 candidates. These were then judged more carefully on the initial criteria again (in more detail), as well as on new criteria. Included in the new criteria were the following: applicability to our problem domain, ease of development, and product maturity. This process highlighted 3 front-running candidates. These were JADE 3.0b1 [26], MadKit 3.1b5 [37], and RePast 2.0.2 [42]. An extensive analysis was performed on these three frameworks. This included the development of sample applications with each. Based on this analysis, we eliminated the JADE framework primarily because we felt it was too heavyweight for our problem domain. We then chose RePast over MadKit. We did this because we concluded that RePast was the best fitting for our problem domain, learned that its developers actively and freely supported users, and discovered that it had been used in several successful projects by other researchers. Moreover, the code base of RePast is freely available, and well written. This was a key determinant in selecting RePast over the runner-up. [9]

[9] This proved to be an important factor. Not many, but a few times we altered the code base of RePast either to patch bugs or to customize it for our needs

3.3 Test-bed Data Flows

A high-level diagram of the inputs and outputs of the test-bed is given in figure 3.6.

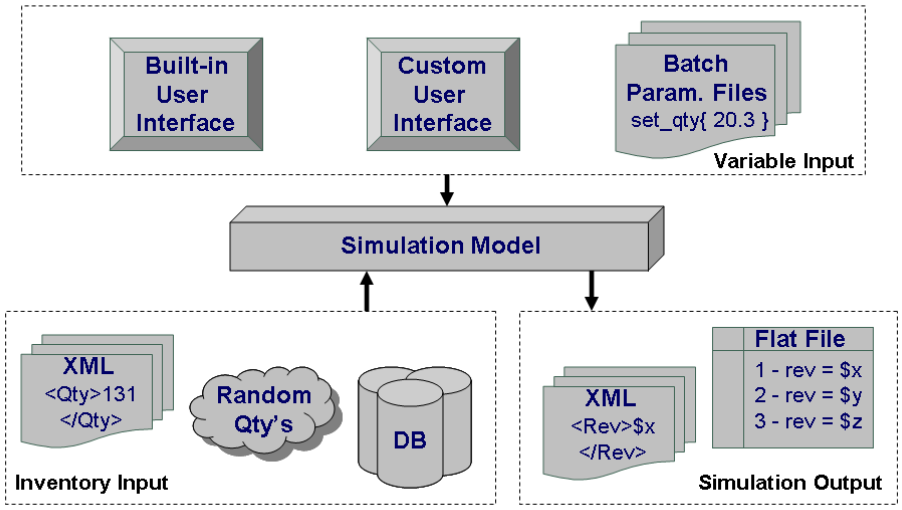

Figure 3.6: Input and Output Data Flow

There are several key elements of this diagram. For one, there are several ways in which parameter data may be input into the simulation. The Graphical User Interface (GUI) built-in to the RePast framework, for instance, may be use to input data. A screenshot of this GUI is given in figure 3.7.

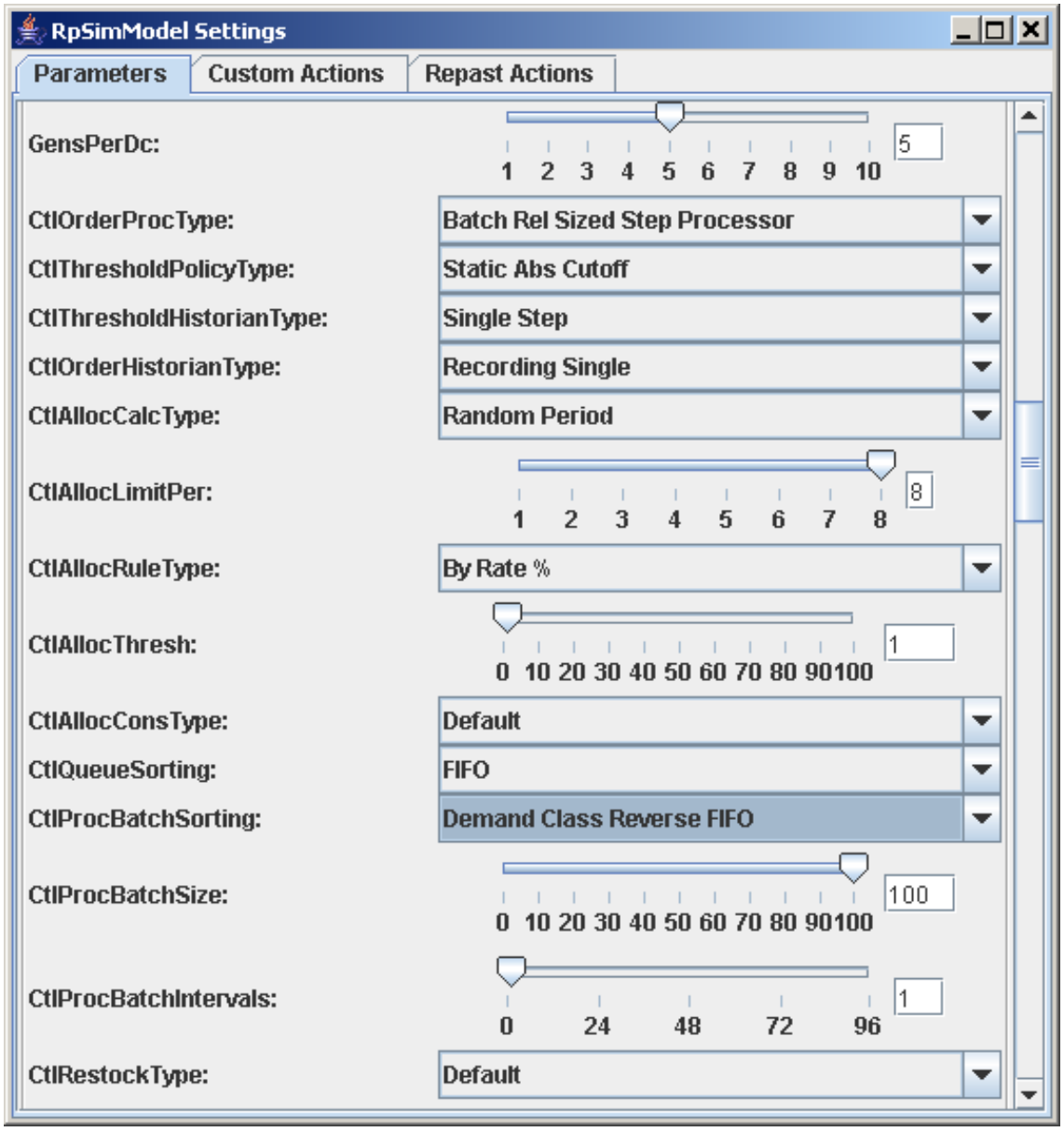

Figure 3.7: Built-In RePast GUI

This GUI is convenient to input parameter values manually. Using it to repeat a simulation on the order of 30 times, however, simply is uneconomical. Fortunately, the framework provides a batch mode, whereby input parameters are read from files. This greatly accelerates the process of running a simulations a statistically significant number of times. As a third alternative, a customized GUI may be used to input parameter values into the test-bed. This option is attractive to developer who wishes to mask parameters that are invariant, therefore allowing the user to concentrate only on the more important parameters.

The ability of the test-bed to use persistent data stores as data input sources is another key feature. An `InventoryStore` agent, for example, may read an XML file to determine what its initial inventory should be. Additionally, an `OrderGenerator` agent may read from an XML file in order to produce an exact sequence of orders. This is helpful in order to maintain a stringent control environment when testing a new revenue management policy.

A final noteworthy feature is the ability to output simulation results to different sources. For example, the test-bed is able to write simulation results to persistent data sources. It also offers options for data output, including XML files, delimited flat files, and arbitrary JAVA^{TM} classes that realize a common interface. Providing conveniently formatted output files is a critical capability of the test-bed, as it enables and accelerates the analysis of simulation results.

The test-bed is also able to chart data dynamically in while running a simulation. An example of this is shown in figure 3.8.

3.4 Implementation Methodology

An incremental life-cycle approach was used throughout the development of the test-bed. This was synchronized with periodic reviews of high-level requirements. This promoted phased releases of the test-bed so that it could be used and evaluated throughout the development process.

As described previously, an object-oriented development methodology was used to implement the test-bed. To exploit this to our advantage, several powerful tools were used. Among these were Rational Rose, for the creation of system diagrams; Concurrent Versions System (CVS),

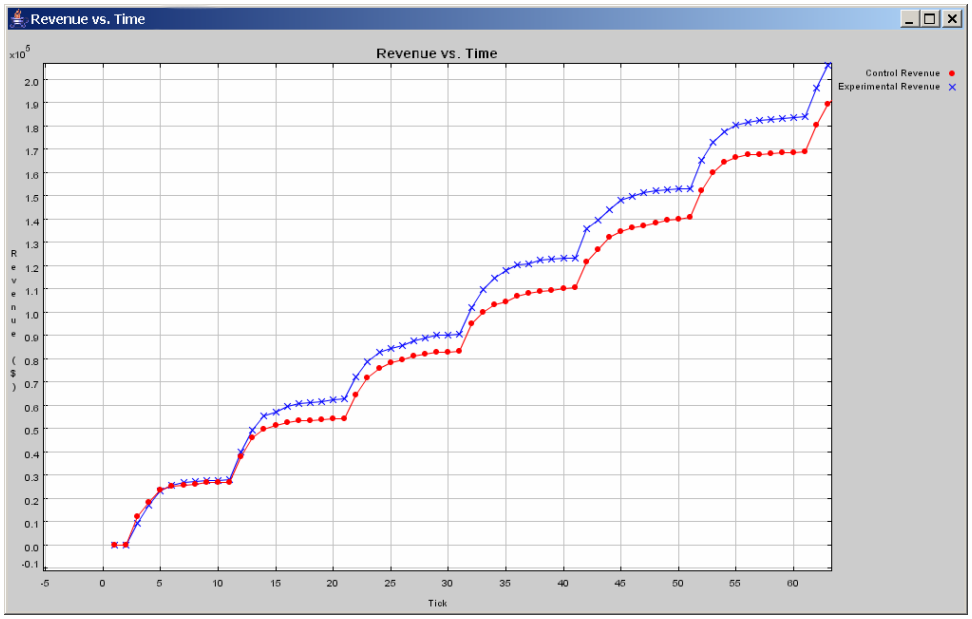

Figure 3.8: Graphical Output of Simulation Results

to manage the code base; IntelliJ, to edit, build, and test the code base; and YourKit, to profile the application.

Additionally, several techniques, in particular, were used considerably throughout development. One of these was unit testing. Unit tests were integrated into the build process, via the Apache ANT Project. [10] This provided a means of regression testing when new code was introduced into the system. Also, the code base was refactored often during development. This helped to maintain the large code base, and to eliminate redundant code. Finally, the application of several design patterns provided well-known, structured solutions that accelerated the development process.

The extensive use of the Factory and Dependency Injection Patterns warrants distinct attention.[11] Most parameters of the test-bed are given values at run-time. This dynamic binding precludes the use of hard-wired relationships among concrete classes within the system. By exploiting the Factory Pattern and the Dependency Injection Pattern, however, relationships defined at run-time become substantially simpler to manage. Specifically, hard-wiring concrete classes

[10] Freely available at http://ant.apache.org

[11] Both patterns are described in Martin's book [38].

together becomes largely unnecessary; only relationships among abstract classes and interfaces must be established. This effect enormously simplifies development. In our case, it enabled us to create the pluggable framework of our test-bed, which otherwise would have been fantastically challenging. As such, the use of these patterns was an indispensable aspect of our development process.

3.5 Validation and Verification

This final section describes how our test-bed was validated and verified. As a point of definition, *validation* refers to the process of ensuring that the design of a system fulfills accurately and completely the need for which it is being built. Validation commonly is described as "doing the right thing." *Verification* is defined differently: It refers to the process of ensuring that a system performs as is described in its requirements. Verification commonly is described as "doing the thing right."

The majority of our system validation was performed in the initial design phases. This consisted of group efforts to ensure that our initial models captured the essential elements of a generic business environment. We also continued validating the model during its implementation. As mentioned previously, we employed the iterative software development life-cycle. This particular life-cycle accepts new system requirements periodically. Correspondingly, at each of these instances, we revalidated the system model. Much of this effort involved prioritizing new features to add, and then ensuring that only the essential features were included in the revised model.

Verifying the test-bed throughout its development was a crucial practice in ensuring the accuracy of its simulation results. This constant verification was performed on many dimensions. First, unit tests were run following each new build of the system. The failure of a unit test prohibited code check-in until its cause was addressed. This helped us to verify that a number of small, but fundamental, units of functionality were not affected by a change to the code base.

Verification was also performed on third-party libraries that were used in the system. For example, we integrated the `Colt` random number generator library into the test-bed. Prior to

doing so, simple applications were created to test the accuracy of its random output. Similar procedures were performed for additional third-party libraries used, too.

As the system was constructed, system level verification was incorporated into the development process. Both white-box and black-box verification was included. White-box verification consisted primarily of live debugging and live profiling. This offered a pervasive view of the operations of the test-bed. Live debugging, in combination with live profiling, helped enormously in verifying system state requirements. This was especially beneficial as the non-deterministic properties of the agents involved precluded an analytic verification of system behavior; only by live, empirical investigation, were we able to analyze accurately system state.

Black-box verification offered a different confidence: It provided corroboration that the simulation produced correct results when given a certain set of inputs. Early on in the development life-cycle, simple, almost trivial tests were used to black-box verify the system. The expected outputs of these tests were calculated manually, and were then compared to the actual outputs of the system. This proved to be a useful practice during the construction of the test-bed. As the test-bed matured, we used more complicated black-box tests. An example of such a test is included as a case study in this thesis (see section 4.8). Like the simple tests, these more complete tests helped us to identify and amend errors in the test-bed, some of which were very subtle.

It is paramount to us that we offer a test-bed that is both flexible and accurate. Through the validation of our system model, we are assured that the design of our test-bed addresses the correct business problems. Also, the in-depth verification performed further assures us that the system produces correct output. As such, we are satisfied that we offer a complete product.

Chapter 4

Case Studies

This chapter presents the results of our six case studies that were performed using our test-bed. Four of the six case studies simulated a business environment that was based on the company, Dell, Inc. The other two case studies simulated an airline reservation system.

4.1 Data Collection Product Prices Used in the Case Studies

The products modeled in several of the case studies are derivations of a set of computers and electronics advertised by Dell, Inc. Specifically, 5 products were used. These are shown in table 4.1.

Type	**Name**	**Base Price**
1	Dell A920 Printer	$100
2	PL2010M 20-inch Monitor	$500
3	Dell Dimension 2400	$1,000
4	Dell Precision Workstation 370	$2,500
5	PPM50H3 50-inch Plasma	$5,000

Table 4.1: Products Used in Case Studies

Notice that only the base price is given for each product. This base price is not necessarily the final price for which the product is sold; the final selling price, in fact, is calculated based on the demand class of the customer. This is done by scaling the base price by an amount relative to each demand class. The test-bed provides of a number of different price scaling options, among which are the following: *none* (i.e. use the same price for each demand class), *simple linear* (i.e. the

price offered to demand class n will be half as much as the price offered to demand class $2n$), *simple formula-based* (an arbitrary function parameterized by product quantity, demand class, and base price), and *stepwise formula-based* (a demand class based step-wise function, also parameterized by product quantity and base price).

A *simple formula-based* method was used in the case studies that modeled the Dell product set. This formula was derived by mining data directly from the Dell website. Several shopping carts were created for each unique demand class. Each cart contained a same (or very similar) set of representative products. Linear regression was applied to the different product prices in order to establish a relationship between demand class and a product's nominal base price. The result of this analysis was the following formula:

$$FinalPrice = (BasePrice) * (1.0 + 0.1133 * (n - i)) \tag{4.1}$$

Where:

$$n = Number\ of\ Demand\ Classes$$

$$i = Demand\ Class\ of\ Customer$$

This formula establishes approximately an 11% incremental mark-up for each demand class (i.e. demand class i pays about an 11% premium over demand class $i + 1$). This is not an exact representation of the pricing of Dell's products, but rather a simplified estimation of final pricing derived through regression analysis. As such, it provides a rough, but consistent, illustration of overall revenue trends in the proceeding case studies.

The case studies that did not use this product set instead used a collection of airline tickets as inventory. The prices for these tickets (fares), were constant for each demand class. This structure fit exactly into the stepwise pricing formula; for each demand class, one constant value was output, which equaled the fare for that particular class.

4.2 Development of Shipping Rates and Costs Used in the Case Studies

To model deliverable products (e.g. a computer), shipping prices are added to a customer's final invoice. In a typical business environment, these prices vary in accordance to the shipping method (e.g. overnight, two-day, etc.). Identically to calculating final product prices, these shipping prices are calculated through parameterized formulas input into the test-bed. For the case studies that simulated the sale of deliverable products, a *simple formula-based* method was used. The parameters of this formula were derived by the same regression process that was used to derive the product price formulas. This effort produced a formula that calculates the final shipping price paid by a customer.

It is important to recognize that, in this test-bed, the customer does not necessarily chose the shipping method. Instead, the customer submits a requested due date with each order. Depending on the interpretation by each order promising policy, this requested due date may or may not represent a specific shipping method. For instance, a policy may use this input field necessarily as a representation of the shipping method to use; this is rather straightforward.

Alternatively, a policy may use this input field as the date by which a product should be delivered to the customer. In this scenario, the onus is placed on the manufacturer to select when and by which method to ship its product. For example, a customer may request that a product be delivered within 4 days. A manufacturer has several ways to accommodate this request: 1) ship the product immediately (end of day 1), using 3-day ground; 2) ship the product at the end of day 2, using 2-day express; or 3) ship the product at the end of day 3 using overnight shipping. There are obvious trade-offs to consider when selecting which course of action to take. First, the manufacturer must consider its current inventory: If, for example, the product is not available until day 3, then the first two options are infeasible. Second, the manufacturer must consider the shipping costs of accelerated delivery methods: Speedier delivery methods will cost more, but less expensive methods may not deliver the product on time. Third, the manufacturer must consider future demand: Although selecting slower delivery methods will be less expensive, it may be imprudent to promise the immediately available inventory (i.e. inventory ready to be

shipped at the end of the current day). This is because future orders may be placed by higher-paying customers for this same inventory. These trade-offs are those that are typically addressed in different order promising strategies.

In the latter example above, the customer pays a shipping price that is scaled for 4-day delivery. A rational manufacturer will prefer to select a shipping method that will cost an amount less than or equal to this shipping price. The manufacturer may realize a loss by selecting a shipping method that costs more than the shipping price paid by the customer. Likewise, the manufacturer may realize a profit by selecting a shipping method that costs less than the shipping price paid by the customer. In this latter case, however, a lateness penalty fee may offset any profit.

To establish this more realistic view of shipping expenses, shipping costs are integrated into the test-bed . Specifically, these costs represent the fees that the manufacturer pays to a shipping agency (e.g. UPS) to deliver its products to its customers. Like the shipping prices that are paid by a customer, the shipping costs incurred by the manufacturer vary according to the delivery mode; effectively, the manufacturer is now a customer of its shipping agency. The case studies that included shipping costs used a parameterized formula that was backed out from the customer's shipping price formula. This was done to maintain a relative balance between the two. That is, it was simple and reasonable to define first the margin between the price paid by the customer and the cost incurred by the manufacturer, and *then* to develop the appropriate cost formula to enforce this margin. Defining the price-cost margin, therefore, was the immediate prerequisite to defining the costing formula. As implied previously, this margin depends on the relationship between the due date requested by the customer, and the shipping method selected by the manufacturer. To establish concretely this margin for the case studies included herein, two simplifying assumptions were made.

(i) Near Negligible Margin Size in the Base Case - The manufacturer does not realize a substantial profit or loss if a shipping method is selected delivers the product in exactly $d-1$ days, where d is the requested delivery date of the customer. For example, if the customer

requests a product to be delivered in 4 days, then the shipping price-cost margin of 3-day shipping will be close to zero. This requires that the manufacturer ship the product at the end of the day on which the order was received, less it be delivered late. This assumption was the basis for calculating the shipping method costing formula.

(ii) Approximately Equal Marginal Increases in Customers' Shipping Prices and Manufacturer's Shipping Costs - The shipping price formula was derived from data mined from the Dell, Inc. website. This is a linear formula, the slope of which is dependent on the delivery date requested. A nearly identical slope was inserted to the shipping price formula. A slight adjustment was made to reflect the slightly higher marginal increase a customer pays to reduce delivery time by one day. This accounts for the additional risk a speedier delivery imparts on the manufacturer, such as the increased chance that the order may be delivered late. This difference in slopes acts to increase the margin of the base case as the requested delivery time decreases. That is, the margin for an overnight order that is filled on the ordering date is greater than the margin for an 8-day order that is filled on the ordering date. In none of the case studies presented in this thesis does this represent a significant portion of the manufacturer's profit, but nonetheless it provides an accurate view of the business scenario modeled.

The results of the analysis described above are the two following formulas:

$$ShippingCost = (BaseProductPrice) * (1.07 + 0.012 * (n - d)) \tag{4.2}$$

$$ShippingPrice = (BaseProductPrice) * (1.10 + 0.024 * (n - d)) \tag{4.3}$$

Where:

$$n = Number\ of\ Available\ Due\ Dates$$

$$d = Due\ Date\ Requested$$

The calculated margin for a product with a $1,000 base price and a maximum delivery duration of 8 days forward is given in table 4.2. The columns of the table represent the number of days forward, from the ordering date, within which a customer requests delivery. For example, "2 days forward" means a customer expects to receive a product no later than 2 days after the order was received. The rows of the table represent the speed of the shipping methods available to the manufacturer. For example, "1 day" represents an overnight delivery option. The base case, as described above, is represented along the main diagonal of the table (i.e. where $Days\ Requested = Days\ of\ Shipping$). Entries below the diagonal represent orders that are delivered late. Entries above the diagonal represent orders that ship later than they necessarily have to ship, provided sufficient inventory is available to fill the order immediately.

Days Forward Requested (Columns) vs. Days of Shipping (Rows)								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	$19.00	($5.00)	($29.00)	($53.00)	($77.00)	($101.00)	($125.00)	($149.00)
2	$41.00	$17.00	($7.00)	($31.00)	($55.00)	($79.00)	($103.00)	($127.00)
3	$63.00	$39.00	$15.00	($9.00)	($33.00)	($57.00)	($81.00)	($105.00)
4	$85.00	$61.00	$37.00	$13.00	($11.00)	($35.00)	($59.00)	($83.00)
5	$107.00	$83.00	$59.00	$35.00	$11.00	($13.00)	($37.00)	($61.00)
6	$129.00	$105.00	$81.00	$57.00	$33.00	$9.00	($15.00)	($39.00)
7	$151.00	$127.00	$103.00	$79.00	$55.00	$31.00	$7.00	($17.00)
8	$173.00	$149.00	$125.00	$101.00	$77.00	$53.00	$29.00	$5.00

Table 4.2: Shipping Pricing for $1,000 Product

This same relationship is displayed graphically in figure 4.1.

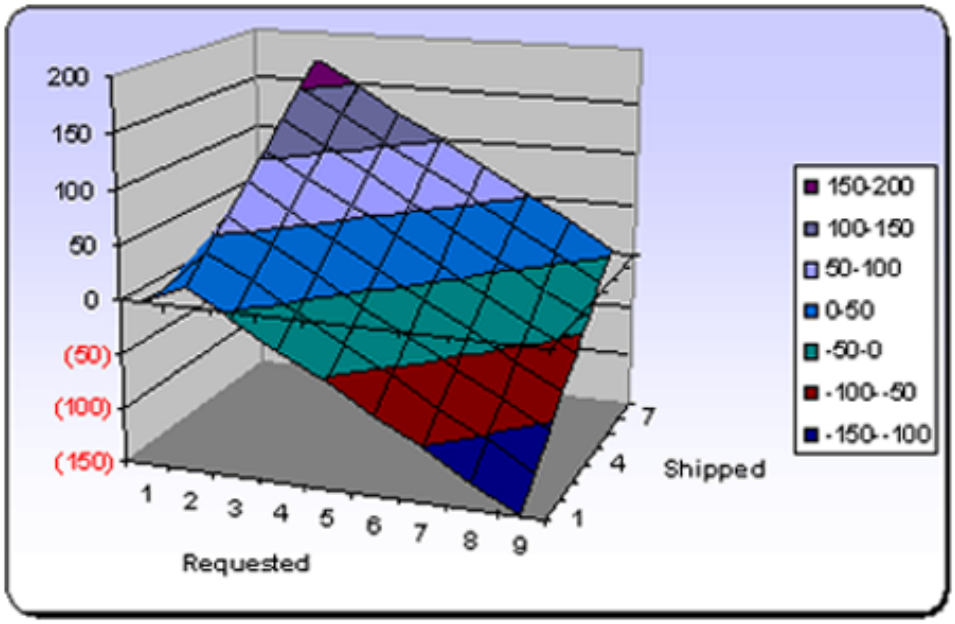

Figure 4.1: Shipping Price-Cost Margins

The case studies used a collection of airline tickets as inventory did not incorporate shipping costs. This represented more accurately the true business model.

4.3 Demand Class Distributions

Several of the case studies below involve testing a policy under different uncertainties of demand. To create these different environments, the distributions that control the order attributes of customers were altered. An example of how this may be done is given in figure 4.2. In this instance, each demand class uses a different normal distribution to generate values for a particular order attribute. This is useful when creating a demand environment in which different customer segments behave differently (this is described also in section 3.2.4, on page 28).

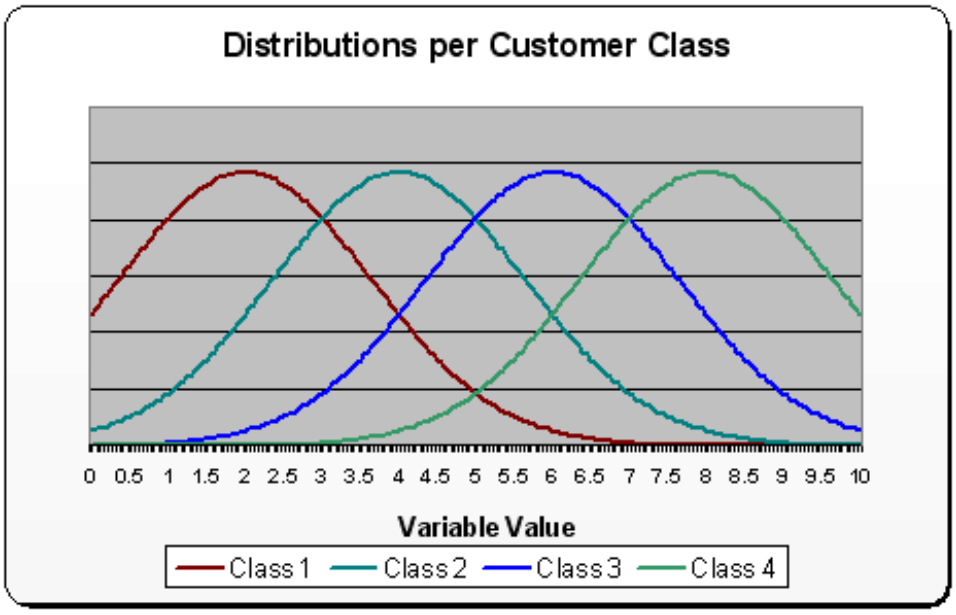

Figure 4.2: Example of Using a Different Distribution for Each Demand

In each case study, just one order attribute is altered when creating different demand environments. This helps to distinguish the effects of the attribute that is varied. Baseline values are given to the constant order attributes. The baseline distribution for the number of orders each customer places is constant; it is 1. The baseline distribution for the size of an order is has a mean of 5 and a standard deviation of 1. The baseline distribution for the requested quantity of a product has identical parameters (mean of 5 and standard deviation of 1). The baseline distribution for the requested due date of an order has a mean of 4 days, and a standard deviation of 1. These baseline distributions are illustrated in the following few figures.

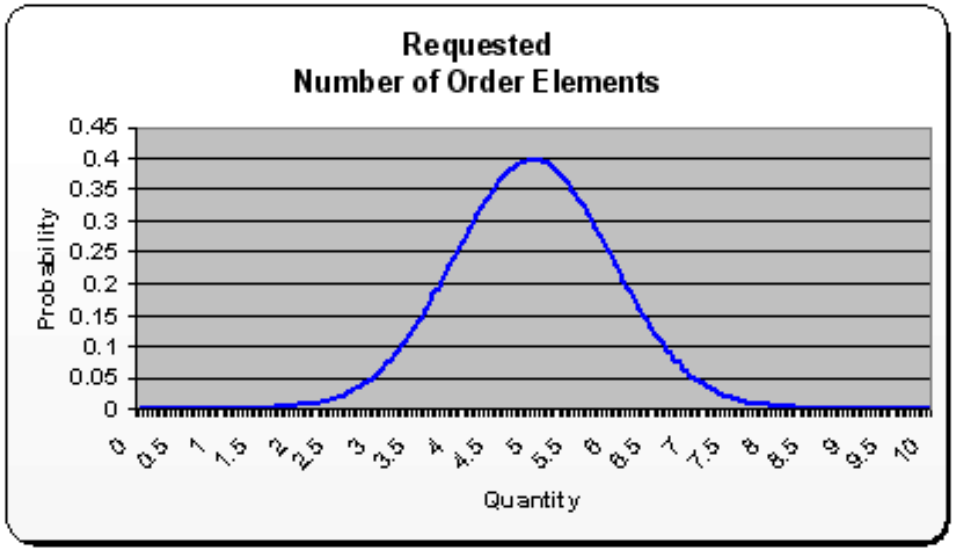

Figure 4.3: Baseline Distribution for the Size of Order

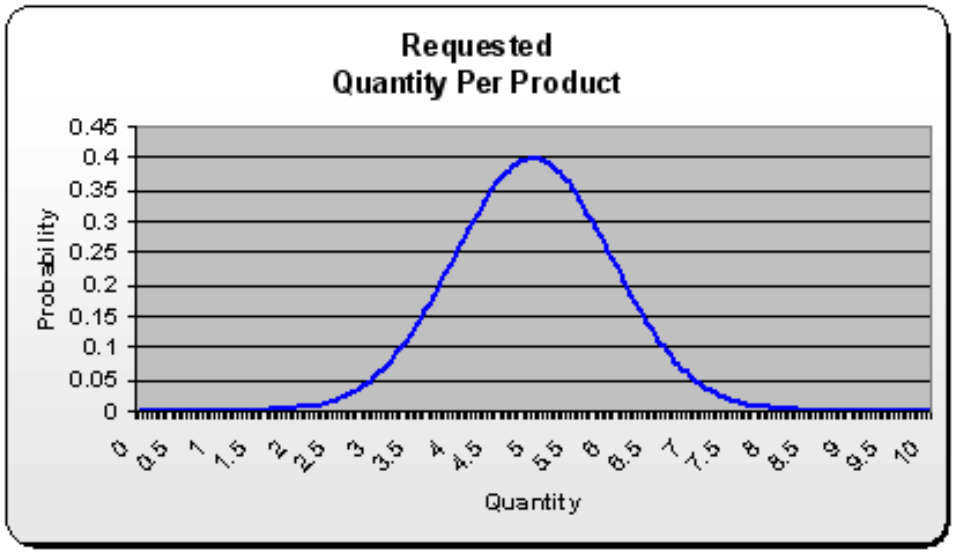

Figure 4.4: Baseline Distribution for the Requested Quantity of Product (SKU) in an Order

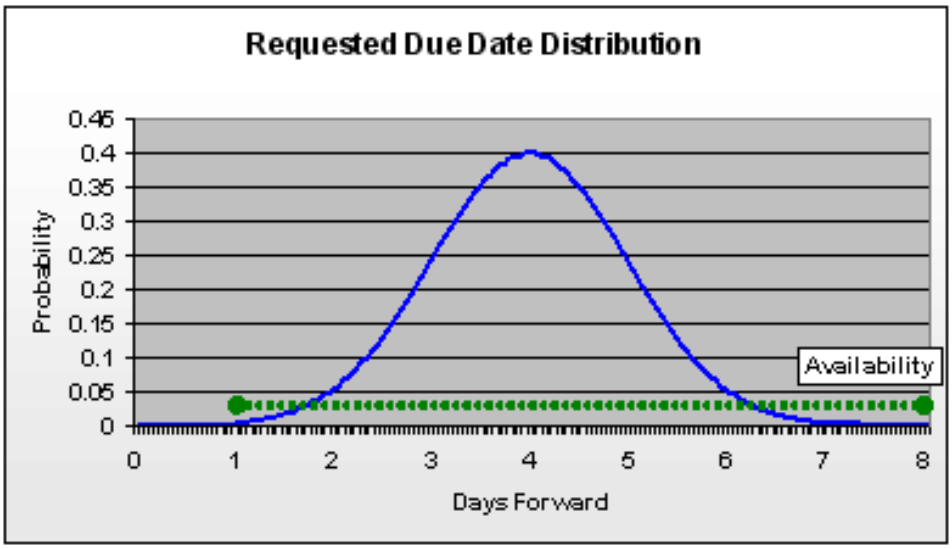

Figure 4.5: Baseline Distribution for the Requested Delivery Date in an Order

4.4 Experimental Methodology

Each of the experiments presented below was repeated exactly 30 times. This provided a statistical basis on which conclusions about significant effects were made. Results of each experiment were output into data files. These files then were processed by a PERL script. This script used the `Statistics::TTest` module to calculate the basic statistics of each case study, and then to perform comprehensive t-tests on these statistics. The accuracy of this module is corroborated in a 2004 review by Baiocchi [2]. The script output the calculated statistics into summarized data files, which were then loaded into Microsoft Excel^{TM}. Finally, charts and formatted data tables were created within Microsoft Excel^{TM}.

The first three case studies each simulated 100 days of placing and promising orders. For each day simulated, the daily revenue generated was recorded. Summing this daily revenue over all 100 days produced the aggregate statistic, *Total Revenue*. Average allocation rates for each demand class were also calculated. These were calculated by dividing the total quantity of products allocated to customers of a demand class by the total quantity of products requested by a customers of a demand class (see equation 4.4).

$$\textit{Allocation Rate} = (\textit{Quantity Promised}) \ / \ (\textit{Quantity Requested}) \tag{4.4}$$

This statistic was calculated for each demand class to illustrate how some policies give preference to higher paying demand classes. In the following case studies, the demand classes are ordered by highest paying (demand class 1) to lowest paying (demand class n, where n is the number of demand classes).

When these statistics are presented in tabular form, the following headings are used:

(i) Total Revenue - This column gives the total revenue generated by a policy.

(ii) Class i % - This column gives the allocation rate for demand class i.

For some case studies, the total allocation rate is given. This is simply the allocation rate of all customers in all demand classes. In other words, no distinction is made between customers in

different demand classes. The reason this is done is to capture aggregate behavior over all demand classes, which is an important metric for some of the policies that were evaluated.

As mentioned above, t-tests were performed on the statistics from each case study. The results of several of these t-tests are given in matrix form within the following sections. The interpretation of these matrices is straightforward: The policies that are tested are given the first row and the first column of the matrix. At each intersecting point within the matrix, a "0" (zero) or "1" (one) is given. A "0" in square (n, m) indicates that there is no significant difference between policy n and policy m. A "1", on the other hand, indicates that there is a significant difference between policy n and m.

4.5 Batching Case Studies

As introduced section 2.1, a key decision in an order promising policy is determining how frequently to process orders. Orders may be processed continuously, responding to each order immediately after it arrives. Orders also may be collected over time, sorted by priority, and processed periodically as a batch. The case studies in this section examine the impact of varying this processing frequency. This was simulated by decreasing the batching size from 100% of the total orders received in a day, to 0%, which represents the real-time (order-by-order) continuous processing method. Intervals of 10% are used to create performance curves that map the incremental effects of adjusting the batch size.

This experiment was performed in three different demand environments. In each environment four demand classes are used. In the first environment, labeled "High to Low Demand," customers in the two higher priority demand classes place more orders than the lower priority demand classes. In the second environment, labeled "Even Demand," the mean quantity of orders placed by customers in all demand classes is equal. Finally, in the third environment, labeled, "Low to High Demand," customers in the two lower priority demand classes place more orders than the higher priority demand classes. These three environments were tested individually to examine the how the impact of the batching policies vary in different demand environments.

4.5.1 High to Low Demand

In this scenario, the two higher priority demand classes place more orders than the lower priority demand classes. The specific distributions that were used are given in table 4.3 (definitions of the abbreviations used are given in section 3.2.4 on page 29).

Demand Class	**1**	**2**	**3**	**4**
Scenario High to Low				
Number Gens	9	7	3	1
μ Ords/step	1	1	1	1
Min Ords / Step	1	1	1	1
Max Ords/Step	1	1	1	1
σ Ords/Step	0	0	0	0
μ OE/ Ord	5	5	5	5
Min OE	1	1	1	1
Max OE	9	9	9	9
σ OE	1	1	1	1
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	4	4	4	4
Min DD	1	1	1	1
Max DD	8	8	8	8
σ DD	1	1	1	1

Table 4.3: High to Low Demand

The numeric results of this experiment are given in table 4.4. Note that the allocation rates of the four demand classes are nearly equal for the real-time policy. As the batching size is increased, however, the allocation rate of the highest priority demand class increases, and the allocation rates of the other demand classes decrease.

	Total Revenue	**Class 1 %**	**Class 2 %**	**Class 3 %**	**Class 4 %**
0	$152,836,187.70	0.712	0.707	0.711	0.721
5	$152,777,900.36	0.710	0.710	0.715	0.714
10	$157,395,391.76	0.708	0.712	0.701	0.708
20	$157,685,065.33	0.722	0.700	0.699	0.687
30	$141,708,458.31	0.731	0.698	0.679	0.683
40	$141,741,072.38	0.736	0.695	0.677	0.666
50	$151,253,732.62	0.750	0.687	0.648	0.644
60	$151,352,309.59	0.749	0.692	0.648	0.639
70	$154,004,095.81	0.754	0.690	0.644	0.622
80	$154,502,932.97	0.762	0.690	0.641	0.609
90	$151,047,177.70	0.775	0.682	0.608	0.577
100	$151,852,359.91	0.795	0.671	0.585	0.549

Table 4.4: Simulation Results of Policies

The experimental results are illustrated graphically in figure 4.6. This figure illustrates how the allocation percentage of the higher priority demand classes increases as the batching size increases. This an expected outcome of the batching policy.

It is also expected that the total revenue will increase as the batching size is increased. This, however, was not observed in this experiment. Figure 4.6 suggests that the total revenue for each policy varies greatly between policies. This is misleading, though, as there were no significant differences measured between any two policies. That is, t-testing each policy against all other policies revealed no significant differences in total revenue. As a consequence, the total revenue calculated for each batching policy is considered to be statistically equivalent.

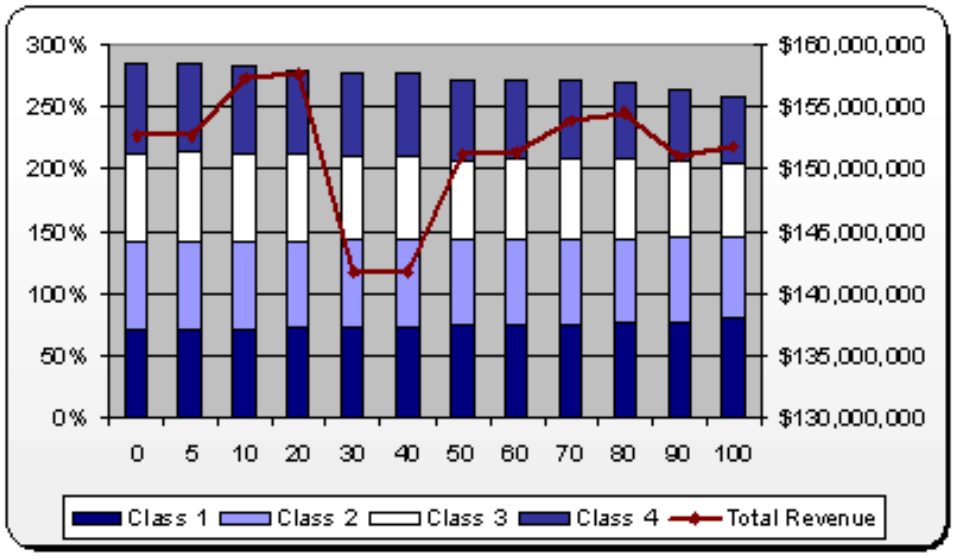

Figure 4.6: Allocation % (left axis) and Total Revenue (right axis) of High to Low Demand

The reason why there are no significant differences in total revenue stems from the pricing formula used in this case study. This formula did not mark up product prices for the highest priority demand class far enough over the other demand classes. As a result, allocating a greater percentage of orders from this demand class did not yield enough additional revenue to increase significantly the total revenue generated. A pricing formula that promotes the prices paid by the highest priority demand class more substantially, however, should produce a more observable increase in total revenue as the batching size is increased. This experiment illustrates the need to apply a proper pricing model to revenue management strategy. As total revenue generated typically is the most important metric of a business strategy, it is not enough simply to concentrate on order promising improvements (i.e. allocation rates). Pricing strategies must be configured to exploit such improvements, and consequently to increase total revenue.

The t-test results of the allocation percentage of the highest priority demand class are given in table 4.5. This table illustrates that adjusting the batching percentage by 10% almost always signficantly increases the allocation percentage of the highest priority demand class. The t-test results for total revenue are not given, as there are no significant differences between policies for that statistic.

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	1	1	1	1	1	1	1	1	1
5	0	0	0	1	1	1	1	1	1	1	1	1
10	0	0	0	1	1	1	1	1	1	1	1	1
20	1	1	1	0	1	1	1	1	1	1	1	1
30	1	1	1	1	0	0	1	1	1	1	1	1
40	1	1	1	1	0	0	1	1	1	1	1	1
50	1	1	1	1	1	1	0	0	0	1	1	1
60	1	1	1	1	1	1	0	0	0	1	1	1
70	1	1	1	1	1	1	0	0	0	0	1	1
80	1	1	1	1	1	1	1	1	0	0	1	1
90	1	1	1	1	1	1	1	1	1	1	0	1
100	1	1	1	1	1	1	1	1	1	1	1	0

Table 4.5: T-Test For Highest Priority Demand Class

4.5.2 Even Demand

In this scenario, the mean quantity of orders placed by all demand classes is equal. The specific distributions that were used are given in table 4.6.

Demand Class	**1**	**2**	**3**	**4**
Scenario Even				
Number Gens	5	5	5	5
μ Ords/step	1	1	1	1
Min Ords / Step	1	1	1	1
Max Ords/Step	1	1	1	1
σ Ords/Step	0	0	0	0
μ OE/ Ord	5	5	5	5
Min OE	1	1	1	1
Max OE	9	9	9	9
σ OE	1	1	1	1
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	4	4	4	4
Min DD	1	1	1	1
Max DD	8	8	8	8
σ DD	1	1	1	1

Table 4.6: Even Demand

The numeric results of this experiment are given in table 4.7. Note again how the allocation rate of demand class 1 increases as the batching size increases.

	Total Revenue	**Class 1 %**	**Class 2 %**	**Class 3 %**	**Class 4 %**
0	$134,626,626.51	0.700	0.699	0.694	0.706
5	$134,383,305.34	0.697	0.700	0.697	0.699
10	$143,199,599.03	0.706	0.700	0.703	0.691
20	$142,720,879.27	0.718	0.704	0.696	0.678
30	$139,539,657.85	0.727	0.707	0.692	0.664
40	$140,045,924.39	0.738	0.718	0.687	0.659
50	$140,227,463.42	0.762	0.726	0.684	0.637
60	$140,037,845.11	0.762	0.728	0.678	0.636
70	$144,982,586.85	0.760	0.728	0.687	0.630
80	$144,778,074.32	0.777	0.737	0.677	0.614
90	$148,428,854.73	0.790	0.740	0.665	0.589
100	$148,635,939.47	0.816	0.746	0.668	0.564

Table 4.7: Simulation Results of Policies

The experimental results are illustrated graphically in figure 4.7. This figure illustrates how the allocation percentage of the higher priority demand classes increases as the batching size increases. This is an expected outcome of the batching policy.

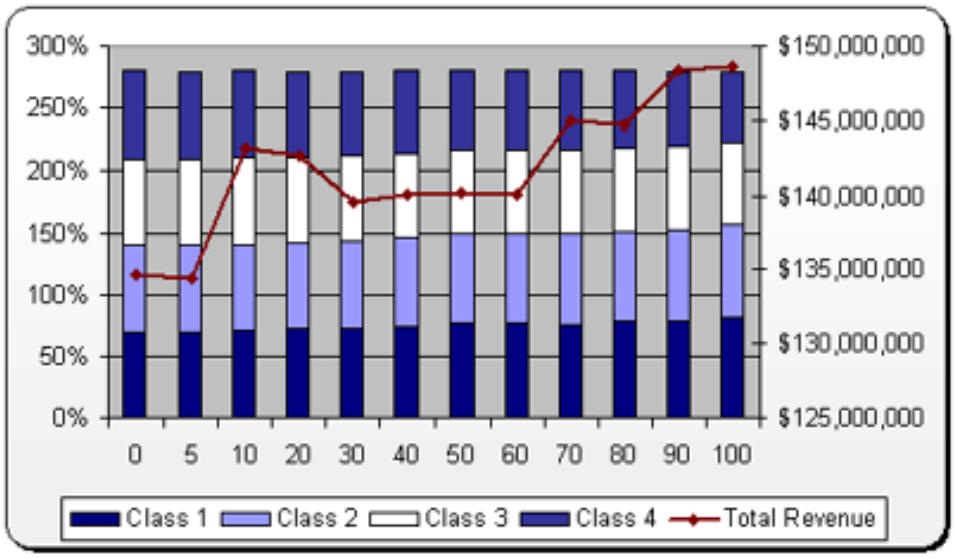

Figure 4.7: Allocation % (left axis) and Total Revenue (right axis) of Even Demand

For the same reasons given in High-to-Low experiment, the total revenue does not vary significantly between policies. There are some exceptions, though. There is a significant difference between the total revenue generated by the 100% policy and the real-time (0%) policy. There is also a significant difference between the total revenue generated by the 90% policy and the real-time (0%) policy. This is expected: A large batching size should allow a manufacturer to promise more orders from higher-paying customers, which in turn should lead to greater revenue. The extent to which this occurs, however, is determined largely by the pricing strategy employed. The pricing strategy used in this experiment did not exploit the increases in allocation rate of the highest paying customers well enough to produce significant revenue differences between each policy. This is the reason why there are no significant differences in total revenue other than the two mentioned above. The t-test results for total revenue are given in table 4.8.

	0	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	0	0	0	0	0	0	1	1
10	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0	0	0
70	0	0	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0
90	1	0	0	0	0	0	0	0	0	0	0
100	1	0	0	0	0	0	0	0	0	0	0

Table 4.8: T-Test For Total Revenue

The t-test results of the allocation percentage of the highest priority demand class are given in table 4.9. The table is similar to the result table of the High-to-Low scenario (see table 4.5). As in the High-to-Low case, adjusting the batching percentage by 10% almost always signficantly increases the allocation percentage of the highest priority demand class. This illustrates a key benefit of using a larger batching size.

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	1	1	1	1	1	1	1	1	1
5	0	0	0	1	1	1	1	1	1	1	1	1
10	0	0	0	1	1	1	1	1	1	1	1	1
20	1	1	1	0	0	1	1	1	1	1	1	1
30	1	1	1	0	0	0	1	1	1	1	1	1
40	1	1	1	1	0	0	1	1	1	1	1	1
50	1	1	1	1	1	1	0	0	0	1	1	1
60	1	1	1	1	1	1	0	0	0	1	1	1
70	1	1	1	1	1	1	0	0	0	1	1	1
80	1	1	1	1	1	1	1	1	1	0	1	1
90	1	1	1	1	1	1	1	1	1	1	0	1
100	1	1	1	1	1	1	1	1	1	1	1	0

Table 4.9: T-Test For Highest Priority Demand Class

4.5.3 Low to High Demand

In this scenario, the two higher priority demand classes place fewer orders than the lower priority demand classes. The specific distributions that were used are given in table 4.10.

Demand Class	**1**	**2**	**3**	**4**
Scenario Low to High				
Number Gens	1	3	7	9
μ Ords/step	1	1	1	1
Min Ords / Step	1	1	1	1
Max Ords/Step	1	1	1	1
σ Ords/Step	0	0	0	0
μ OE/ Ord	5	5	5	5
Min OE	1	1	1	1
Max OE	9	9	9	9
σ OE	1	1	1	1
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	4	4	4	4
Min DD	1	1	1	1
Max DD	8	8	8	8
σ DD	1	1	1	1

Table 4.10: Low to High Demand

The numeric results of this experiment are given in table 4.11.

	Total Revenue	**Class 1 %**	**Class 2 %**	**Class 3 %**	**Class 4 %**
0	$142,185,811.79	0.697	0.712	0.709	0.709
5	$142,360,943.35	0.708	0.710	0.712	0.707
10	$143,956,856.44	0.723	0.721	0.713	0.703
20	$143,126,910.39	0.747	0.722	0.716	0.694
30	$143,275,721.45	0.721	0.740	0.718	0.692
40	$143,408,059.19	0.755	0.750	0.725	0.682
50	$147,445,111.28	0.773	0.761	0.730	0.670
60	$147,830,141.86	0.778	0.768	0.734	0.668
70	$141,505,825.99	0.788	0.770	0.734	0.668
80	$141,233,416.33	0.790	0.780	0.739	0.656
90	$144,066,015.07	0.804	0.802	0.748	0.641
100	$144,847,089.46	0.844	0.813	0.754	0.624

Table 4.11: Simulation Results of Policies

The experimental results are illustrated graphically in figure 4.8. This figure illustrates how the allocation percentage of the higher priority demand classes increases as the batching size increases. This is the expected outcome of the batching policy. The total revenue, identically to the first case, does not differ with statistical significance at all.

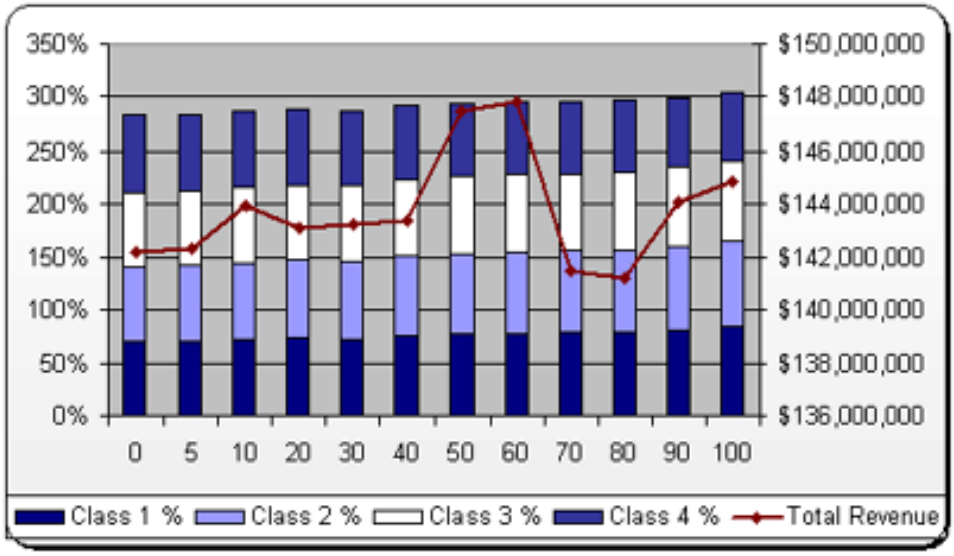

Figure 4.8: Allocation % (left axis) and Total Revenue (right axis) of High to Low Demand

The t-test results of the allocation percentage of the highest priority demand class are given in table 4.12. The table is similar to the result tables of the other two cases.

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	1	1	1	1	1	1	1	1	1	1
5	0	0	0	1	0	1	1	1	1	1	1	1
10	1	0	0	1	0	1	1	1	1	1	1	1
20	1	1	1	0	1	0	1	1	1	1	1	1
30	1	0	0	1	0	1	1	1	1	1	1	1
40	1	1	1	0	1	0	0	1	1	1	1	1
50	1	1	1	1	1	0	0	0	0	0	1	1
60	1	1	1	1	1	1	0	0	0	0	1	1
70	1	1	1	1	1	1	0	0	0	0	0	1
80	1	1	1	1	1	1	0	0	0	0	0	1
90	1	1	1	1	1	1	1	1	0	0	0	1
100	1	1	1	1	1	1	1	1	1	1	1	0

Table 4.12: T-Test For Highest Priority Demand Class

4.5.4 Remarks

In all three demand environments, increasing the batching percentage produced significant increases in the allocation rate of the highest priority demand class. This is sensible - the more orders that can be collected and sorted before being processed, the greater the probability that the manufacturer will fill higher priority orders before exhausting its inventory. This improvement in allocation rate was most significant in the "Low-to-High" demand environment, second most significant in the "Even Demand" environment, and least significant in the "High-to-Low environment." This is attributed to proportion of total demand that is comprised by orders from higher paying customers: As this proportion decreases, (i.e. the number of orders from higher paying customers lessens), it becomes increasingly valuable to prioritize these orders to ensure they are filled.

These results suggest that a manufacturer should consider the demand environment of its market when determining the batching size to be used. Specifically, doing so will help the manufacturer determine the worth of increasing the batching size. Once this is determined, the manufacturer may balance more knowledgeably the trade-off between responsiveness to all customers and preference to high priority customers. Briefly, this trade-off is as follows: Increasing the batching size will increase the average allocation percentage of higher paying customers, but will also increase the time needed to respond to customers' orders; decreasing the batching size will do the opposite, and decrease both of these measures. Understanding the factors involved in this trade-off will aid a manufacturer make an informed decision, hence the worth of such batching experiments.

Finally, the effects measured on total revenue did not always match the expected results. This is attributable to the pricing model that was used throughout this case study. The pricing structure used did not effectively promote the premium paid by the highest paying demand class. As a result, even though the allocation rate of the highest paying demand class improved consistently and significantly by increasing the batching size, the total revenue generated by different policies did not improve significantly (minus two exceptions in the "Even Demand" case). Applying a pricing model that amplifies this effect is a critical factor to consider when employing

a revenue management strategy. Without such a pricing model in place, improvements in order promising techniques simply may not be reflected in the bottom line.

4.6 Order Splitting Comparison Case Studies

A second key decision in an order promising policy is determining whether to accept orders that may can not be filled completely. Accepting such orders is described as "splitting." This refers to the way in which an order may be split up into two, half of which is filled and half of which is not. This is in contrast to accepting orders only if all components of the order can be filled 100%. There are two basic types of splitting, horizontal splitting and vertical splitting. It may help to refer to the illustration of an order in figure 4.9 on page 64 to understand the difference.

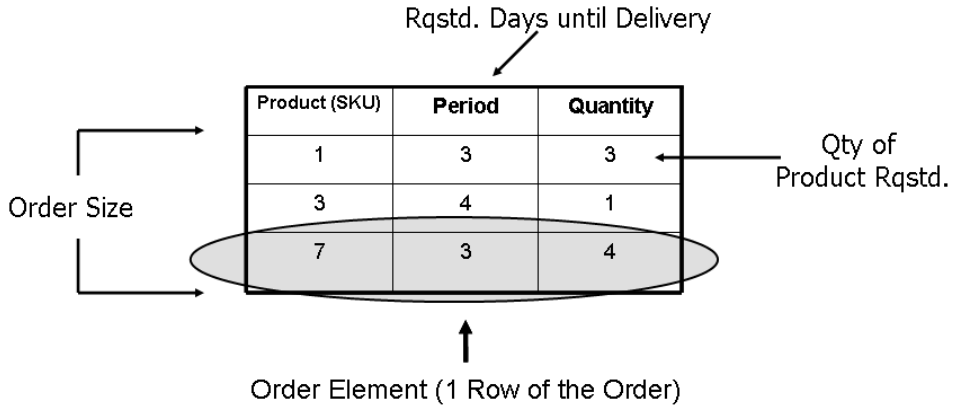

Figure 4.9: Structure of an Order

Horizontal splitting considers each Order Element as an atomic component; either it is filled completely, or it is not filled at all. For example, consider Product 7 in figure 4.9. The customer is requesting a quantity of 4 to be delivered within 3 days. If the manufacturer has just 3 to promise, then this Order Element will not be filled at all.

Vertical splitting allows Order Elements to be filled partially. In the exampled given above, the manufacturer would be allowed to promise a quantity of 3, despite the customer's request of 4. The case studies in this section consider only vertical splitting. Each policy uses an allocation rate threshold to determine if an order may be promised. That is, if the total rate of an order is

greater than the allocation threshold, then the order may be promised; otherwise, it may not be promised. The calculation of the allocation rate is straightforward. It is simply the sum of the quantities that are available to be promised (over all elements in the order) divided by the sum of the requested quantities. Returning to the example above, assume that the manufacturer is able to fill the first two order elements completely. Therefore, the allocation rate of this order is calculated to be:

$$(3 + 1 + 3)/(3 + 1 + 4) = 7/8 = 87.5\%$$

If the allocation rate threshold of the order promising policy used by the manufacturer is below 87.5%, this order may not be promised. Otherwise, the manufacturer may promise the order.

Intuitively, it may seem that implementing a policy that allows partially fillable orders to be promised will increase a manufacturer's revenue. This is because the manufacturer may now accept some orders that it can not fill entirely, as opposed to denying these orders completely. As a trade-off to this increased promising flexibility, the customers of the manufacturer may become unhappy that a greater proportion of their orders are not filled completely. This tends to occur as the manufacturer exhausts its inventory more rapidly by promising it to orders that it can fill only partially. Effectively, the overall order allocation rate decreases as as the allocation threshold is decreased. This phenomenon becomes increasingly significant as the average customer order size increases, but it nonetheless is observable for smaller sized orders, too.

The case studies in this section examine the impact of varying the allocation rate threshold. This was performed by decreasing the threshold from 100% (all or nothing), to near 0% (any portion of the order). Intervals of 10% are used to create performance curves to illustrate the incremental effects of decreasing the allocation rate threshold.

This experiment was performed in three different demand environments. In each environment four demand classes are used. In the first environment, labeled "Many Small Orders," customers in all demand classes place large quantities of relatively small orders. In the second environment, labeled "Even Sized Orders," customers in all demand classes place moderate amounts of moderately sized orders. Finally, in the third environment, labeled, "Few Large Orders," cus-

tomers in all demand classes place relatively few large orders. These three environments were tested individually to examine the how the impact of the splitting policies vary in different demand environments.

4.6.1 Many Small Orders

In this scenario, many relatively small-sized orders are placed by all demand classes. The specific distributions that were used are given in table 4.13.

Demand Class	**1**	**2**	**3**	**4**
Scenario Many Small Orders				
Number Gens	7	7	7	7
μ Ords/step	5	5	5	5
Min Ords / Step	1	1	1	1
Max Ords/Step	9	9	9	9
σ Ords/Step	1	1	1	1
μ OE/ Ord	2	2	2	2
Min OE	1	1	1	1
Max OE	3	3	3	3
σ OE	0.333	0.333	0.333	0.333
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	4	4	4	4
Min DD	1	1	1	1
Max DD	8	8	8	8
σ DD	1	1	1	1

Table 4.13: Many Small Orders

The numeric results of this experiment are given in table 4.14.

	Total Revenue	**Class 1 %**	**Class 2 %**	**Class 3 %**	**Class 4 %**
0	$266,981,410.67	0.487	0.486	0.490	0.490
5	$266,623,303.22	0.489	0.489	0.489	0.487
10	$281,649,351.71	0.491	0.489	0.489	0.493
20	$281,380,058.18	0.490	0.489	0.490	0.490
30	$276,307,908.19	0.489	0.488	0.490	0.488
40	$276,316,303.31	0.487	0.488	0.490	0.488
50	$266,862,010.67	0.487	0.486	0.486	0.486
60	$265,481,315.35	0.483	0.484	0.480	0.484
70	$280,859,698.80	0.484	0.482	0.481	0.485
80	$279,846,047.60	0.479	0.483	0.482	0.483
90	$262,659,749.24	0.477	0.480	0.479	0.478
100	$262,641,534.41	0.477	0.477	0.479	0.480

Table 4.14: Simulation Results of Policies

The experimental results are illustrated graphically in figure 4.10. It is not entirely obvious, but there is a general increase in allocation percentage for each demand class as the splitting threshold is lowered. This is presented in table 4.15, which shows the t-test results of the allocation percentage of the highest priority demand class (used as a representative of all other demand classes, all of which exhibited near identical relationships). Note that decreasing the threshold below 50% provides no significant increase in allocation percentage. This is an interesting effect that was not predicted prior to running the simulation.

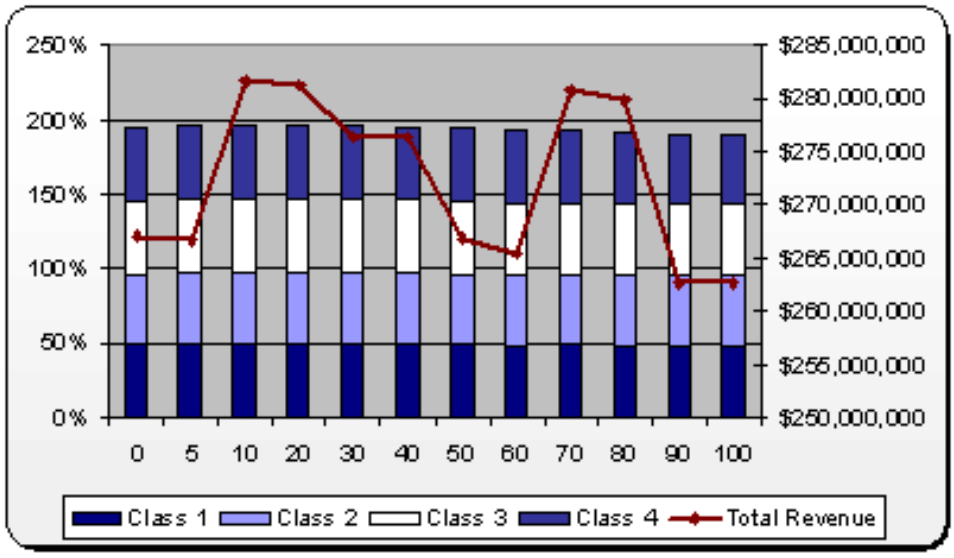

Figure 4.10: Allocation % (left axis) and Total Revenue (right axis) of Many Small Orders

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	0	0	0	0	1	0	1	1	1
5	0	0	0	0	0	0	0	1	0	1	1	1
10	0	0	0	0	0	0	0	1	1	1	1	1
20	0	0	0	0	0	0	0	1	1	1	1	1
30	0	0	0	0	0	0	0	1	1	1	1	1
40	0	0	0	0	0	0	0	1	0	1	1	1
50	0	0	0	0	0	0	0	0	0	1	1	1
60	1	1	1	1	1	1	0	0	0	0	1	1
70	0	0	1	1	1	0	0	0	0	0	1	1
80	1	1	1	1	1	1	1	0	0	0	0	0
90	1	1	1	1	1	1	1	1	1	0	0	0
100	1	1	1	1	1	1	1	1	1	0	0	0

Table 4.15: T-Test For Highest Priority Demand Class

The total revenue in figure 4.6 varies from policy to policy, but this difference is never significant. This is illustrated in table 4.16.

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0	0	0	0
70	0	0	0	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0	0
90	0	0	0	0	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.16: T-Test For Total Revenue

4.6.2 Even Sized Orders

In this scenario, the number of orders placed is balanced with the size of each order. The specific distributions that were used are given in table 4.17.

Demand Class	**1**	**2**	**3**	**4**
Scenario Even				
Number Gens	3	3	3	3
μ Ords/step	5	5	5	5
Min Ords / Step	1	1	1	1
Max Ords/Step	9	9	9	9
σ Ords/Step	1	1	1	1
μ OE/ Ord	5	5	5	5
Min OE	4	4	4	4
Max OE	6	6	6	6
σ OE	0.3	0.3	0.3	0.3
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	4	4	4	4
Min DD	1	1	1	1
Max DD	8	8	8	8
σ DD	1	1	1	1

Table 4.17: Even Sized Orders

The numeric results of this experiment are given in table 4.18.

	Total Revenue	**Class 1 %**	**Class 2 %**	**Class 3 %**	**Class 4 %**
0	$312,914,695.21	0.495	0.498	0.497	0.495
5	$313,162,811.75	0.496	0.495	0.496	0.495
10	$310,185,617.51	0.499	0.496	0.497	0.492
20	$310,139,332.21	0.494	0.497	0.497	0.497
30	$302,630,874.45	0.495	0.492	0.493	0.496
40	$302,178,927.71	0.492	0.493	0.494	0.494
50	$309,897,260.20	0.490	0.490	0.492	0.491
60	$309,137,385.54	0.489	0.491	0.487	0.492
70	$316,299,098.61	0.486	0.486	0.484	0.489
80	$311,745,671.62	0.482	0.480	0.477	0.481
90	$292,989,323.50	0.469	0.464	0.463	0.470
100	$290,801,512.33	0.463	0.465	0.464	0.461

Table 4.18: Simulation Results of Policies

The experimental results are illustrated graphically in figure 4.11. Like the previous case, there is a general increase in allocation percentage for each demand class as the splitting threshold is lowered. This is illustrated in table 4.19, which shows the t-test results of the allocation percentage of the highest priority demand class (used as a representative of all other demand classes, all of which exhibited near identical relationships). The effects of decreasing the splitting threshold are a little more pronounced in this case than in the "Many Small Orders" case.

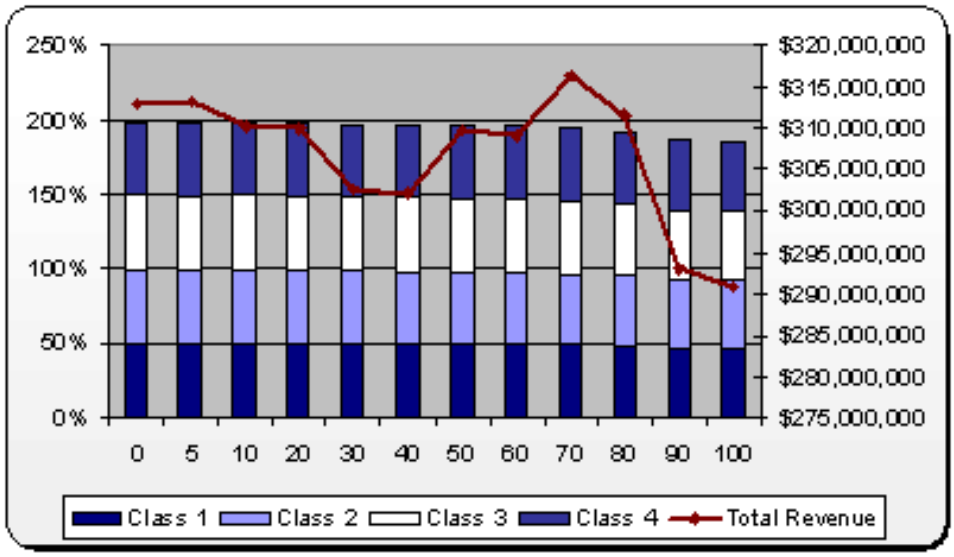

Figure 4.11: Allocation % (left axis) and Total Revenue (right axis) of Even Sized Orders

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	0	0	0	0	0	1	1	1	1
5	0	0	0	0	0	0	0	1	1	1	1	1
10	0	0	0	0	0	1	1	1	1	1	1	1
20	0	0	0	0	0	0	0	0	1	1	1	1
30	0	0	0	0	0	0	0	0	1	1	1	1
40	0	0	1	0	0	0	0	0	0	1	1	1
50	0	0	1	0	0	0	0	0	0	1	1	1
60	0	1	1	0	0	0	0	0	0	1	1	1
70	1	1	1	1	1	0	0	0	0	0	1	1
80	1	1	1	1	1	1	1	1	0	0	1	1
90	1	1	1	1	1	1	1	1	1	1	0	0
100	1	1	1	1	1	1	1	1	1	1	0	0

Table 4.19: T-Test For Highest Priority Demand Class

There is a significant difference in the total revenue between the extreme cases: The 90% and 100% threshold policies both differ significantly from the 0% and 5% threshold policies. Table 4.20 illustrates this effect. Though the revenue calculations are may be specific to a particular firm, this effect was not seen in the "Many Small Orders" case. This relative difference suggests that the more liberal splitting policies become increasingly effective as the average order size of customers increases.

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	0	0	0	0	0	0	0	1	1
5	0	0	0	0	0	0	0	0	0	0	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0	0	0	0
70	0	0	0	0	0	0	0	0	0	0	1	1
80	0	0	0	0	0	0	0	0	0	0	0	1
90	1	1	0	0	0	0	0	0	1	0	0	0
100	1	1	0	0	0	0	0	0	1	1	0	0

Table 4.20: T-Test For Total Revenue

4.6.3 Few Large Orders

In this scenario, few relatively large-sized orders are placed by all demand classes. The specific distributions that were used are given in table 4.21.

Demand Class	**1**	**2**	**3**	**4**
Scenario Few Large Orders				
Number Gens	5	5	5	5
μ Ords/step	5	5	5	5
Min Ords / Step	1	1	1	1
Max Ords/Step	9	9	9	9
σ Ords/Step	1	1	1	1
μ OE/ Ord	8	8	8	8
Min OE	7	7	7	7
Max OE	9	9	9	9
σ OE	0.3	0.3	0.3	0.3
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	4	4	4	4
Min DD	1	1	1	1
Max DD	8	8	8	8
σ DD	1	1	1	1

Table 4.21: Few Large Orders

The numeric results of this experiment are given in table 4.22.

	Total Revenue	**Class 1 %**	**Class 2 %**	**Class 3 %**	**Class 4 %**
0	$307,472,299.27	0.919	0.917	0.919	0.918
5	$306,684,746.87	0.919	0.919	0.919	0.919
10	$305,604,560.17	0.912	0.914	0.913	0.913
20	$306,178,757.01	0.912	0.913	0.913	0.909
30	$306,876,681.05	0.909	0.910	0.909	0.910
40	$307,038,499.79	0.913	0.912	0.910	0.911
50	$304,086,860.31	0.910	0.911	0.909	0.911
60	$303,766,296.45	0.906	0.905	0.902	0.907
70	$292,985,900.39	0.901	0.903	0.906	0.903
80	$286,805,895.48	0.882	0.885	0.887	0.888
90	$273,223,331.81	0.824	0.827	0.834	0.833
100	$267,920,666.65	0.813	0.813	0.818	0.818

Table 4.22: Simulation Results of Policies

The experimental results are illustrated graphically in figure 4.12. Like the previous two cases, there is a general increase in allocation percentage for each demand class as the splitting threshold is lowered. This is illustrated in table 4.23, which shows the t-test results of the allocation percentage of the highest priority demand class (used as a representative of all other demand classes, all of which exhibited near identical relationships). The effects of decreasing the splitting threshold are substantially more visible in this case than in the previous two.

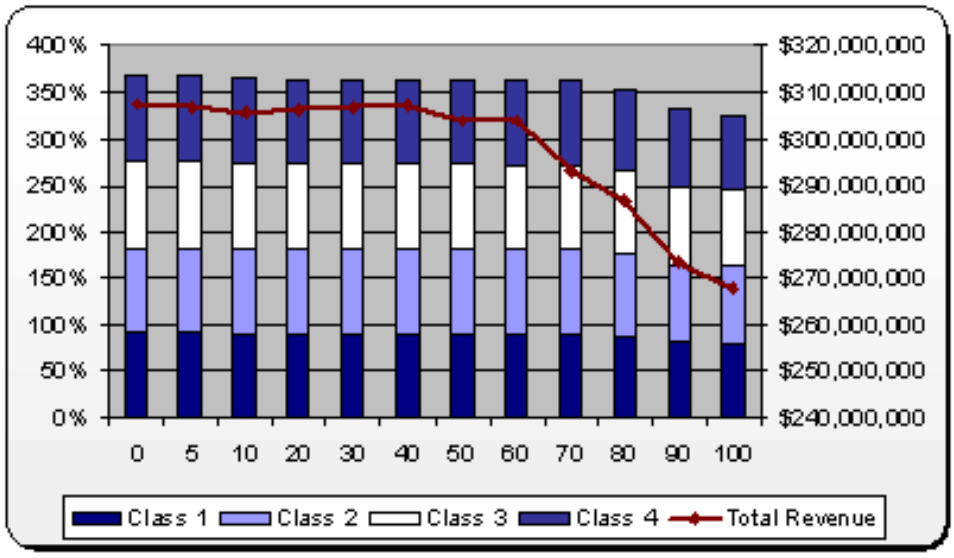

Figure 4.12: Allocation % (left axis) and Total Revenue (right axis) of Few Large Orders

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	1	1	1	0	1	1	1	1	1	1
5	0	0	1	1	1	0	1	1	1	1	1	1
10	1	1	0	0	0	0	0	0	1	1	1	1
20	1	1	0	0	0	0	0	0	1	1	1	1
30	1	1	0	0	0	0	0	0	1	1	1	1
40	0	0	0	0	0	0	0	0	1	1	1	1
50	1	1	0	0	0	0	0	0	1	1	1	1
60	1	1	0	0	0	0	0	0	0	1	1	1
70	1	1	1	1	1	1	1	0	0	1	1	1
80	1	1	1	1	1	1	1	1	1	0	1	1
90	1	1	1	1	1	1	1	1	1	1	0	0
100	1	1	1	1	1	1	1	1	1	1	0	0

Table 4.23: T-Test For Highest Priority Demand Class

In this third case, there are more significant differences in total revenue between different policies than in either of the previous two cases. Table 4.24 provides the t-test results.

	0	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
0	0	0	0	0	0	0	0	0	0	1	1	1
5	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	0	1	1
20	0	0	0	0	0	0	0	0	0	1	1	1
30	0	0	0	0	0	0	0	0	0	1	1	1
40	0	0	0	0	0	0	0	0	0	1	1	1
50	0	0	0	0	0	0	0	0	0	0	1	1
60	0	0	0	0	0	0	0	0	0	0	1	1
70	0	0	0	0	0	0	0	0	0	0	1	1
80	1	1	0	1	1	1	0	0	0	0	0	1
90	1	1	1	1	1	1	1	1	1	0	0	0
100	1	1	1	1	1	1	1	1	1	1	0	0

Table 4.24: T-Test For Total Revenue

4.6.4 Remarks

The effects of decreasing the splitting threshold became more significant as the customers' average order size increased. This is a fairly intuitive result. These case studies also uncovered less intuitive trends. One such example is the relationship revealed among policies in table 4.15. This table illustrates that, in the "Many Small Orders" case, decreasing the splitting threshold below 50% provides no significant increase in allocation percentage. This insight is readily gained through simulation, but is substantially more difficult to predict analytically.

Overall, these case studies reveal that the main determinant of the effectiveness of a splitting policy is the average size of the orders received. As such, a manufacturer should take expressive care to consider factor when deciding on a splitting policy.

4.7 Reserving Case Studies

A third key decision in an order promising policy is determining whether to reserve inventory for future orders when filling earlier orders. The trade-off involved in this challenge is that of realizing immediate revenue by filling as best as possible as they arrive, versus taking the risk that a higher paying customer will request the same inventory sometime in the future. A further consideration is the selection of shipping method by the manufacturer. Provided the options exist, the manufacturer must decide whether to allocate immediately available inventory and chose a slower shipping method, or whether to allocate inventory that will be ready several days in the future, and ship it using a speedier method. This challenge is also discussed in section 4.2.

The case studies in this section examine the effects of choosing to allocate inventory from different dates of availability. Specifically, three policies are considered:

(i) Forward Allocation - The manufacturer fills orders with inventory that is immediately available before allocating inventory that will be available in the future. Effectively, the manufacturer is working *forward* in time, from the current date to the due date, to select the inventory that will be allocated. This is the greedy policy; it seeks to generate revenue from the current order, without consideration of future demand. This method is illustrated in

figure 4.13.

(ii) Reverse Allocation - The manufacturer considers inventory that is close to the customer's order due date before considering more immediately available inventory. Effectively, the manufacturer is working backwards in time (hence *reverse*), from the due date to the current date, to select the inventory that will be allocated. This is a complete reservation policy; it seeks to preserve as much immediately available inventory as possible in order to fill future rush-orders. This method is illustrated in figure 4.14.

(iii) Random Allocation - The manufacturer considers inventory randomly. This is provided as a baseline case against which to measure the forward and reverse policies. There is no business meaning to this particular case.

To simulate the true business model more accurately, these policies were given an additional parameter. This parameter represented the number of days forward after which inventory was not considered. In the case studies below, this parameter was set at 5. As such, inventory that was available to ship on days 6, 7, and 8 was prohibited to promise. For example, consider an order with a requested delivery date of 8 days. Using this parameterized policy, inventory available from 1 day forward (the current day) to 5 days forward is eligible to be allocated. Inventory available after day 6 is not eligible.

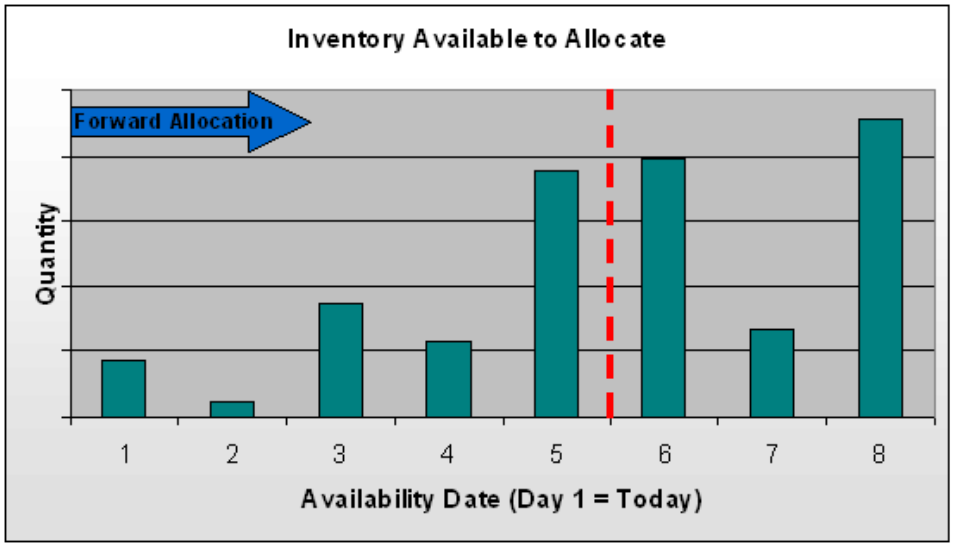

Figure 4.13: Forward Allocation Policy

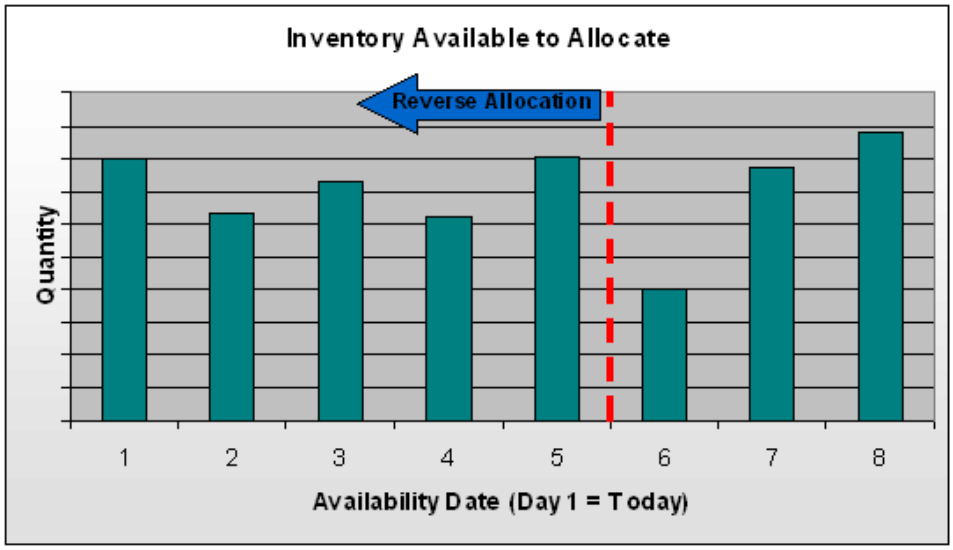

Figure 4.14: Reverse Allocation Policy

This parameter limits the number of last-minute overnight shipments the manufacturer may promise, especially when using the Reverse policy. In the example above, a raw version of the Reverse policy would consider inventory available 8 days forward, then 7 days forward, and so on, until a sufficient quantity is allocated. Assuming there is sufficient inventory available on day 8, the manufacturer would be faced with an expensive overnight shipment to deliver its products on day 9. Parameterizing the policy will preclude this from happening, as the speediest shipment it is allowed to send is 3-day delivery (at the end of day 5). The trade-off of using this parameter is reducing the pool of inventory by which a manufacturer may fill orders; no longer will the inventory on days 6, 7, and 8 be available to allocate.

Each of these policies, Forward, Reverse, and Random, were evaluated in three different demand environments. In each environment , four demand classes are used. In the first environment , labeled "Sooner to Later Due Dates," customers in the two higher priority demand classes place orders with early requested due dates, while customers in the two lower priority demand classes place orders with late requested due dates. In the second environment , labeled "Equal and Moderate," customers in all demand classes place orders with mid-range due dates. In the third environment , labeled "Later to Sooner Due Dates," customers in the two higher priority demand classes place orders with late requested due dates, while customers in the two lower priority demand classes place orders with early requested due dates. These three environments were tested individually to examine the how the impact of the reservation policies vary in different demand environments.

4.7.1 Sooner to Later Due Dates

In this scenario, the two higher priority demand classes request earlier due dates than the lower priority demand classes. The specific distributions that were used are given in table 4.25. Graphically, the distribution of requested due dates is given in figure 4.15.

Demand Class	**1**	**2**	**3**	**4**
Scenario Sooner to Later				
Number Gens	5	5	5	5
μ Ords/step	1	1	1	1
Min Ords / Step	1	1	1	1
Max Ords/Step	1	1	1	1
σ Ords/Step	0	0	0	0
μ OE/ Ord	5	5	5	5
Min OE	1	1	1	1
Max OE	9	9	9	9
σ OE	1	1	1	1
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	1	2	4	6
Min DD	1	2	4	6
Max DD	1	2	4	6
σ DD	0	0	0	0

Table 4.25: Sooner to Later Due Dates

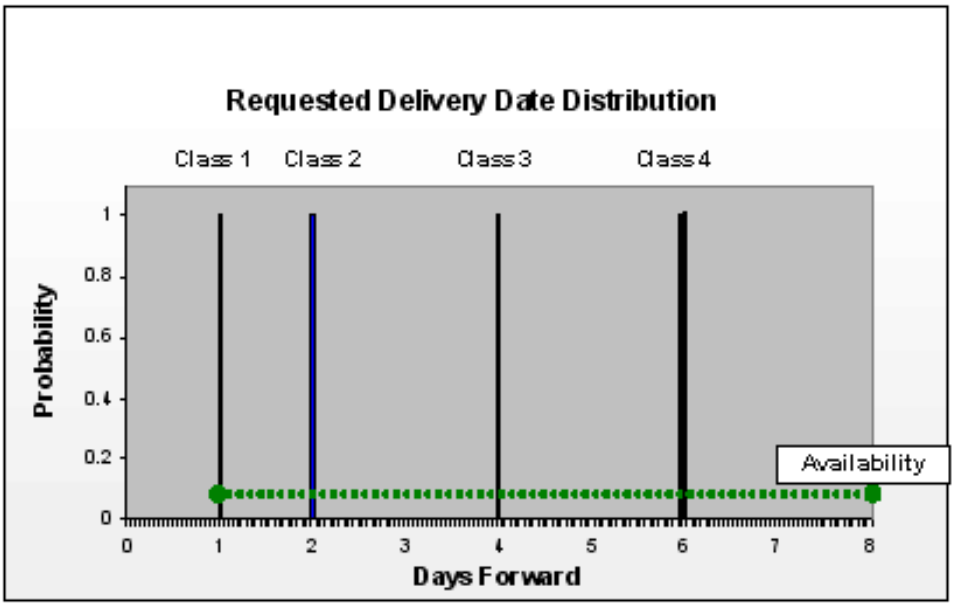

Figure 4.15: Sooner to Later Distribution

4.7.2 Equal and Moderate Due Dates

In this scenario, all demand classes request mid-range due dates. The specific distributions that were used are given in table 4.26. Graphically, this is given in figure 4.16.

Demand Class	**1**	**2**	**3**	**4**
Scenario Even				
Number Gens	5	5	5	5
μ Ords/step	1	1	1	1
Min Ords / Step	1	1	1	1
Max Ords/Step	1	1	1	1
σ Ords/Step	0	0	0	0
μ OE/ Ord	5	5	5	5
Min OE	1	1	1	1
Max OE	9	9	9	9
σ OE	1	1	1	1
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	3.5	3.5	3.5	3.5
Min DD	3	3	3	3
Max DD	4	4	4	4
σ DD	0.167	0.167	0.167	0.167

Table 4.26: Moderate Due Dates

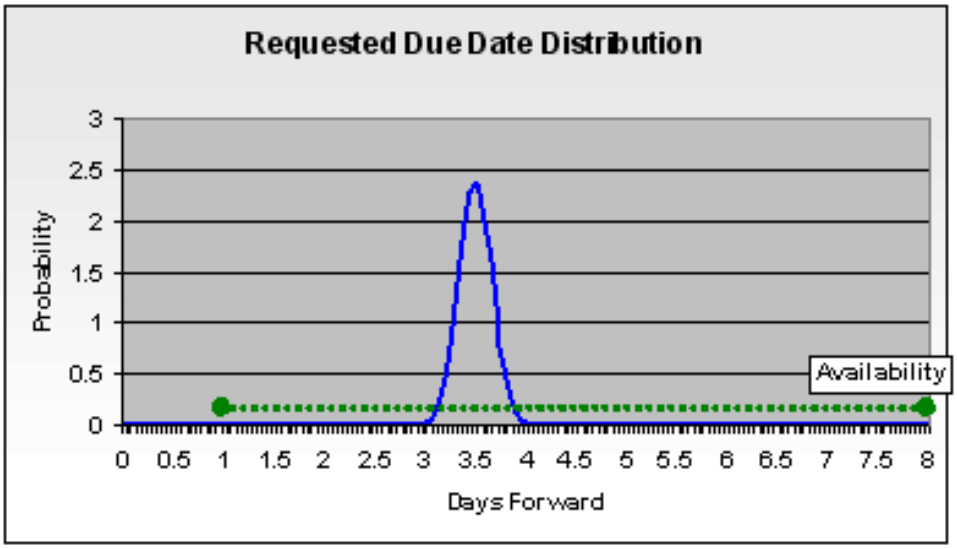

Figure 4.16: Moderate Distribution

4.7.3 Later to Sooner Due Dates

In this scenario, the two higher priority demand classes request later due dates than the lower priority demand classes. The specific distributions that were used are given in table 4.27. Graphically, the distribution of requested due dates is given in figure 4.17.

Demand Class		**1**	**2**	**3**	**4**
Scenario	Later to Sooner				
Number Gens		5	5	5	5
μ Ords/step		1	1	1	1
Min Ords / Step		1	1	1	1
Max Ords/Step		1	1	1	1
σ Ords/Step		0	0	0	0
μ OE/ Ord		5	5	5	5
Min OE		1	1	1	1
Max OE		9	9	9	9
σ OE		1	1	1	1
Qty/P		5	5	5	5
Min Qty/P		1	1	1	1
Max Qty/P		9	9	9	9
σ Qty/P		1	1	1	1
μ DD		6	4	2	1
Min DD		6	4	2	1
Max DD		6	4	2	1
σ DD		0	0	0	0

Table 4.27: Later to Sooner Due Dates

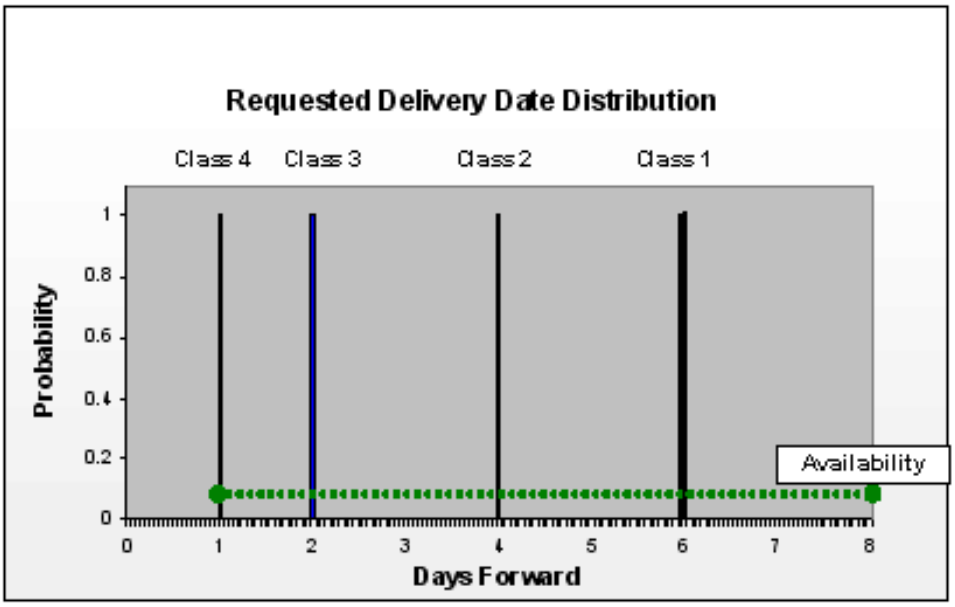

Figure 4.17: Later to Sooner Distribution

4.7.4 Results

The overall order allocation percentage for each of the three experiments is given in table 4.28. This contains some interesting results, which are displayed graphically in figure 4.18. Firstly, the allocation percentages of the "Soon to Far" and the "Far to Soon" environments are nearly identical. To understand why this is, it's important to recognize that these reservation policies do not distinguish high-priority demand classes from low-priority demand classes. As a result, all demand classes were treated equally. Because of this, the demand distributions of the "Soon to Far" and "Far to Soon" scenarios, as seen by the manufacturer, were identical. For a visual explanation, refer back to figure 4.15 and figure 4.17; eliminate the demand class labels and the figures become identical. This is the manufacturer's view of the demand.

Secondly, the allocation percentage for the moderate case is nearly equal for each of the three policies. This relates to the more concentrated distribution of due dates present in this environment. More specifically, each policy becomes limited to the inventory available in days 1 - 4 (this assumes that a negligible amount of orders are placed with a due date later than 4 days). The demand for each case is greater than the supply, and so each policy exhausts this inventory daily. This inventory represents about 76% of the total demand, and hence each policy has an overall allocation percentage of about 76%.

Thirdly, the Forward policy is inferior to both the Random and the Reverse policy in both the "Soon to Far" and the "Far to Soon" scenarios. The explanation for this is straightforward: The Forward policy exhausts the inventory that is available immediately to fill both rush and non-rush orders. It does this without regard to the possibility of receiving potential rush orders. The wide distribution of due dates in these cases, however, guarantees that rush orders will be placed throughout the day. As a result, the Forward policy will be unable to fill rush orders that arrive late in the day. The Reverse policy, on the other hand, will have reserved this immediately available inventory. This policy will be able to fill late-arriving rush orders, and hence allocate a greater percentage of orders.

	Soon to Far	Med to Med	Far to Soon
Forward	0.926	0.760	0.928
Reverse	0.998	0.760	0.999
Random	0.966	0.762	0.968

Table 4.28: Allocation % Results

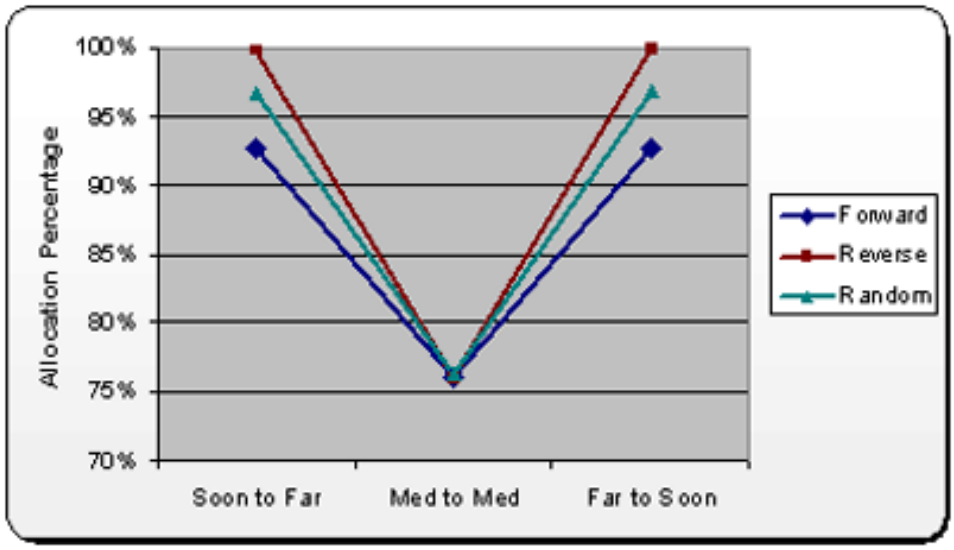

Figure 4.18: Allocation Percentage of Different Policies

For both the "Soon to Far" and the "Far to Soon" cases, the allocation percentages for all three policies differ significantly. This is illustrated in table 4.29 and table 4.30. The moderate case exhibits no significant differences (not shown).

	Forward	**Reverse**	**Random**
Forward	0	1	1
Reverse	1	0	1
Random	1	1	0

Table 4.29: T-Test For Allocation % (Sooner to Later)

	Forward	**Reverse**	**Random**
Forward	0	1	1
Reverse	1	0	1
Random	1	1	0

Table 4.30: T-Test For Allocation % (Later to Sooner)

It is also interesting to examine the revenue results of the case studies in this section. This is under the caveat that absolute revenue measurements are dependent on the specific pricing structure used. Nonetheless, because consistent price formulas were used in all three policies, relative effects among the policies reveal general trends.

The total revenue results are given in table 4.31. These are also given graphically in figure 4.19. Of particular interest in these results is the "Far to Soon" case. In this case, the Reverse policy is significantly superior than the Forward policy (see table 4.32 for t-test results). This is the only instance in which one policy produces a significant difference in revenue than another (i.e. all other differences are statistically insignificant). Recall though that the allocation percentages for both the "Soon to Far" and the "Far to Soon" case were near identical. This implies that the composite allocations in the two scenarios differed. This is almost surely a result of the Reverse policy filling more orders from higher paying customers in the "Far to Soon" environment than in the "Soon to Far" environment. This makes sense when the demand distributions are considered. In the "Far to Soon" environment, the higher paying customers place more rush orders. These are difficult to fill because there is a relatively small amount of inventory available to fill such orders. For example, to fill an order that must be received within 2 days, only inventory from day 1 may be used. In contrast, to fill an order that must be received within 8 days, inventory from days 1-7 may be used. Given this, it make sense that more higher-priority orders were filled in the "Far to Soon" environment than in the "Soon to Far" environment. Consequently, more revenue was generated, as is evident in these simulation results.

	Soon to Far	**Med to Med**	**Far to Soon**
Forward	\$136,975,753.34	\$143,324,246.19	\$144,991,918.38
Reverse	\$149,501,955.25	\$143,120,744.81	\$161,160,604.58
Random	\$149,934,677.18	\$146,551,530.12	\$155,782,802.90

Table 4.31: Total Revenue Results

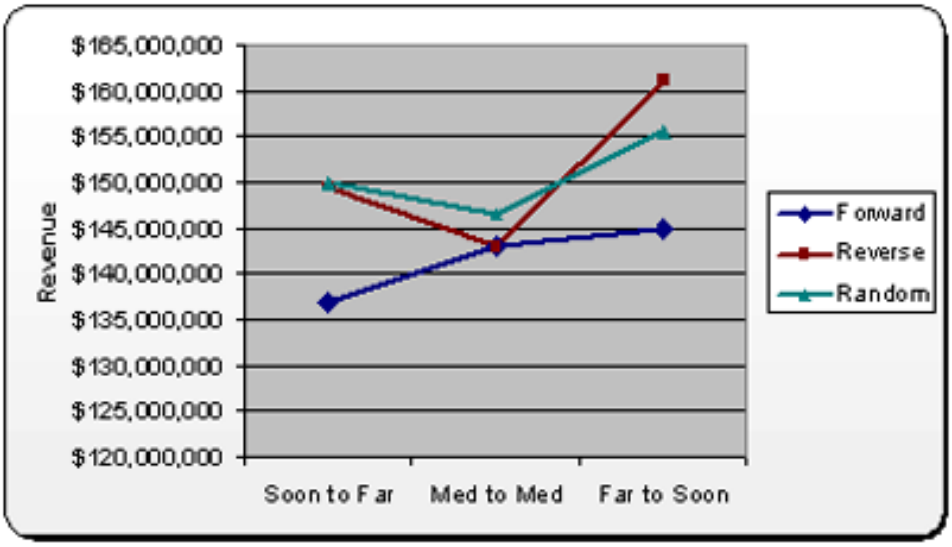

Figure 4.19: Revenue of Different Policies

	Forward	**Reverse**	**Random**
Forward	0	1	0
Reverse	1	0	0
Random	0	0	0

Table 4.32: T-Test For Revenue (Far to Soon)

4.7.5 Remarks

The effects of reserving inventory for future orders is illustrated in the results of the case studies in this section. These effects are dependent on the due date distribution of customers' orders. It, therefore, is important for a manufacturer to understand this distribution prior to implementing a reservation strategy.

4.8 Thresholding Case Study

The case studies in this section replicate two of the thresholding experiments documented by Wollmer in his 1992 work [54]. The first case study in this section replicates the no-class restriction policy. The second study replicates the optimization model. These are given primarily as an illustration of the capabilities of our test-bed , and not necessarily to provide insight into a particular business policy.

4.8.1 No-Class Restriction Policy

In this scenario, no protection levels were enforced for the ticket inventory. The demand distributions that were used are given in table 4.33. These are identical to the distributions used in the original experiment.

Demand Class	**1**	**2**	**3**	**4**	**5**
Scenario Base					
Number Gens	1	1	1	1	1
μ Ords/step	17.326	45.052	39.55	34.018	19.786
Min Ords / Step	0	0	0	0	0
Max Ords/Step	35	91	80	69	40
σ Ords/Step	5.775	15.017	13.183	11.339	6.595
μ OE/ Ord	1	1	1	1	1
Min OE	1	1	1	1	1
Max OE	1	1	1	1	1
σ OE	0	0	0	0	0
Qty/P	1	1	1	1	1
Min Qty/P	1	1	1	1	1
Max Qty/P	1	1	1	1	1
σ Qty/P	1	1	1	1	1
μ DD	1	1	1	1	1
Min DD	1	1	1	1	1
Max DD	1	1	1	1	1
σ DD	0	0	0	0	0

Table 4.33: Base Scenario

The revenue results of the simulated policy are statistically equivalent with 95% confidence to the results of the original experiment. This was determined by calculating a 95% confidence interval around the mean of both sets of revenue results. The component values of these confidence intervals are displayed in table 4.34 and in table 4.35. Figure 4.20 illustrates that these confidence intervals overlap, which is an indication that the replication of this experiment produced accurate results.

	DF	**T-Value (95%)**	**Mean**	**Std. Dev.**
Simulation	299	1.968	60,579.95	6,199.82
Wollmer (1992)	199,997	1.960	60,892.42	5,355.56

Table 4.34: T-Test for Thresholding Policy

	Low	**Mean**	**High**
Simulation	59,875.53	60,579.95	61,284.36
Wollmer (1992)	60,868.95	60,892.42	60,915.89

Table 4.35: Confidence Intervals for Non-Thresholding Policy

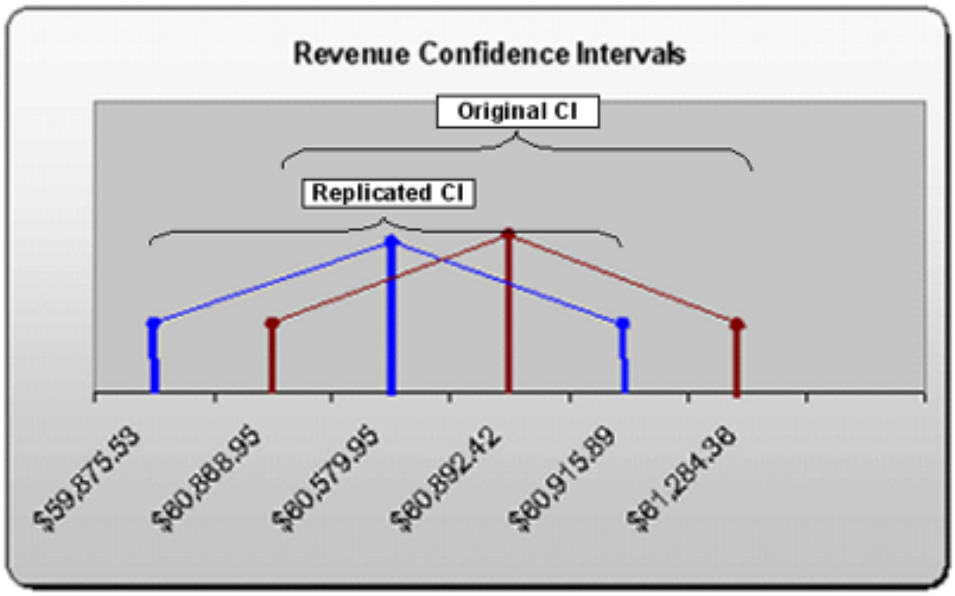

Figure 4.20: Confidence Intervals for Non-Thresholding Policy

4.8.2 Optimization Policy

In this scenario, optimal protection levels were enforced for the ticket inventory. The demand distributions used in this scenario were the same as used in the previous scenario, and are given in table 4.33.

The revenue results of the simulated policy are statistically equivalent with 95% confidence to the results of the original experiment. This was determined by calculating a 95% confidence interval around the mean of both sets of revenue results. The component values of these confidence intervals are displayed in table 4.36 and in table 4.37. Figure 4.21 illustrates that these confidence intervals overlap, which is an indication that the replication of this experiment produced accurate results.

	DF	**T-Value (95%)**	**Mean**	**Std. Dev.**
Simulation	299	1.968	69,205.47	6,199.82
Wollmer (1992)	199,997	1.960	69,365.73	5,355.56

Table 4.36: T-Test for Thresholding Policy

	Low	**Mean**	**High**
Simulation	68,501.06	69,205.47	69,909.89
Wollmer (1992)	69,342.26	69,365.73	69,389.20

Table 4.37: Confidence Intervals for Thresholding Policy

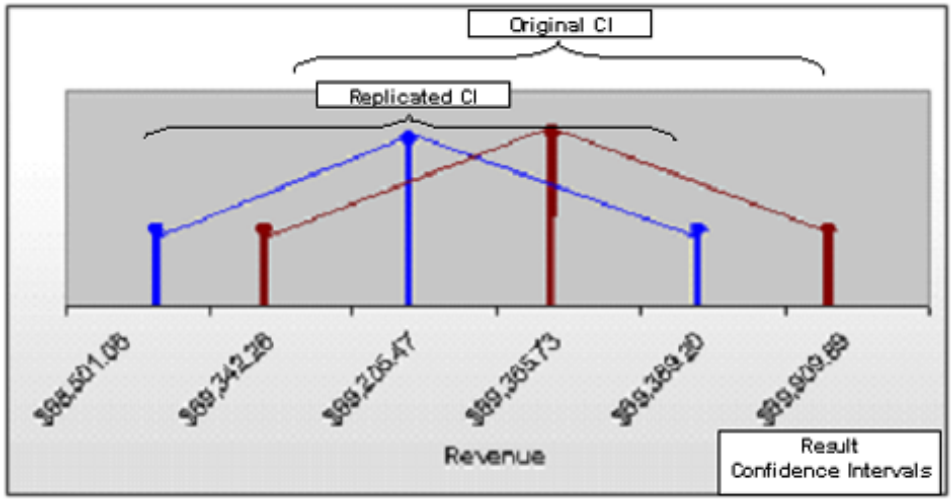

Figure 4.21: Confidence Intervals for Thresholding Policy

4.8.3 Remarks

The precision of the results of the replicated experiments in this section was corroborated to a 95% confidence level. This illustrates the ability of our test-bed to replicate accurately evaluations of existing revenue management policies.

4.9 Adapting Case Study

In this section, the stochastic adaptive algorithm as developed by van Ryzin and McGill (2000) is replicated [50]. This is complemented with a replication of the optimal policy that is also derived in their work. Like the previous section, these are given primarily as an illustration of the capabilities of our test-bed, and not necessarily to provide insight into a particular business policy.

4.9.1 Adaptive Policy

The demand distributions that were used in this scenario are given in table 4.38. These are identical to the distributions used in the original experiment.

Demand Class	**1**	**2**	**3**	**4**
Scenario Base				
Number Gens	1	1	1	1
μ Ords/step	17.326	45.052	73.568	19.786
Min Ords / Step	0	0	0	0
Max Ords/Step	35	91	148	40
σ Ords/Step	5.775	15.017	17.389	6.595
μ OE/ Ord	1	1	1	1
Min OE	1	1	1	1
Max OE	1	1	1	1
σ OE	0	0	0	0
Qty/P	1	1	1	1
Min Qty/P	1	1	1	1
Max Qty/P	1	1	1	1
σ Qty/P	1	1	1	1
μ DD	1	1	1	1
Min DD	1	1	1	1
Max DD	1	1	1	1
σ DD	0	0	0	0

Table 4.38: Base Scenario

The values of the protection levels over time are illustrated in figure 4.22. No numerical analysis was done to compare these values to the original values, but the convergence is similar to that presented in the original work.

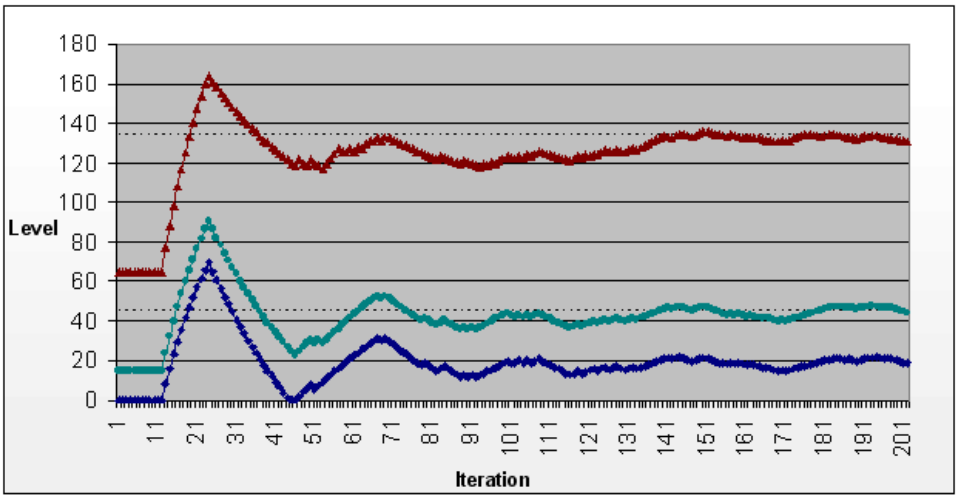

Figure 4.22: Thresholds from the Adaptive Policy

4.9.2 Remarks

The results of this section do not provide an analytical measurement of accuracy, but they do provide a visual assurance that the experiment was replicated successfully. Further work may be done to corroborate this more effectively. Nonetheless, the implementation of this adaptive policy provides another example of the flexibility offered by our test-bed.

4.10 Responding Case Study

This section presents a study to illustrate the capability of our test-bed to include dynamic customer behavior. Specifically, the customers modeled respond to the results of the orders they place (i.e. to what extend their orders are filled completely). If they are unsatisfied with the results, the customers will discontinue placing orders temporarily. This type of behavior may influence the policy decisions made by the manufacturer. For instance, a large number of unhappy lower paying customers may provide motivation to the manufacturer to decrease its customer preference based on segmentation of demand classes.

The customer behavior described above was implemented by calculating a simple moving average of a customer's most recent order allocation rates. A moving average was used to represent the emphasis of a customer's short-term memory when making purchasing decisions in an elastic market. Before placing an order on a given day, each customer compared this moving average to a pre-determined threshold. If the moving average was below the threshold value, the customer did not place an order on that day. The moving average formula considered non-orders to be non-detrimental. This ensured that a dissatisfied customer agent resumed placing orders as soon it "forgot" about its poor order history.

A baseline case in which customers do not respond to their order status is also given in this section. This is done to provide a benchmark against which the effects of dynamic customer behavior may be compared.

4.10.1 Results

The demand distributions used in this case study are given in table 4.39. This represents a typical base-case, in which all distributions have moderate values.

Demand Class	**1**	**2**	**3**	**4**
Scenario Base				
Number Gens	5	5	5	5
μ Ords/step	1	1	1	1
Min Ords / Step	1	1	1	1
Max Ords/Step	1	1	1	1
σ Ords/Step	0	0	0	0
μ OE/ Ord	5	5	5	5
Min OE	1	1	1	1
Max OE	9	9	9	9
σ OE	1	1	1	1
Qty/P	5	5	5	5
Min Qty/P	1	1	1	1
Max Qty/P	9	9	9	9
σ Qty/P	1	1	1	1
μ DD	4	4	4	4
Min DD	1	1	1	1
Max DD	8	8	8	8
σ DD	1	1	1	1

Table 4.39: Base Scenario

Figure 4.23 illustrates the revenue realized by a manufacturer in two different demand scenarios. The first scenario includes dynamic customer behavior (the responding case), and the second does not (the non-responding case). It is clear that there are differences in revenue between the two scenarios. In some instances, these differences are significant, and in others they are not. The total mean revenue for both scenarios is given in table 4.40. This table also presents the results of a t-test performed against the revenues of both scenarios. There is no significant difference between the two mean revenue statistics, but only by a small amount.

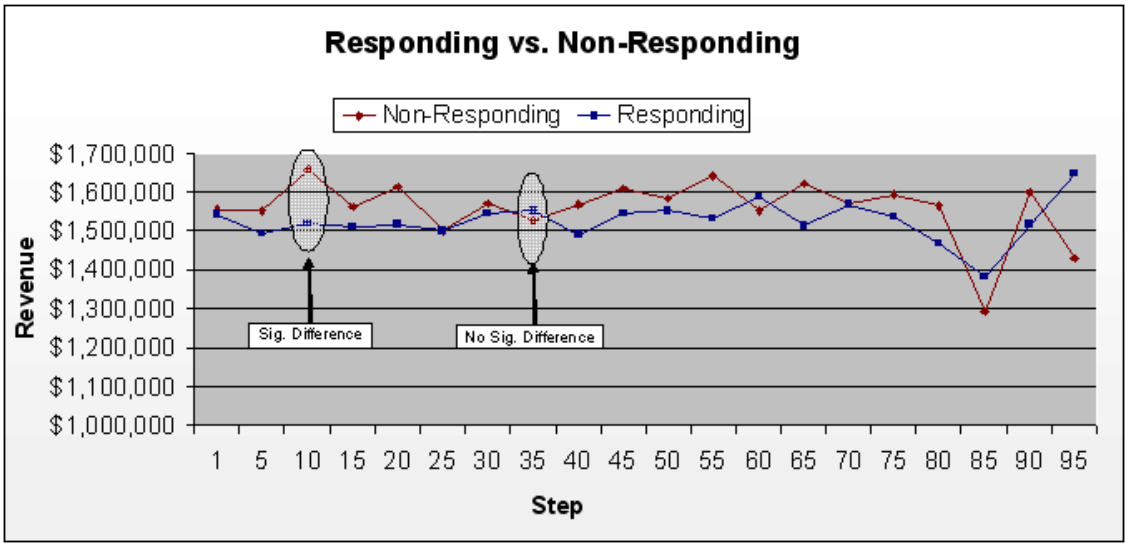

Figure 4.23: Revenue from Different Customer Types

	Non-Responding	**Responding**
Mean	$150,512,650.00	$144,748,950.00
Std. Dev	$9,200,941.97	$13,080,542.57
T-Stat (95%)	1.974	
T-Crit (2 Tail)	2.002	

Table 4.40: Aggregated Results

4.10.2 Remarks

A purpose of this experiment was to illustrate how dynamic customer behavior may affect the total revenue generated by a manufacturer. This experiment did not produce a significant difference in mean revenue, but it did highlight significant daily differences. Comparing revenue statistics in this study, however, was secondary to its primary purpose. This was to provide a proof of concept for our test-bed. To this end, it was successful as it illustrated the ability of our test-bed to incorporate dynamic customer behavior into revenue management simulations.

Chapter 5

Conclusions and Future Research

The chief objective of this thesis is to produce an extensible test-bed as a tool to simulate revenue management policies in a vast range of business scenarios. Three sub-objectives are defined as requisites of accomplishing this goal. The first is to translate the essential components of different business environments into an agent-oriented software model. Chapter 3 describes how this was accomplished, and presents the main elements of this software model. The second sub-objective is to construct an extensible test-bed using the software model as a blueprint. The key steps taken to achieve this also are described in chapter 3. Insight to some of the techniques used is also provided. The last sub-objective is to use the test-bed to evaluate order promising and revenue management policies in different business scenarios. Chapter 4 presents several case studies that test different revenue management policies in simulated demand environments. Additionally, an analysis of experimental results is provided to illustrate how the test-bed provides strategic business insight.

5.1 Observations and Future Extensions

We achieved our chief objective by producing an agent-based test-bed that meets our business requirements. We also performed several revenue management experiments to evaluate different revenue management policies. We are satisfied with these accomplishments, but recognize that there are many opportunities to extend the system further. Several possible extensions are suggested below.

Currently, the test-bed runs on one machine as one process. It would be interesting to extend this to include the use of distributed agents. Decentralizing `OrderGenerator` agents, for

example, introduces the possibility of using humans as agent controllers. Imagine the following design: Remote terminals are used to house distributed `OrderGenerator` agents. A human directly controls the actions of each agent through a user interface (UI). An agents receives this input, and then communicates with other agents to realize its goals. In this scenario, the human effectively has become an agent within the test-bed. Because of its agent-oriented design, however, this substitution is transparent to the other components in the system. Further, because the underlying JAVA^{TM} framework, RePast, is open-sourced, the implementation of this extension is substantially simplified.

A second possible extension is to introduce aggregate spikes in demand at arbitrary times in the simulation. These may represent environmental factors to which real-life customers respond. For instance, consider a simulation of an airline's reservation process. Demand for flights typically follow historical trends, and, as such, a forecasted demand curve frequently is used to model aggregate demand. Occasionally, though, an event is announced that induces an acute increase in demand. The revealing of the NCAA Men's Tournament Bracket is one example of this - announcing where and when colleges play post-season basketball games prompts a sudden ticket demand by loyal fans. To simulate this effect, a broadcast communication must notify all `OrderGenerator` agents of the announcement. The means to construct this device is provided by the RePast framework. Additionally, the `OrderGenerator` agents must understand the meaning of the announcement, as well as how to react to it. Imparting this understanding requires augmenting the agents' knowledge of its environment. Both of these modifications are readily performed in our test-bed, and so this suggestion is also a feasible extension.

In a real-world business environment, customers often have the choice of upgrading to a higher-priority demand class; typically, a customer pays a cost premium to do this. For example, an air traveler may purchase a first-class ticket if all lower-fare coach tickets are sold out. Allowing the `OrderGenerator` agents the freedom to make this choice is the sole addition our model needs test-bed to simulate this effect. You (2001) analyzes such an scenario in his 2001 work, but no use of simulation is included [56]. As such, applying You's policies to an empirical study using our

test-bed would be an interesting and feasible experiment to perform.

A final observation is on the use of the RePast framework. We invested a considerable amount of time analyzing over one-hundred frameworks before we chose RePast. We are satisfied, however, that this was a prudent decision. RePast provided a sound object-oriented foundation on top of which we were able to build our agent-based test-bed. Further, the developers of RePast assisted us several times when we encountered an implementation issue. In a manner of reciprocation, we also contributed assistance to others with similar issues. On the whole, we are very satisfied with the functionality provided by RePast, and the support offered by its community.

Appendix A

Test-bed Implementation

A.1 A Factory Pattern and a Dependency Injection Pattern Example

In this section, an illustration is given of the use of the Factory and Dependency Injection Patterns. Specifically, this concentrates on the relationship between the class, `SimOrderProcessor`, and the abstract class, `SimOrderOutputRecorder`. This relationship is simply put: A `SimOrderProcessor` uses exactly one `SimOrderOutputRecorder` to record the results of its order processing. Moreover, although there are many different types of `SimOrderOutputRecorder`s, an `SimOrderProcessor` must only be cognizant of the general contract that must be fulfilled by any instance of any type of a `SimOrderOutputRecorder`. This is because this contract effectively separates the underlying implementation of each type of `SimOrderOutputRecorder` from the interface it presents to other classes. As such, the communication between a `SimOrderProcessor` and any type of `SimOrderOutputRecorder` is identical. Hence, the behavior of the `SimOrderProcessor` does not change according to the type of `SimOrderOutputRecorder` it uses. An illustration of this relationship is given in figure A.1.

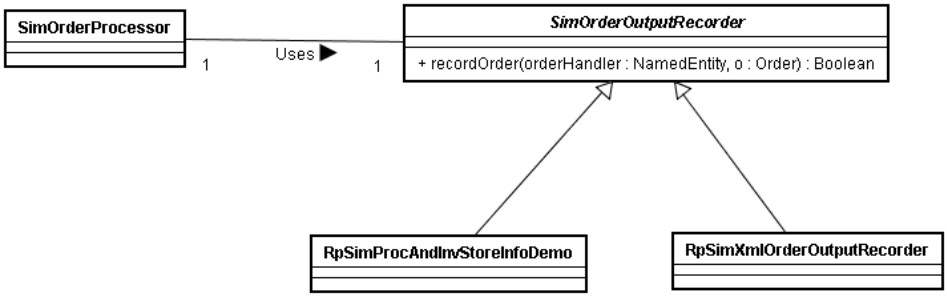

Figure A.1: Relationship between `SimOrderProcessor` and `SimOrderOutputRecorder`

The specific type of `SimOrderOutputRecorder` that is used by the `SimOrderProcessor`s typically is chosen by a user from the RePast GUI when the test-bed loads. In this case, the user has a discrete set of options from which to chose. This set is defined in the `RpSimOrderOutputRecorderFactory` class. This class acts as a Factory of `SimOrderOutputRecorder`s. That is, it is used to produce instances of the `SimOrderOutputRecorder` class for use by other objects in the test-bed. This occurs when a `RpSimOrderOutputRecorderFactory` object receives a request for a particular type of `SimOrderOutputRecorder`. The internals of this `RpSimOrderOutputRecorderFactory` class are given on the following pages.

```java
package edu.umd.isr.dcsim.implementation.repastsim.logic;

import edu.umd.isr.common.tools.implementation.SingletonManager;
import edu.umd.isr.dcsim.implementation.logic.SimOrderOutputRecorder;
import edu.umd.isr.dcsim.implementation.logic.SimOrderOutputRecorderFactory;
import edu.umd.isr.dcsim.logic.NamedEntity;
import edu.umd.isr.dcsim.logic.Order;
import edu.umd.isr.dcsim.logic.StepDataProvider;
import edu.umd.isr.dcsim.util.FactoryInitInfo;

/**
 ********************************************************************************
 * File        : $Source: /afs/glue.umd.edu/.../RpSimOrderOutputRecorderFactory.java,v $
 * Version     : $Revision: 1.12 $
 * Date        : $Date: 2005/03/25 19:08:13 $
 * Modified by : $Author: ffaber $
 * Copyright   : Copyright (c) 2003 University of Maryland
 ********************************************************************************
 */
public class RpSimOrderOutputRecorderFactory extends SimOrderOutputRecorderFactory {

    // Tags representing the different types of OrderOutputRecorders
    public static final String DOM4J_XML_TYPE_PLACED = "dom4jXmlTypePlaced";
    public static final String DOM4J_XML_TYPE_PROCED = "dom4jXmlTypeProced";
    public static final String FLAT_FILE_PROCED = "flatFileProced";
    public static final String SUM_FLAT_FILE_PROCED = "sumFlatFileProced";
    public static final String PLUG_TYPE_FOR_PROCED = "plugType";
    public static final String FILE_NAME = "fileName";
    public static final String PLUG_CLASS = "plugClass";

    // Specialty output recorders
    public static final String AGGR_REV_ALR_DC_ALR_COL_FLAT_FILE_PROCED = "aggrRevAlrDcAlrColFlatFileProced";
    public static final String AGGR_REV_ALR_COL_FLAT_FILE_PROCED = "aggrRevAlrColFlatFileProced";
    public static final String AGGR_REV_COL_FLAT_FILE_PROCED = "aggrRevColFlatFileProced";
    public static final String PER_STEP_REV_FLAT_FILE_PROCED = "perStepRevFlatFileProced";

    private static RpSimOrderOutputRecorderFactory instance;
    static { instance = new RpSimOrderOutputRecorderFactory(); }

    public static RpSimOrderOutputRecorderFactory getInstance() { return instance; }

    private RpSimOrderOutputRecorderFactory() { super(); }

    /**
     * Method returns a SimOrderOutputRecorder, the specific type of which is determined
     *  the method parameters.
     * @param sii - An object is populated by the class requesting a specific
     *  SimOrderOutputRecorder type.  This object holds the parameters necessary to identify
     *  which type is needed, and, in some cases, what its initial state should be.
     * @return An object that realizes the SimOrderOutputRecorder interface.
     */
    public SimOrderOutputRecorder createSimOrderOutputRecorder( FactoryInitInfo sii ) {

        String s = sii.getClassTag();
        SimOrderOutputRecorder sdo = null;

        // Determine the specific type needed using the "ClassTag" string
        if( s.equals( RpSimOrderOutputRecorderFactory.DOM4J_XML_TYPE_PLACED ) ) {
            RpSimXmlOrderOutputRecorder rsdo = new RpSimXmlPlacedOrderOutputRecorder();
            String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
            rsdo.setFileName( fileName );
            sdo = rsdo;
        } else if( s.equals( RpSimOrderOutputRecorderFactory.DOM4J_XML_TYPE_PROCED ) ) {
            RpSimXmlOrderOutputRecorder rsdo = new RpSimXmlProcessedOrderOutputRecorder();
            String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
            rsdo.setFileName( fileName );
            sdo = rsdo;
        } else if( s.equals( RpSimOrderOutputRecorderFactory.FLAT_FILE_PROCED )) {
            RpSimFlatFileProcessedOrderOutputRecorder rff = new RpSimFlatFileProcessedOrderOutputRecorder();
            String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
            rff.setFileName( fileName );
            sdo = rff;
        } else if( s.equals( RpSimOrderOutputRecorderFactory.SUM_FLAT_FILE_PROCED ) ) {
            RpSimSumFlatFileProcessedOrderOutputRecorder rff = new RpSimSumFlatFileProcessedOrderOutputRecorder();
            String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
            rff.setFileName( fileName );
            sdo = rff;
        } else if (s.equals( RpSimOrderOutputRecorderFactory.AGGR_REV_ALR_DC_ALR_COL_FLAT_FILE_PROCED ) ) {
            RpSimGenericColSumFlatFileProcessedOrderOutputRecorder rff =
                    new RpSimGenericColSumFlatFileProcessedOrderOutputRecorder();
            rff.setWriteAggrRev( true );
            rff.setWriteDcAlr( true );
            String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
```

```
84              rff.setFileName( fileName );
85              sdo = rff;
86          } else if (s.equals( RpSimOrderOutputRecorderFactory.AGGR_REV_ALR_COL_FLAT_FILE_PROCED ) ) {
87              RpSimGenericColSumFlatFileProcessedOrderOutputRecorder rff =
88                      new RpSimGenericColSumFlatFileProcessedOrderOutputRecorder();
89              rff.setWriteAggrAlr( true );
90              rff.setWriteAggrRev( true );
91              String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
92              rff.setFileName( fileName );
93              sdo = rff;
94          } else if (s.equals( RpSimOrderOutputRecorderFactory.AGGR_REV_COL_FLAT_FILE_PROCED ) ) {
95              RpSimGenericColSumFlatFileProcessedOrderOutputRecorder rff =
96                      new RpSimGenericColSumFlatFileProcessedOrderOutputRecorder();
97              rff.setWriteAggrRev( true );
98              String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
99              rff.setFileName( fileName );
100             sdo = rff;
101         } else if (s.equals( RpSimOrderOutputRecorderFactory.PER_STEP_REV_FLAT_FILE_PROCED ) ) {
102             RpSimGenericColSumFlatFileProcessedOrderOutputRecorder rff =
103                     new RpSimGenericColSumFlatFileProcessedOrderOutputRecorder();
104             rff.setWriteStepRev( true );
105             String fileName = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME );
106             rff.setFileName( fileName );
107             sdo = rff;
108         } else if( s.equals( RpSimOrderOutputRecorderFactory.PLUG_TYPE_FOR_PROCED ) ) {
109             String className = (String)sii.getClassProp( RpSimOrderOutputRecorderFactory.PLUG_CLASS );
110             SingletonManager sm = SingletonManager.getInstance();
111             Object plugRef = sm.getReference( className );
112             sdo = (SimOrderOutputRecorder)plugRef;
113         } else { // base case
114             sdo = new SimOrderOutputRecorder() {
115                 public Boolean initOutSource() { return Boolean.FALSE; }
116                 public Boolean recordOrder( NamedEntity orderHandler, Order o ) { return Boolean.FALSE; }
117                 public void alertToPreStep() { return; }
118                 public void alertToPostStep() { return; }
119                 public void alertToFinalStep() { return; }
120                 public void setStepDataProvider( StepDataProvider sdp ) { return; }
121          };
122         }
123
124         // Return the object instantiated and populated above
125         return sdo;
126
127     }
128
129 }
130
```

To exploit fully the `RpSimOrderOutputRecorderFactory` class, the test-bed must offer options to the user to select different types of `SimOrderOutputRecorder`s. This is done by mapping human-readable options (i.e. to display on the GUI) to unique class-identifying tags embedded in the code. This mapping is shown in the lines 2813-2839 of the `RpSimSimpleModel` class (shown in the following pages). After the user makes a selection and starts the simulation, the test-bed must pass the selection to the Factory class. The Factory class, in response, returns a `SimOrderOutputRecorder` object. This is shown in lines 4382-4393 of the `RpSimSimpleModel` class. Once this is done, the test-bed must link the `SimOrderOutputRecorder` object to the appropriate `SimOrderProcessor` object. Otherwise phrased, the test-bed is *injecting* an object on which the `SimOrderProcessor` object *depends* into the `SimOrderProcessor`; this describes exactly the Dependency Injection Pattern. This is applied because delegating this responsibility into the framework greatly simplifies its implementation. This in contrast to establishing relationships directly among `SimOrderProcessor` objects and other supporting infrastructure, such as the RePast GUI. Doing this would exponentially increase the complexity of the test-bed, and severely hinder its development. Hence, this is why the test-bed acts as a liaison between mechanical structures and functional components.

```
2813     /**
2814      * Method maps types of OrderOutputRecorders to different options on the RePast GUI.
2815      * When a user selects an option on the GUI, the tag associated with it will be used to
2816      * determine which type of OrderOutputRecroder the user selected.
2817      * @return PropertyDescriptor - An object to map class tags to GUI options
2818      */
2819     protected PropertyDescriptor createOrdProcRecTypesPd() {
2820         Hashtable ordProcRecTypes = new Hashtable();
2821         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.DOM4J_XML_TYPE_PROCED, "XML Recorder" );
2822         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.FLAT_FILE_PROCED, "Flat File Recorder" );
2823         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.SUM_FLAT_FILE_PROCED, "Sum Flat File Recorder" );
2824         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.PLUG_TYPE_FOR_PROCED, "Pluggable Type" );
2825         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.AGGR_REV_ALR_DC_ALR_COL_FLAT_FILE_PROCED,
2826                              "AggrRevAlr DcAlr Col Sum Flat File Type" );
2827         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.AGGR_REV_ALR_COL_FLAT_FILE_PROCED,
2828                              "AggrRevAlr Col Sum Flat File Type" );
2829         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.AGGR_REV_COL_FLAT_FILE_PROCED,
2830                              "AggrRev Col Sum Flat File Type" );
2831         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.PER_STEP_REV_FLAT_FILE_PROCED,
2832                              "PerStepRev Col Sum Flat File Type" );
2833
2834         ordProcRecTypes.put( RpSimOrderOutputRecorderFactory.DEFAULT_TYPE, "Default" );
2835         ListPropertyDescriptor ordProcRecTypesPd =
2836                 new ListPropertyDescriptor( PN_ORD_PROC_REC_TYPE, ordProcRecTypes );
2837
2838         return ordProcRecTypesPd;
2839     }
.........
3281     /**
3282     ** Method buildModel() creates the objects used in the simulation.
3283     ** Agents are created in this method and given handles to their environment.
3284     */
3285     public void buildModel() {
.........
3827
3828         // Create the OutputRecorder for the Experimental OrderProcessor
3829         expOrdProcRec =
3830                 this.createOrderOutputRecorderForProcAndProcName( EXP_PROC_NAME );
3831         expOrdProcRec.setStepDataProvider( stepDataProvider );
3832         expOrdProcRec.initOutSource();
3833
3834         // Inject the OrderRecorder into the OrderProcessor.  The specific type of
3835         // OutputRecorder is irrelevent to the OrderProcessor; any type injected will
3836         // fulfill the contract defined by the SimOrderProcessor interface.
3837         expOrdProc.setOrderOutputRecorder( expOrdProcRec );
.........
3938
3939         return;
3940     }
.........
4379
4380     // .. further in the code
4381
4382     protected SimOrderOutputRecorder createOrderOutputRecorderForProcAndProcName( String n ) {
4383         FactoryInitInfo sii = new SimCompactFactoryInitInfo();
4384         sii.setClassTag( this.getOrdProcRecType() );
4385         sii.setClassProp( RpSimOrderOutputRecorderFactory.FILE_NAME, this.getOrdProcOutputFileNameForProcName(n) );
4386         sii.setClassProp( RpSimOrderOutputRecorderFactory.PLUG_CLASS, this.getOrdProcOutputPlugClass() );
4387         return this.createOrderOutputRecorderForInitInfo( sii );
4388     }
4389
4390     protected SimOrderOutputRecorder createOrderOutputRecorderForInitInfo( FactoryInitInfo sii ) {
4391         SimOrderOutputRecorderFactory fact = this.getSimOrderOutputRecorderFactory();
4392         return fact.createSimOrderOutputRecorder(sii);
4393     }
```

The code on the following pages presents two particular types of `SimOrderOutputRecorder`s. The first class is the `RpSimXmlOrderOutputRecorder` class. This is used to record order processing results into XML files. The class second is the `RpSimProcAndInvStoreInfoDemo` class. This is used for a very different purpose - it calculates policy statistics to update the display of a customized GUI. These are given to emphasize how wide the difference may be between two sub-types of the `SimOrderOutputRecorder` class. Despite these dissimilarities, however, a `SimOrderOutputRecorder` object communicates with each type in exactly the same manner. Specifically, this is done via the `recordOrder()` method, the parameters and return value of which are defined in the contract of a `SimOrderOutputRecorder`. Assuring such consistent means of communication is the value of creating and using well-defined interfaces.

```java
package edu.umd.isr.dcsim.implementation.repastsim.logic;

import edu.umd.isr.dcsim.implementation.logic.SimOrder;
import edu.umd.isr.dcsim.implementation.logic.SimOrderOutputRecorder;
import edu.umd.isr.dcsim.implementation.util.SimLogger;
import edu.umd.isr.dcsim.implementation.util.SimLoggerManager;
import edu.umd.isr.dcsim.implementation.util.SimXmlOutputRecorder;
import edu.umd.isr.dcsim.logic.*;
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;

import java.text.SimpleDateFormat;
import java.util.Date;

/**
 ********************************************************************************
 * File        : $Source: /afs/glue.umd.edu/.../RpSimXmlOrderOutputRecorder.java,v $
 * Version     : $Revision: 1.8 $
 * Date        : $Date: 2005/01/26 22:16:57 $
 * Modified by : $Author: ffaber $
 * Copyright   : Copyright (c) 2003 University of Maryland
 ********************************************************************************
 */
public class RpSimXmlOrderOutputRecorder extends SimXmlOutputRecorder
                                         implements SimOrderOutputRecorder {

    private static final SimLogger logger =
            SimLoggerManager.getSimLogger( RpSimXmlOrderOutputRecorder.class.getName() );
    private static SimLogger getLogger() { return logger; }

    public RpSimXmlOrderOutputRecorder() { super(); }

    public synchronized void alertToPreStep() {
        // begin <step> tag
        Document doc = this.getXmlDoc();
        Element root = doc.getRootElement();
        Element stepElem = root.addElement( STEP_TAG );
        stepElem.addElement( STEP_COUNT_TAG ).addText( this.getStringStepCount() );
        this.setCurrStepElem( stepElem );
        this.setXmlDoc( doc );
        return;
    }

    public synchronized void alertToPostStep() { return; }

    public void alertToFinalStep() {
        this.writeXmlFile();
        return;
    }

    public synchronized Boolean recordOrder( NamedEntity orderHandler, Order o ) {
        Boolean retVal = Boolean.TRUE;
        return retVal;
    }

    protected Element createOrderMetaDataXml( Element parentElem, Order o ) {

        Element e = parentElem;
        DemandClass dc = o.getDemandClass();
        OrderPlacer op = o.getOrderPlacer();
        String placerName = op.getName();

        Date datePlaced = (Date)o.getProperty( SimOrder.DATE_PLACED );
        String dateString = null;
        SimpleDateFormat sdf = new SimpleDateFormat( DATE_FORMAT );
        dateString = sdf.format( datePlaced );

        String nanosString = o.getProperty( SimOrder.NANOS_PLACED ).toString();

        e = e.addElement( ORD_DATA_TAG );
        e.addElement( ORD_PLACER_TAG ).addText( placerName );
        e.addElement( ORD_DC_TAG ).addText( dc.dcToString() );
        e.addElement( ORD_DATE_TAG ).addText( dateString );
        e.addElement( ORD_MILLIS_TAG ).addText( nanosString );

        return e;
    }

    protected Element createOrderElementXml( Element parentElem, OrderElement oe ) {

        Element e = parentElem;
        ProductRequest pr = oe.getProductRequest();
```

```
84          Product p = pr.getProduct();
85          Integer q = pr.getIntInvQty();
86          Period r  = pr.getPeriod();
87
88          Object obj = null;
89          String prodType = ( (obj = p.getProductType()) == null ? DEF_PROD_TYPE : obj.toString() );
90
91          e = e.addElement( ORD_ELEMENT_TAG );
92          e.addElement( OE_PRODUCT_TYPE_TAG ).addText( prodType );
93          e.addElement( OE_QTY_TAG ).addText( q.toString() );
94          e.addElement( OE_ABS_PERIOD_TAG ).addText( r.getIntegerVal().toString() );
95
96          return e;
97
98      }
99
100     protected void initializeXmlDoc() {
101         Document doc = DocumentHelper.createDocument();
102         Element root = doc.addElement( RpSimXmlOrderOutputRecorder.ORDERS_PER_STEP_TAG );
103         doc.setRootElement( root );
104         this.setXmlDoc( doc );
105         return;
106     }
107
108
109 }
110
```

```java
package edu.umd.isr.apps.dcsimdemo.implementation.repastsimdemo.logic;

import edu.umd.isr.apps.dcsimdemo.logic.OrderProcessorActionListener;
import edu.umd.isr.apps.dcsimdemo.logic.ResourceInventoryStoreActionListener;
import edu.umd.isr.common.math.PercentagePair;
import edu.umd.isr.common.tools.implementation.SingletonManager;
import edu.umd.isr.dcsim.implementation.logic.SimOrderOutputRecorder;
import edu.umd.isr.dcsim.implementation.logic.SimPeriod;
import edu.umd.isr.dcsim.implementation.logic.SimResourceInvStoreOutputRecorder;
import edu.umd.isr.dcsim.implementation.repastsim.logic.RpSimSimpleModel;
import edu.umd.isr.dcsim.implementation.util.SimLogger;
import edu.umd.isr.dcsim.implementation.util.SimLoggerManager;
import edu.umd.isr.dcsim.logic.*;

import java.util.*;

/**
 * ***************************************************************************
 * File        : $Source: /afs/glue.umd.edu/.../RpSimProcAndInvStoreInfoDemo.java,v $
 * Version     : $Revision: 1.16 $
 * Date        : $Date: 2005/01/26 22:16:56 $
 * Modified by : $Author: ffaber $
 * Copyright   : Copyright (c) 2004 University of Maryland
 * ***************************************************************************
 */
/**
 * Class subclasses off the RpSimModelDemo class to use the RpSimSimpleModel model.
 * It specializes in the listeners and action events unique to the demo.
 */
public class RpSimProcAndInvStoreInfoDemo extends RpSimModelDemo implements SimResourceInvStoreOutputRecorder,
                                                                             SimOrderOutputRecorder {

    private static final SimLogger logger =
            SimLoggerManager.getSimLogger( RpSimProcAndInvStoreInfoDemo.class.getName() );
    private static SimLogger getLogger() { return logger; }

    public static final String QUALIFIED_CLASS_NAME  =
            "edu.umd.isr.apps.dcsimdemo.implementation.repastsimdemo.logic.RpSimProcAndInvStoreInfoDemo";

    // Proc stat variables
    public static final String AGGR_STATS_KEY = "aggrStats";
    public static final String DC_STATS_KEY = "dcStats";
    public static final String PROD_STATS_KEY = "productStats";
    public static final String ABS_PER_STATS_KEY = "absPeriodStats";
    public static final String REL_PER_STATS_KEY = "relPeriodStats";

    public static final String AGGR_VAL = "Aggregate";
    public static final String DEF_PROD_TYPE = "0";

    protected List<String> procNames;
    public List<String> getProcNames() { return this.procNames; }
    public void setProcNames( List<String> l ) { this.procNames = l; }

    protected Map<String, Map<String, Map<String, PercentagePair>>> percentageStats;
    public Map<String, Map<String, Map<String, PercentagePair>>> getPercentageStats() {
        return this.percentageStats;
    }
    public void setPercentageStats(
            Map<String,Map<String, Map<String, PercentagePair>>> m ) {
        this.percentageStats = m;
    }

    protected Map<String, Map<String,  Map<String, Double>>> monetaryStats;
    public Map<String, Map<String, Map<String, Double>>> getMonetaryStats() { return this.monetaryStats; }
    public void setMonetaryStats( Map<String, Map<String, Map<String, Double>>> m ) { this.monetaryStats = m; }


    // Inv stat variables
    public static final int MAX_FORWARD_PERS = 10;

    protected List<String> invNames;
    public List<String> getInvNames() { return this.invNames; }
    public void setInvNames( List<String> l ) { this.invNames = l; }

    // Proc -> product -> ({per,qty}, {per,qty}, {per,qty}...)
    protected Map<String, Map<String, List<Integer>>> inventoryStats;
    public Map<String, Map<String, List<Integer>>> getInventoryStats() { return this.inventoryStats; }
    public void setInventoryStats( Map<String, Map<String, List<Integer>>> m ) { this.inventoryStats = m; }


    // Event variables
    protected OrderProcessorActionListener orderProcessorActionListener;
    public OrderProcessorActionListener getOrderProcessorActionListener() { return this.orderProcessorActionListener;
```

```
84      public void setOrderProcessorActionListener( OrderProcessorActionListener pal ) {
85          this.orderProcessorActionListener = pal;
86      }
87
88      protected ResourceInventoryStoreActionListener resourceInventoryStoreActionListener;
89      public ResourceInventoryStoreActionListener getResourceInventoryStoreActionListener() {
90          return this.resourceInventoryStoreActionListener;
91      }
92      public void setResourceInventoryStoreActionListener( ResourceInventoryStoreActionListener isl ) {
93          this.resourceInventoryStoreActionListener = isl;
94      }
95
96      public RpSimProcAndInvStoreInfoDemo() {
97          super();
98          linkToModel();
99          initStatData();
100         initEvents();
101         initListeners();
102         registerAsPluggableClass();
103     }
104
105     protected void initListeners() {
106         OrderProcessorActionListener pal = new OrderProcessorActionListener() {
107             public void alertToOrderProcessorAction( OrderProcessor op, Order o ) {
108                 String name = op.getName(),
109                         msg = "Getting order (" + o.toString() + ") from " + name;
110                 getLogger().debug( msg );
111                 RpSimProcAndInvStoreInfoDemo.this.fireSimModelDemoEvent( ordProcEvent );
112             }
113         };
114         this.setOrderProcessorActionListener( pal );
115
116         ResourceInventoryStoreActionListener isl = new ResourceInventoryStoreActionListener() {
117             public void alertToInventoryStoreAction( ResourceInventoryStore ris ) {
118                 String name = ris.getName();
119                 getLogger().debug( "Getting inventory change from: " + name );
120                 RpSimProcAndInvStoreInfoDemo.this.fireSimModelDemoEvent( prodInvEvent );
121             }
122         };
123         this.setResourceInventoryStoreActionListener( isl );
124     }
125
126     protected void registerAsPluggableClass() {
127         SingletonManager sm = SingletonManager.getInstance();
128         sm.addReference( QUALIFIED_CLASS_NAME, this);
129     }
130
131
132     public Boolean initOutSource() { return Boolean.TRUE; }
133
134     protected void initStatData() {
135
136         // 1. Processor
137         HashMap<String, Map<String, Map<String, PercentagePair> > >perStats =
138                 new HashMap<String, Map<String, Map<String, PercentagePair>>>( 12, .75f );
139         HashMap<String, Map<String, Map<String, Double> > > monStats =
140                 new HashMap<String, Map<String, Map<String, Double>>>( 12, .75f );
141
142         List< String > procNames = RpSimSimpleModel.getProcessorNames();
143         this.setProcNames( procNames );
144         for( String name : procNames ) {
145
146             // 1. Percentage stats
147             Map<String, Map<String, PercentagePair>> procPerStats =
148                     new HashMap<String, Map<String, PercentagePair>>( 12, .75f );
149
150             procPerStats.put( AGGR_STATS_KEY, new HashMap<String, PercentagePair>() );
151             procPerStats.put( DC_STATS_KEY, new HashMap<String, PercentagePair>() );
152             procPerStats.put( ABS_PER_STATS_KEY, new HashMap<String, PercentagePair>() );
153             procPerStats.put( REL_PER_STATS_KEY, new HashMap<String, PercentagePair>() );
154             procPerStats.put( PROD_STATS_KEY, new HashMap<String, PercentagePair>() );
155             perStats.put( name, procPerStats );
156
157             // 2. Monetary stats
158             Map<String, Map<String, Double>> procMonStats  =  new HashMap<String, Map<String, Double>>( 12, .75f );
159             procMonStats.put( AGGR_STATS_KEY, new HashMap<String, Double>() );
160             procMonStats.put( DC_STATS_KEY, new HashMap<String, Double>() );
161             procMonStats.put( ABS_PER_STATS_KEY, new HashMap<String, Double>() );
162             procMonStats.put( REL_PER_STATS_KEY, new HashMap<String, Double>() );
163             procMonStats.put( PROD_STATS_KEY, new HashMap<String, Double>() );
164
165             monStats.put( name, procMonStats );
166         }
167         this.setPercentageStats( perStats );
168         this.setMonetaryStats( monStats );
169
170         // 2. Inventory data
```

```
171         Map<String, Map<String, List<Integer>>> invStats =
172                 new HashMap<String, Map<String, List<Integer>>>( 200, .75f );
173 
174         List< String > invNames = RpSimSimpleModel.getInventoryNames();
175         this.setInvNames( invNames );
176         for( String invName : invNames ) {
177 
178             final int
179                     maxPersPerProd =
180                         (Integer)this.getModelProperty( RpSimModelDemoDataTags.MP_MAX_NUM_LT_PERS_PER_PROD ),
181                     maxSteps = RpSimSimpleModel.MAX_NUM_STEPS,
182                     totSteps = maxPersPerProd + maxSteps,
183                     maxNumProds = RpSimSimpleModel.MAX_PROD_TYPE;
184 
185             Map<String, List<Integer>> prodInvStats =  new HashMap<String, List<Integer>>(maxNumProds*2, .75f);
186 
187             for( int j = RpSimSimpleModel.MAX_PROD_TYPE; j-- > 0; ) {
188 
189                 List< Integer > periodProdInvStats = new ArrayList< Integer >(totSteps);
190                 // For all periods
191                 for( int k = totSteps; --k >= 0; ) { periodProdInvStats.add( 0 ); }
192                 prodInvStats.put( String.valueOf(j), periodProdInvStats );
193             }
194             invStats.put( invName, prodInvStats );
195         }
196 
197         this.setInventoryStats( invStats );
198         return;
199 
200     }
201 
202     public synchronized Boolean recordOrder( NamedEntity orderHandler, Order o ) {
203 
204         Boolean retVal = Boolean.TRUE;
205 
206         String  aggrVal = AGGR_VAL,
207                 demandClass = o.getDemandClass().dcToString(),
208                 prodType = null,
209                 oeQty = null,
210                 oeAbsPeriod = null,
211                 oeRelPeriod = null,
212                 procName = orderHandler.getName();
213 
214         Object obj = null;
215 
216         // Statistic Data
217         Map<String,Map<String, Map<String, PercentagePair>>> allProcPerStats = this.getPercentageStats();
218         Map<String, Map<String, PercentagePair>> percentageStats = allProcPerStats.get( procName );
219 
220         Map<String, PercentagePair> aggrPercentageStats = percentageStats.get( AGGR_STATS_KEY ),
221                 dcPercentageStats = percentageStats.get( DC_STATS_KEY ),
222                 aperPercentageStats = percentageStats.get( ABS_PER_STATS_KEY ),
223                 rperPercentageStats = percentageStats.get( REL_PER_STATS_KEY ),
224                 prodPercentageStats = percentageStats.get( PROD_STATS_KEY );
225 
226         Map<String, Map<String, Map<String, Double>>> allProcMonStats = this.getMonetaryStats();
227         Map<String, Map<String, Double> > monetaryStats = allProcMonStats.get( procName );
228 
229         Map<String, Double> aggrMonetaryStats = monetaryStats.get( AGGR_STATS_KEY ),
230                 dcMonetaryStats = monetaryStats.get( DC_STATS_KEY ),
231                 aperMonetaryStats = monetaryStats.get( ABS_PER_STATS_KEY ),
232                 rperMonetaryStats = monetaryStats.get( REL_PER_STATS_KEY ),
233                 prodMonetaryStats = monetaryStats.get( PROD_STATS_KEY );
234 
235         List ordElems = o.getOrderElements();
236         int ordElemsSize = ordElems.size();
237 
238         if( ordElemsSize > 0 ) {
239 
240             for( int i = ordElemsSize; --i >= 0; ) {
241                 OrderElement oe = (OrderElement)ordElems.get(i);
242                 ProductRequest pr = oe.getProductRequest();
243                 Product p = pr.getProduct();
244                 prodType = ( (obj = p.getProductType()) == null ? DEF_PROD_TYPE : obj.toString() );
245 
246                 oeQty = pr.getIntInvQty().toString();
247 
248                 Period absPer = pr.getPeriod(),
249                         relPer = absPer.absoluteToRelativePeriod( absPer );
250                 oeAbsPeriod = absPer.getIntegerVal().toString();
251                 oeRelPeriod = relPer.getIntegerVal().toString();
252 
253                 OrderStatus os = o.getOrderStatus();
254                 boolean hasAllocation = os.isCommitted().booleanValue();
255 
256                 List prodAllocs = oe.getProductRequestAllocations();
257                 int prodAllocsSize = prodAllocs.size();
```

```
258
259                 double  oeAllocatedQty = 0.0d,
260                         oeRequestedQty = Double.valueOf( oeQty ),
261                         oeMonetaryRel = 0.0d;
262
263                 // The order has been at least partially allocated - sum the amounts
264                 if( hasAllocation || prodAllocsSize == 0 ) {
265
266                     for( int j = prodAllocsSize; --j >= 0; ) {
267                         ProductAllocation pa = (ProductAllocation)prodAllocs.get(j);
268                         oeAllocatedQty += Double.valueOf( pa.getIntInvQty().toString() );
269                         oeMonetaryRel += Double.valueOf( pa.getMonentaryRelation().toString() );
270                     }
271                 }
272
273                 // Add the new totals to the running stats
274                 PercentagePair  pp = null,
275                         aggrPp = (pp = aggrPercentageStats.get( aggrVal )) == null ?
276                                  new PercentagePair( 0.0d, 0.0d ) : pp,
277                         dcPp = (pp = dcPercentageStats.get( demandClass )) == null ?
278                                new PercentagePair( 0.0d, 0.0d ) : pp,
279                         aperPp = (pp = aperPercentageStats.get( oeAbsPeriod )) == null ?
280                                  new PercentagePair( 0.0d, 0.0d ) : pp,
281                         rperPp = (pp = rperPercentageStats.get( oeRelPeriod )) == null ?
282                                  new PercentagePair( 0.0d, 0.0d ) : pp,
283                         prodPp = (pp = prodPercentageStats.get( prodType )) == null ?
284                                  new PercentagePair( 0.0d, 0.0d ) : pp;
285
286                 aggrPp.addFraction( oeAllocatedQty, oeRequestedQty );
287                 dcPp.addFraction( oeAllocatedQty, oeRequestedQty );
288                 aperPp.addFraction( oeAllocatedQty ,oeRequestedQty );
289                 rperPp.addFraction( oeAllocatedQty ,oeRequestedQty );
290                 prodPp.addFraction( oeAllocatedQty, oeRequestedQty );
291
292                 // Simplify for cases where keys aren't present above
293                 aggrPercentageStats.put( aggrVal, aggrPp );
294                 dcPercentageStats.put( demandClass, dcPp );
295                 aperPercentageStats.put( oeAbsPeriod, aperPp );
296                 rperPercentageStats.put( oeRelPeriod, rperPp );
297                 prodPercentageStats.put( prodType, prodPp );
298
299
300                 Double  d = null;
301                 double  aggrMonRel = (d = aggrMonetaryStats.get( aggrVal )) == null ?
302                                      new Double(0.0d) : d,
303                         dcMonRel = (d = dcMonetaryStats.get( demandClass )) == null ?
304                                    new Double(0.0d) : d,
305                         aperMonRel = (d = aperMonetaryStats.get( oeAbsPeriod )) == null ?
306                                      new Double(0.0d) : d,
307                         rperMonRel = (d = rperMonetaryStats.get( oeRelPeriod )) == null ?
308                                      new Double(0.0d) : d,
309                         prodMonRel = (d = prodMonetaryStats.get( prodType )) == null ?
310                                      new Double(0.0d) : d;
311
312                 // Autoboxing doesn't mutate
313                 aggrMonRel += oeMonetaryRel;
314                 dcMonRel += oeMonetaryRel;
315                 aperMonRel += oeMonetaryRel;
316                 rperMonRel += oeMonetaryRel;
317                 prodMonRel += oeMonetaryRel;
318
319                 aggrMonetaryStats.put( aggrVal, new Double( aggrMonRel ) );
320                 dcMonetaryStats.put( demandClass, new Double( dcMonRel ) );
321                 aperMonetaryStats.put( oeAbsPeriod, new Double( aperMonRel ) );
322                 rperMonetaryStats.put( oeRelPeriod, new Double( rperMonRel ) );
323                 prodMonetaryStats.put( prodType, new Double( prodMonRel ) );
324
325             }
326
327         }
328
329
330         return  retVal;
331     }
332
333
334
335     public synchronized Boolean recordResourceInvStore( ResourceInventoryStore ris ) {
336         Boolean retVal = Boolean.TRUE;
337
338         String  risName = ris.getName();
339
340         Integer oldInvQty = null,
341                 newInvQty = null,
342                 addInvQty = null,
343                 invPeriod = null;
344
```

```
345         Map<String, Map<String, List<Integer>>> allInvStats = this.getInventoryStats();
346         Map<String, List<Integer>> invStats = allInvStats.get( risName );
347
348
349         int winPeriodSize = this.getNumDifferentPers();
350         final Period windowPeriod = new SimPeriod( winPeriodSize );
351         final Period absWinPeriod = windowPeriod.relativeToAbsolutePeriod( windowPeriod );
352
353         // Iterate over all products
354         List prods = ris.getProds();
355         for( int i = prods.size(); --i >= 0; ) {
356             Product p = (Product)prods.get(i);
357
358             // Get all remaining inventory group sets for that period (List of Sets)
359             List invGroups = ris.getInvGroupsForProdWithinPeriod( p, absWinPeriod );
360
361             // Get the current calculation and init the updates
362             String prodType = p.getProductType().toString();
363             List<Integer> prodStats = invStats.get( prodType );
364             int currIndex = absWinPeriod.getIntegerVal();
365             for( int r = windowPeriod.getIntegerVal(); r-- >= 0; ) {
366                 prodStats.set( currIndex - r, 0 );
367             }
368
369             // Increment the period quantity accordingly
370             for( int j = invGroups.size(); --j >= 0; ) {
371                 Set invGrp = (Set)invGroups.get(j);
372
373                 for( Iterator k = invGrp.iterator(); k.hasNext(); ) {
374                     ResourceInventory rsi = (ResourceInventory)k.next();
375
376                     boolean isLinked = rsi.isLinkedToOrder();
377                     if( isLinked ) continue;
378
379                     // Increment the quantity
380                     Period  rsiPeriod = rsi.getPeriod();
381                     invPeriod = rsiPeriod.getIntegerVal();
382
383                      newInvQty = rsi.getIntInvQty();
384                     prodStats.set( invPeriod, newInvQty );
385
386                 }
387             }
388         }
389
390         return retVal;
391     }
392
393
394 }
395
396
```

A.2 A Unit Test Example

Given in the following code is an example of a unit test used in the test-bed. Of note in this unit test is the simplicity of the behavior it tests: It simply sets an initial value of an attribute (line 48), and then tests to see if this is the value it receives when querying the attribute (line 50). Perhaps it even seems embarrassing that this test exists. This notion of testing absolute fundamental functionality, however, is the basis of unit testing. Several times throughout development this exhibited its worth, as simple errors were noticed before they became hopelessly hidden in more complex operations.

```java
package edu.umd.isr.dcsim.implementation.repastsim.test;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import uchicago.src.reflector.Introspector;

import java.beans.IntrospectionException;
import java.lang.reflect.InvocationTargetException;
import java.util.Hashtable;

/**
 * ********************************************************************************
 * File        : $Source: /afs/glue.umd.edu/.../IntrospectorTest.java.html,v $
 * Version     : $Revision: 1.1 $
 * Date        : $Date: 2005/04/19 05:09:19 $
 * Modified by : $Author: ffaber $
 * Copyright   : Copyright (c) 2004 University of Maryland
 * ********************************************************************************
 */
public class IntrospectorTest extends TestCase {

    public static final int DEF_TEST_VAL_1 = 233;
    public static final String PN_TEST_VAL_1 = "TestVal1";
    public int testVal1 = DEF_TEST_VAL_1;
    public int getTestVal1() { return testVal1; }
    public void setTestVal1( int testVal1 ) { this.testVal1 = testVal1; }

    public String[] getInitParam() { return new String[] { PN_TEST_VAL_1 }; }

    protected void setUp() { this.setTestVal1( DEF_TEST_VAL_1 ); }
    protected void tearDown() { ; }
    public static Test suite() { return new TestSuite(IntrospectorTest.class); }

    public IntrospectorTest(String name) { super(name); }

    public void testIntrospectorSetMethods() {

        Introspector i = new Introspector();
        Integer valBeforeCallingSetter   = new Integer( this.getTestVal1() ),
                valAfterCallingSetter    = null,
                valWithWhichToCallSetter = new Integer( 344 );

        Hashtable modelProps = null;

        try {
            i.introspect( this, this.getInitParam() );
            i.invokeSetMethod( PN_TEST_VAL_1, valWithWhichToCallSetter );
            valAfterCallingSetter = new Integer( this.getTestVal1() );
            assertEquals( valWithWhichToCallSetter, valAfterCallingSetter );

            modelProps = i.getPropValues();
            Object testVal = modelProps.get( PN_TEST_VAL_1 );
            assertEquals( valWithWhichToCallSetter, testVal );

        } catch( IntrospectionException e ) {
            e.printStackTrace( System.err );
        } catch( InvocationTargetException e ) {
            e.printStackTrace( System.err );
        } catch( IllegalAccessException e ) {
            e.printStackTrace( System.err );
        }

        return;
    }

    public static void main( String[] args ) {
        new junit.textui.TestRunner().doRun(suite());
    }



}
```

A.3 Product Metrics

Three final product metrics are provided in this section. They are:

Cp(rec) - The number of classes and interfaces contained in a package, and recursively in its subpackages.

LOCp(rec) - The number of lines of product code contained in a package, and recursively in its subpackages. Product code includes both comments and source code, but not whitespace.

SLOC(rec) - The number of lines of source code contained in a package, and recursively in its subpackages. This includes only lines of source code.

package	**Cp(rec)**	**LOCp(rec)**	**SLOC(rec)**
edu	545	30348	24097
edu.umd	545	30348	24097
edu.umd.isr	545	30348	24097
edu.umd.isr.apps	35	2926	2532
edu.umd.isr.apps.dcsimdemo	34	2912	2527
edu.umd.isr.apps.dcsimdemo.implementation	29	2728	2391
edu.umd.isr.apps.dcsimdemo.implementation.logic	9	279	218
edu.umd.isr.apps.dcsimdemo.implementation.repastsimdemo	20	2449	2173
edu.umd.isr.apps.dcsimdemo.implementation.repastsimdemo.logic	6	751	686
edu.umd.isr.apps.dcsimdemo.implementation.repastsimdemo.ui	14	1698	1487
edu.umd.isr.apps.dcsimdemo.logic	2	31	13
edu.umd.isr.apps.dcsimdemo.ui	3	153	123
edu.umd.isr.common	37	1505	1180
edu.umd.isr.common.io	12	461	343
edu.umd.isr.common.io.implementation	9	416	325
edu.umd.isr.common.io.implementation.ioexample	3	121	86
edu.umd.isr.common.math	6	290	268
edu.umd.isr.common.tools	9	392	313
edu.umd.isr.common.tools.implementation	7	343	282
edu.umd.isr.common.ui	10	362	256
edu.umd.isr.common.ui.implementation	4	201	145

Table A.1: Product Metrics

package	**Cp(rec)**	**LOCp(rec)**	**SLOC(rec)**
edu.umd.isr.dcsim	472	25893	20370
edu.umd.isr.dcsim.implementation	370	24183	19637
edu.umd.isr.dcsim.implementation.logic	165	6994	5181
edu.umd.isr.dcsim.implementation.repastsim	151	13247	11165
edu.umd.isr.dcsim.implementation.repastsim.logic	116	11569	9760
edu.umd.isr.dcsim.implementation.repastsim.test	25	1377	1137
edu.umd.isr.dcsim.implementation.repastsim.test.aspects	1	17	8
edu.umd.isr.dcsim.implementation.repastsim.ui	10	301	268
edu.umd.isr.dcsim.implementation.test	29	2428	2149
edu.umd.isr.dcsim.implementation.test.fox	10	734	603
edu.umd.isr.dcsim.implementation.test.fox.chart	10	734	603
edu.umd.isr.dcsim.implementation.util	25	1514	1142
edu.umd.isr.dcsim.implementation.util.normalplot	4	240	184
edu.umd.isr.dcsim.logic	92	1543	653
edu.umd.isr.dcsim.util	10	167	80
edu.umd.isr.dynreqsel	1	24	15
edu.umd.isr.dynreqsel.implementation	1	24	15
edu.umd.isr.dynreqsel.implementation.ui	1	24	15

Table A.2: Product Metrics (continued)

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Andronache, V. and M. Scheutz. 2004. "Integrating Theory and Practice: The Agent Architecture Framework APOC and its Development Environment ADE." *AAMAS '04*.

[2] Baiocchi, G. 2004. "Using PERL for Statistics: Data Processing and Personal Computing." *Journal of Statistical Software*, **11**(1).

[3] Balakrishnan, A. and J. Geunes. 2000. "Requirements Planning with Substitutions: Exploiting Bill-of-Material Flexibility in Production Planning." *Manufacturing & Service Operations Management*, **2**(2).

[4] Ball, M., C. Chen, and Z. Zhao. 2004 "Available to Promise: Systems and Strategies." In Simchi-Levi, D., S. Wu, and Z-J. Shen (eds). *Handbook of Supply Chain Analysis: Modeling in the eBusiness Era*, chapter 15. Kluwer Academic Publishers.

[5] Bar-Yam, Y. 1997. *Dynamics of Complex Systems*. Perseus Books Group.

[6] Belobaba, P. 1989. "Application of a Probabilistic Decision Model to Airline Seat Inventory Control." *Operations Research*, **37** 183-197.

[7] Bertsimas, D. and R. Shioda R. 2003. "Restaurant Revenue Management." *Operations Research*, **51**(3), 472-486.

[8] Bertsimas, D. and S. de Boer. 2005. "Simulation-Based Booking Limits for Airline Revenue Management." *Operations Research*, **53**(1), 90-106.

[9] Bitran, G. and S. Mondschein. 1995. "An Application of Yield Management to the Hotel Industry Considering Multiple Stays." *Operations Research*, **43**, 427-443.

[10] Bitran, G. and S. M. Gilbert. 1996. "Mailing Decisions in the Catalog sales industry." *Management Science*, **42** 1364-1381.

[11] Bitran, G. and S. Monschein. 1997. "Periodic Pricing of Seasonal Product in Retailing." *Management Science*, **43** 427-443.

[12] Bitran, G. and R. Caldentey. 2003. "An Overview of Pricing Models for Revenue Management." *Manufacturing & Service Operations Management*, **5**(20), 203-229.

[13] Bollapragada, S., H.Cheng, M. Phillips, M. Scholes, T. Gibbs, and M. Humphreville. 2002. "NBC's Optimization Systems Increase its Revenues and Productivity." *Interfaces*, **32**(1).

[14] Brooks, F. 1995. *The Mythical Man-Month*. Addison-Wesley Professional.

[15] Buccafurri, F., D. Rosaci, G.M.L. Sarne, and D. Ursino. 2002. "An Agent-Based Hierarchal Clustering Approach for E-Commerce Environments." In K. Bauknecht, A.M. Tjoa, and G. Quichmayr (eds). *EC-Web 2002*. 109-118. Berlin, Germany: Springer.

[16] Chen, C., Z. Zhao., and M. Ball. 2001. "Quantity and Due Date Quoting Available-to-Promise." *Information Systems Frontiers*, **3**, 477-488.

[17] Chen, C., Z. Zhao., and M. Ball. 2002. "A Model for Batch Advanced Available-to-Promise." *Production and Operations Management*, **11**, 424-440.

[18] Cheng, S., E. Leung, K. Lochner, K. OMalley, D. Reeves, L.J. Schvartzman, and M. Wellman. 2003. "Walverine: A Walrasian Trading Agent." *AAMAS '03*, 465-472.

[19] Coulter, K. 1999. "The Application of Airline Yield Management Techniques to a Holiday Shopping Setting." *Journal of Product & Brand Management*, **8**(1), 61-72.

[20] Eguchi, T. and P. Belobaba. 2004. "Modeling and Simulation of Impact of Revenue Management on Japan's Domestic Market." *Journal of Revenue and Pricing Management*, **3**(2), 119-142.

[21] Franklin, S. and A. Graesser. 1996. "Is it an Agent or Just a Program?: A Taxonomy for Autonomous Agents." *Lecture Notes in Computer Science*, **1193**, 21-35.

[22] Galitsky, B. and R. Pampapathi. 2003. "Deductive and Inductive Reasoning for Processing the Claims of Unsatisfied Customers." In P.W.H Chung, C.J. Hinde, and M. Ali (eds). *IEA/AIE 2003*. 21-30. Berlin, Germany: Springer.

[23] Geraghty, M. K. and E. Johnson. 1997. "Revenue Management Saves National Car Rental." *Interfaces*, **27**, 107-127.

[24] Goren, T. and Belobaba, P. 2004. "Revenue Management Performance in a Low-Fare Airline Environment: Insights from the Passenger Origin-Destination Simulator." *Journal of Revenue and Pricing Management*, **3**(3), 215-236.

[25] Healy, K. and L. W. Schruben. 1991. "Retrospective Simulation Response Optimization", *Proceedings of the 1991 Winter Simulation Conference* 901-906.

[26] JADE 3.0b1. http://sharon.cselt.it/projects/jade/. 17 Oct 2003.

[27] Jennings, N., K. Sycara, and M. Wooldridge. 1998. "A Roadmap of Agent Research and Development." *Autonomous Agents and Multi-agent Systems*, **1**, 7-38.

[28] Jennings, N. 1999. "On Agent-Based Software Engineering." *Artificial Intelligence*, **117**(2000), 277-296.

[29] Julka, N., R. Srinivasan, and I. Karimi. 2002. "Agent-based supply chain management - 1: Framework." *Computers and Chemical Engineering*, **26**, 1755-1769.

[30] Kilger, C. and L. Schneeweiss. "Demand Fulfillment and ATP." In Stadtler, H. and C. Kilger (eds). *Supply Chain Management and Advanced Planning: Concepts, Models, Software, and Case Studies*. 135-148. Berlin, Germany: Springer.

[31] Kimes, S.E., R.B. Chase, S. Choi, and E.N. Ngonzi. 1998. "Restaurant Revenue Management." *Cornell Hotel and Restaurant Administration Quarterly*, **40**(3), 40-45.

[32] Kimes, S.E., D.I. Barrash, and J.E. Alexander. 1999. "Developing a Restaurant Revenue-Management Strategy." *Cornell Hotel and Restaurant Administration Quarterly*, **34**(5), 1-30.

[33] Kimes, S.E. 2004. "Revenue Management: Implementation at Chevy's Arrowhead." *Cornell Hotel and Restaurant Administration Quarterly*, **45**(1), 52-67.

[34] Kimes, S.E. and L. Schruben. 2002. "Golf Course Revenue Management: A Study of Tee Time Intervals." *Journal of Revenue and Pricing Management*, **1**(2).

[35] Laird, J.E., A. Newell, and P.S. Rosenbloom. 1987. "SOAR: An Architecture for General Intelligence." *Artificial Intelligence*, **33**, 1-64.

[36] Liu, Y., R. Goodwin, and S. Koenig. 2003. "Risk-Adverse Auction Agents." *AAMAS '03*.

[37] MadKit 3.1b5. http://www.madkit.org/. 17 Oct 2003.

[38] Martin, Robert C. 2003. *Agile Software Development: Principles, Patterns, and Practices*. New Jersey: Pearson Education, Inc.

[39] McGill, J. and van Ryzin, G. "Revenue Management: Research Overview and Prospects." *Transportation Science*, **33**(2).

[40] Morris, J. 2001. *A Simulation-based Approach to Dynamic Pricing*, Master's Thesis, Massachusetts Institute of Technology, MA.

[41] Oliveria, A. 2002. "Simulating Revenue Management in an Airline Market with Demand Segmentation and Strategic Interaction." *Journal of Revenue and Pricing Management*, **1**(4), 301-318.

[42] RePast 2.0.2. http://repast.sourceforge.net/index.html. 17 Oct 2003.

[43] Rothstein, M. 1971. "An Airline Overbooking Model." *Transportation Science*, **5**, 180-192.

[44] Ruiz-Torres, A. J. and K. Nakatani. 1998. "Application of Real-Time Simulation to Assign Due Dates on Logistic-Manufacturing Networks." *Proceedings of the 1998 Winter Simulation Conference*, 1205-1210.

[45] Russel, S. and P. Norvig. 2002. *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall.

[46] Skugge, G. 2003. "Growing Effective Revenue Managers." *Journal of Revenue and Pricing Management*, **3**(1), 49-61.

[47] Smith, B., J. Leimkuhler, and R. Darrow. 1992. "Yield Management at American Airlines." *Interfaces*, **22**(1), 8-31.

[48] Tsay, A., S. Nahmias, and N. Agrawal. 1999. "Modeling Supply Chain Contracts: A Review." *Quantitative Models for Supply Chain Management.*

[49] Umeda, S. and A. Jones. 1998. "An Integrated Test-Bed System for Supply Chain Management." *Proceedings of the 1998 Winter Simulation Conference*, 1377-1385.

[50] van Ryzin, G. and J. McGill. 2000. "Revenue Management without Forecasting or Optimization: An Adaptive Algorithm for Determining Airline Seat Protection Levels." *Management Science*, **46**(6), 760-775.

[51] van Ryzin, G. and G. Vulcano. 2004. "Simulation-Based Optimization of Virtual Nesting Controls for Network Revenue Management." Unpublished.

[52] Weatherford, L. 2002. "Simulated Revenue Impact a new Revenue Management Strategy under the Assumption of Realistic Fare Data." *Journal of Business and Pricing Management*, **1**(1), 35-49.

[53] Weisbuch, G. 2005. "Complex Systems Dynamics and Generic Properties." *Laboratorie de Physique Statistique de l'Ecloe Normale Superieure.*

[54] Wollmer, R. 1992. "An Airline Seat Management Model for a Single Leg Route When Lower Fare Classes Book First." *Operations Research*, **40**(1), 26-37.

[55] Wooldridge, M. and N. R. Jennings. 1998. "Pitfalls of Agent-Oriented Development." *Proceedings of the* 2^{nd} *Int. Conf. on Autonomous Agents*, 385-391. Minneapolis/St. Paul, MN.

[56] You, P. 2001. "Dynamic Rationing Policies for Products with Incremental Upgrading Demands." *European Journal of Operation Research*, **144**, 128-137.

[57] Zhao, Z., M. Ball, and M. Kotake. 2005. "Optimization Based Available-to-Promise With Multi-Stage Resource Availability." ISR Technical Report TR-2005-4.