Booking Simulator Portage to AJAX

Master's degree in Information Technoogy

Sotftware engineering and Information Systems Thesis

Jordi Planadecursach Soler

Directed by: Benoît Eymard

July 31st 2008
Sophia Antipolis, France

PROJECT INFORMATION

Title: Booking Simulator Portage to AJAX
Date: July 31st 2008

Student: Jordi Planadecursach Soler
Grade: Master´s degree in Information Technology, Software engineering and Information Systems
Credits: 30 ECTS
Director: Benoît Eymard
Company: Amadeus SAS (France)

TRIBUNAL MEMBERS

President: ANTONI OLIVÉ RAMON

Vocal: JULITA CORBALAN GONZALEZ

Secretari: ANTONI URPI TUBELLA

QUALIFICATION

Numeric qualification:
Descriptive qualification:

Date:

Acknowledgments

During this project I needed support and help from various sources. I would like to thank the following people:

- My family, for encouraging and ease my way to the university.
- My university friends for helping me going on.
- My project director, Benoît Eymard, for leading me in the project.
- My responsible, Patrick Grandjean, for giving me technical advice in Amadeus tools and in the Simulator.
- My internship mate of the team, Ariel Waserhole, for helping me to understand French people.
- All the members of the Revenue Management Department for their warm welcome, help and advice that gave to me during all the internship period. Specially Nicholas Brenwald, Karim Duval, Clément Seveillac, Emmanuel Grosse and Elie Yared, the new intern.
- My flat mates during the internship, Ignacio Atienza and Francesc Campoy, for they expertise in France and making me go on.
- All the colleagues I have made during the internship for making me disconnect during the weekends. Thanks Jordi Rocamora, Ferran Pons, Marc Kirchner, Ernesto Gonzalez, Fernando Casanova, Alberto Castro, Jaime Sanchez-Laulhe, Jesus Rojo, Alejandro Asencio, Ricardo Sanchez and all the others I forget. I do not forget to mention all the English and Swedish interns and Viktor.
- Thank you Amadeus for offering internships in la Côte d'Azur.

Index

Introduction

Amadeus

History

Founded in 1987 by an alliance between Air France, Lufthansa, Iberia and SAS, Amadeus is an IT group providing technology for the travel industry acting as a Global Distribution System (GDS). As such it links buyers (travel agencies or enduser customers) to sellers (airlines, hotels, car rental companies) who need to exchange travel services.

In the late 60s, the main US airlines started developing and implementing CRSs (Computerized Reservation Systems). Years later, in the mid 70s, they decided to install terminals in travel agencies, giving access only to limited data. But as a result of the pressure of travel agencies and different official organizations, the CRSs started moving towards neutral systems showing all competitors' information. In the mid 80s, US CRSs were increasing in sophistication and they began to look overseas for opportunities to expand.

Meanwhile, in Europe, many of the national airlines had developed their own reservation systems and distribution networks. These, however, only served their respective national markets. Imminent deregulation of the European travel industry made it imperative to create distribution systems able to serve the European and global market.

In 1987, responding to these needs and business opportunities, Air France, Iberia, Lufthansa and SAS pooled their resources in a project called Amadeus. In 1989 Amadeus was the first non-US GDS (Global Distribution System) to a neutral light availability display and became fully operational in 1992.

There are currently four major GDS companies: Amadeus, Galileo, Sabre and Worldspan. Amadeus, which now employs more than 7,600 people around the world, is the youngest of these companies and the current leader on the market with more than 500 million bookings processed annually and a market share of 31%.

Localization

Headquarters, development and operational activities are split among three main locations across Europe, responding to the origin of the companies who mainly founded the company (Air France from France, Iberia from Spain and Lufthansa from Germany).

The main three locations of Amadeus are:

- Headquarters, commercial and marketing: Madrid, Spain
- Product development: Sophia Antipolis, France
- Data processing centre: Erding, Germany

Additional IT services centers are located in London (UK), Miami and Boston (USA) and Sydney (Australia). These ones made according to the new airlines who joined Amadeus, or new companies who became customers of Amadeus.

Amadeus has subscribers in more than 215 markets worldwide, covering the local needs of those different markets with over 70 National Marketing Companies. Over 5,300 people work in the Amadeus worldwide group, which represents 95 different nationalities.

Systems

Today, Amadeus processes over 400 million bookings annually. Through the Amadeus System some 67,000 travel agency locations and some 10,000 airline sales offices around the world are able to make bookings with:

- some 490 airlines, representing more than 95% of the world's scheduled airline seats
- 51,000 hotel properties
- some 45 car rental companies, serving over 29,000 locations
- other travel provider groups (ferry, rail, cruise, insurance companies and tour operators)

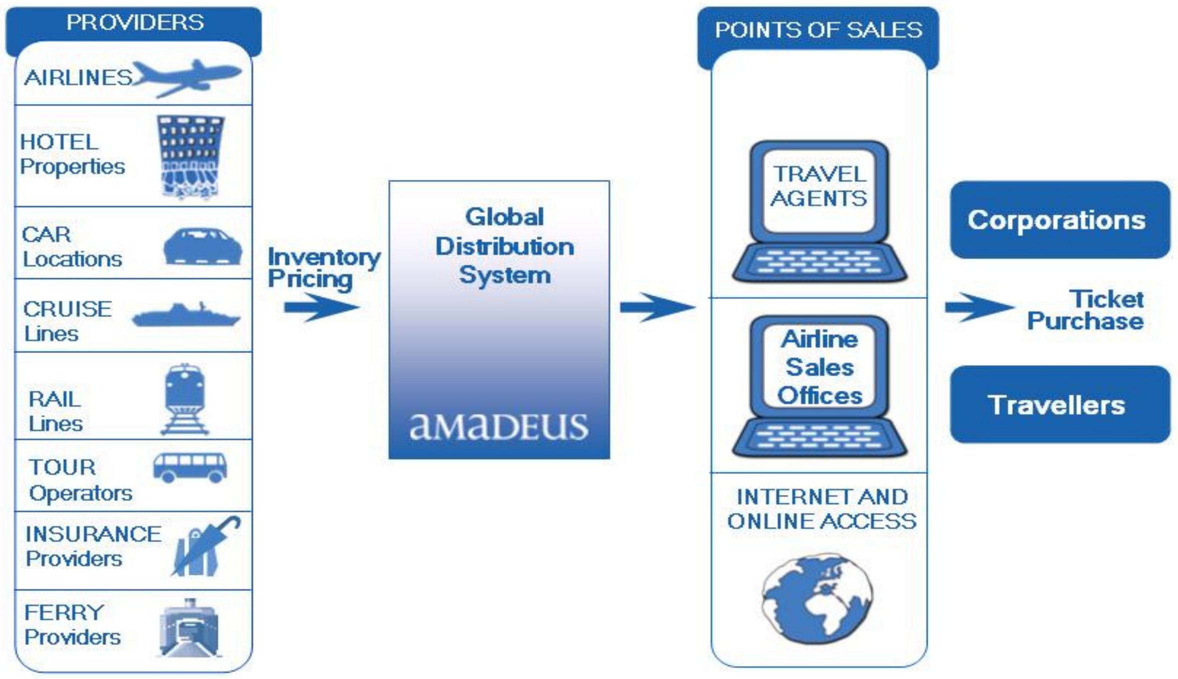


Figure 1: Amadeus is connecting providers with points of sales

The Amadeus data center (in Erding, Germany) is one of the Europe's largest civilian data processing centers. Today, it records data on a total amount of 22 terabyte of data storage disk capacity.

The core of Amadeus' System is clusters of processors running the IBM Real Time Operating System TPF (Transaction Processing Facility). TPF is a specialized operating system, offering very high throughput, very fast response times and very high availability. It is only used by about 50 companies worldwide, most of them in the travel Industry and banking.

- The network is attached to the Front End, which handles all the communication software. The message rate handled by the Front End computer exceeds 20,000 transactions per second.
- The Back End is where the application software runs and to which the main database is attached. More than 70 millions of end-user requests are processed in these every day.

Also, clustered Unix-based systems support sophisticated access to data base systems. For example, one of the UNIX clusters handles a wide range of communication protocols and conversions that are endemic to the travel industry. It is designed to complement the Global Core Front End and allows expanding rapidly market needs in terms of different communication protocols. New applications developed by Amadeus are based on the Linux operating system and on C++ as developing language.

Currently, the company is in a process of migration from de IBM-TPF Back End to the "Open Back Ends" with Unix-SUSE management system, Oracle Database core and Open source applications, as far as possible.

Amadeus solutions for Airlines

Amadeus IT solutions for airlines are grouped under the name Altéa. Altéa Reservation, Altéa Inventory and Altéa Departure Control System are the main products of this offer.

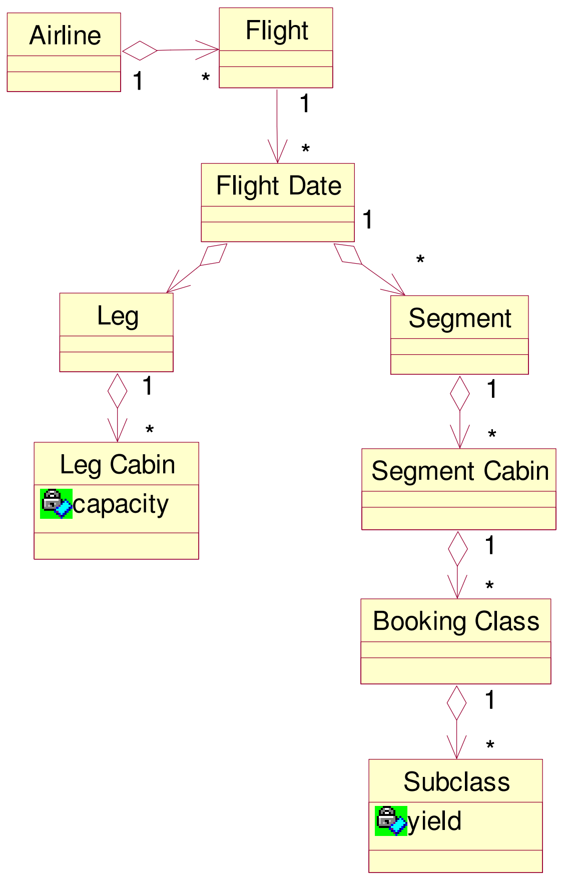
- Altéa Reservation deals in particular with customer profiles, bookings management and ticketing.
- Altéa Departure Control System allows the automation of all processes related to an airline's airport management operations, such as the check-in of passengers and baggage, the boarding control, the management of the flight before departure (calculation of the aircraft load, of its center of gravity…).
- Altéa Inventory system manages the stock of seats provided by a given airline. Given the bookings and already made and the inventory controls such as *maximum booking limits* imposed on each fare class (maximum number of bookings which can be accepted in each fare class), the inventory derives an availability, that means it decides to accept or not a booking request made by a customer through the GDS.

Inventory systems were previously managed by the airlines themselves. The Amadeus Altéa Plan actually aims to integrate the inventory system of airlines to the Amadeus Inventory System (Altéa Inventory).

Amadeus
Altéa Inventory

Airline business general knowledge

The Airlines structure is complex. In order to explain the main elements the figure below give a picture of a simplified structure.

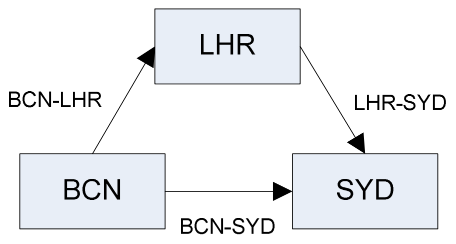


The *Airline* is the company that manages the planes. An A*irline* is identified by a two letters airline code. For instance British Airways is BA and Lufthansa is LH.

Each *Airline* has different flights. A *Flight* is a route between two airports. It has a board point and an off point. A *Flight* is identified by the airline code plus the flight number. For instance BA134 is the flight 134 of British Airways. The airport code is a 3 letters identifier. For instance Barcelona is BCN and Nice is NCE.

The *Flight* can be operated on different dates. This is called *Flight Date*. For instance BA134 at September 4th 2008 is a *Flight Date*.

There are some flight-dates that have several stops. A plane that goes from London to Sydney has to stop in Bangkok to fill up the fuel tanks. Some times there is not a direct plane from the origin to the destination and a change of plane is needed. A direct journey from one point to another point is called a Leg. A Flight Date consists on one or more Legs. On the other hand there is the Segment. A Segment is the commercial name for a journey from one point to another. It can consist of one or more legs. The following figure describes the differences between a Leg and a Segment of a Flight that goes from Barcelona to Sydney stopping in London.



The *Flight* consists of two legs BCN-LHR and LHR-SYD. Nevertheless there are three segments: BCN-LHR, LHR-SYD and BCN-SYD. The *Segment* is the object to be purchased. The *Leg* is how the route is operated.

Each Leg has an aircraft assigned. Normally there is the same aircraft for each Leg. An aircraft has different compartments called Cabins: For instance First Class, Business or Economy. These various travel cabins correspond to different levels of service. Higher travel cabins are more comfortable and more expensive. In practice in the aircraft, the division is often marked with a curtain. Each *Leg Cabin* has an availability that is the number of seats available.

The *Segment Cabin* is the equivalent to the Leg Cabin but seen as a commercial product. The availability of each Segment Cabin is not easily calculated from the Leg Cabins. There are different algorithms for calculating the availabilities.

Moreover, the various cabins are always divided into several fare classes. Two people sitting next to each other in the same cabin (and receiving the same service) may have paid different prices. The fare may reflect restrictions on the ticket (refundable ticket, partly refundable ticket, non-changeable ticket, etc). Fare classes may also vary according to how far ahead the ticket was purchased, or how long the stay at the destination is.

Each *Class* is divided into several subclasses. Each *Subclass* has its own fare. The difference between the subclasses that belong to a class is the point of sale. The point of sale is from where the ticket has been purchased. A booking in the same class may have different prices if the booking is done from Europe or from Australia.

The *Revenue Analyst* is in charge of analysing departed flights and creating the pricing rules for the future flights. The revenue analyst uses a Revenue Management System (RMS) to help them set an optimal configuration in order to get the maximal revenue for the airline.

Revenue Management System

Revenue management is an economic practice used by a number of industries whose purpose is to predict the demand the best way possible in order to adapt their offer to maximize their revenue.

Revenue Management aims thus to forecast consumer behavior at the microeconomic level and to optimize the product segmentation and price, in order to maximize revenue.

Today more and more hotels and rental cars companies are adopting Revenue Management techniques. Revenue Management (RM) has become a strategic tool for many companies. Some studies carried out by the Massachusetts Institute of Technology have thus credited RM practices for 2% to 5% of airlines global revenues.

An RMS allows its users to re-calculate automatically and quickly the forecasts and recommendations for their fleet at regular frequencies but also enables manual interventions on top of those automatically computed recommendations in order to allow the integration of the airline revenue analysts' knowledge.

Revenue Management Systems are mainly composed of two modules: the forecaster and the optimizer.

The forecaster is in charge of estimating:

- The remaining demand for all the Flight-Dates departing in the future. This is the main input provided to the Optimizer for seat allocation.
- Cancellations and no-show rates for all the flight-dates departing in the future.

The optimizer is in charge of computing recommendations, namely:

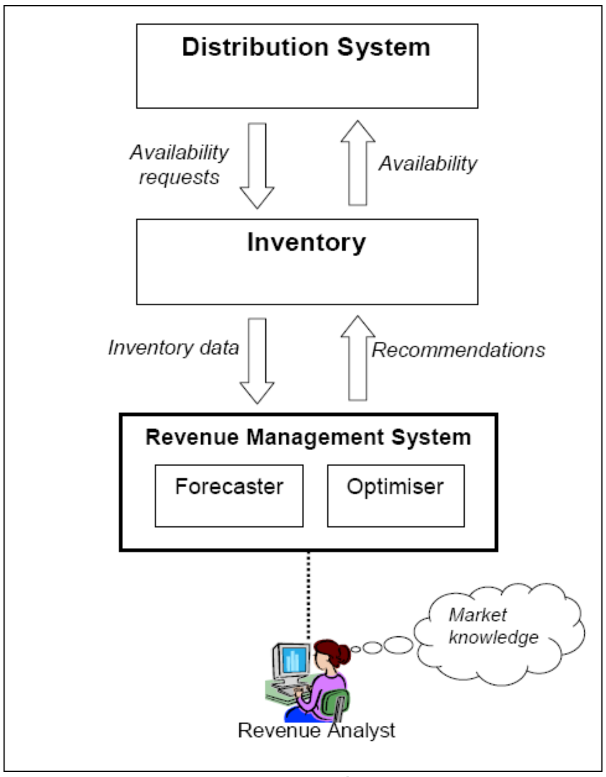
- Overbooking, i.e. the calculation of the optimal offered capacity.
- Optimal seat allocation.

RMS department

Amadeus does not for the moment provide a Revenue Management System to airlines. Airlines use their own Revenue Management System or the RMS of other IT solutions providers such as PROS, Air Max or Lufthansa Systems.

However Amadeus has lots of strategic tools in the field of Revenue Management: it already provides a successful Inventory System and it is a provider of key data such as the real-time evolution of the number of bookings and cancellations made in each fare-class for each flight, the number of no-shows (passengers who have booked a seat but do not show up at departure) which are key inputs for the Revenue Management System. It also has access to the precious PNR (Passenger Name Record) data. PNR includes all the details about a given booking (passenger personal information, details of its trip and its preferences) and are considered as the future of RMS input data. Those are the reasons why Amadeus has decided to invest in the field of Revenue Management.

Since 2005, a team in Amadeus (DWS, Data Workforce Solution) has been working on the subject. First integrated in the department in charge of the Inventory System, it has then been incorporated in a new department, called RMS (Revenue Management System). This new department has 2 teams. One in charge of the forecaster (FCT) and the other in charge of the optimizer (OPT). Currently 7 engineers are working on each team. I have been integrated into the FCT team for my end-of-studies internship.



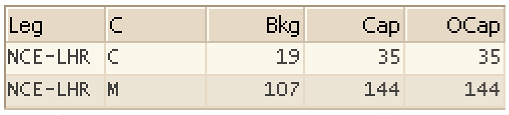
Interactions between RMS and the Inventory

Inventory

An Inventory System, like Altéa Inventory, manages the stock of seats offered by an airline on all of its flights, and on selected flights operated by its commercial partners. This stock of seats is associated with physical capacities on individual flight departures (flight legs), whereas passengers request seat availability for an itinerary of one or more flight segments and pay a price corresponding to both their itinerary and the fare class they have chosen. *Inventory Controls* allow the airline to adjust how seats are made available to its customers, so as to maximize its revenues and achieve marketing objectives.

Inventory Controls are typically generated by an airline's own Revenue Management System (RMS), based on analysis of historical booking data, forecasts of booking demands for future flight departures and optimization of revenues associated with each booking and/or seat sold. Given a set of Inventory Controls, there are many ways, called "availability algorithms", to perform the availability calculation. Typically, an airline will apply a few well-known and consistent availability calculation algorithms for its whole flight network and all the types of request it receives. Based on statistical no-show data, airlines overbook flight capacity in order to account for cancellations occurring during the booking process and no-shows at the airport.

Altéa Inventory GUI is the piece of software in charge of managing the inventory. The GUI is connected to the inventory through the network using EDIFACT protocol. This will be explained in detail later.



Leg	C	Bkg	Cap	OCap
NCE-LHR	C	19	35	35
NCE-LHR	M	107	144	144

A Leg Cabin as displayed in Altéa Inventory.

Leg: Leg identifier (Board Point and Off Point)
Cap: Capacity (saleable capacity)
OCap: Operational capacity (real capacity)

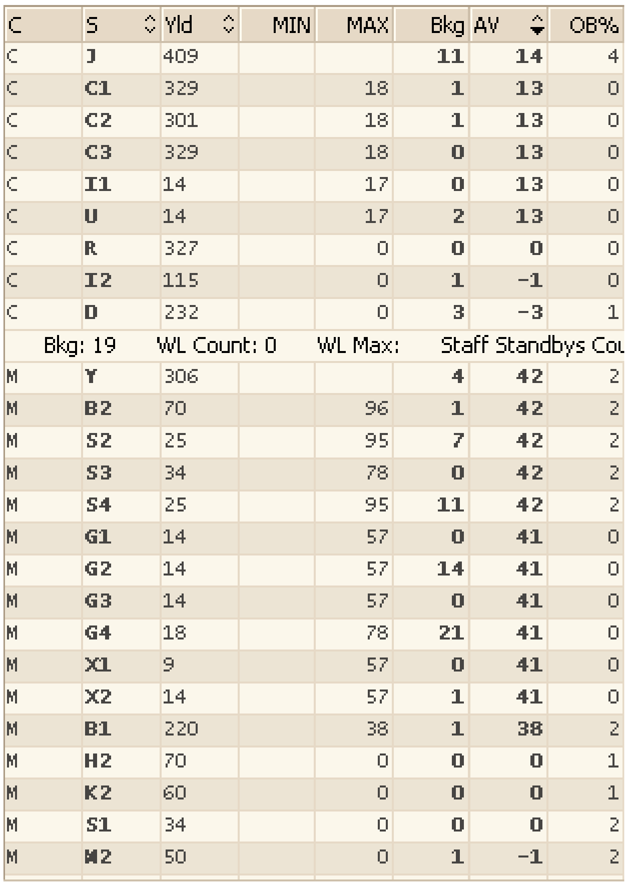
A *Flight Structure* is how the classes and subclasses of a flight are distributed. Each airline has different ways for defining the flight structure. Some classes are nested inside others. This means that if one booking is done in a nested class, the availability will be decreased in both the nested and the container class. Different nesting policies exist.

Given a flight structure the revenue manager can set different parameters. The two relevant parameters in the internship are the booking maximums and minimums. Other parameters such as the overbooking percentage or the yield can be set as well. Normally these parameters are set by the RMS when the flight is optimized.

A flight is created 1 year before its departure. During this year bookings and cancellations can be performed. Not all the classes are available during the flight's life. Some classes are available just in the last minute or some others just during the first days. Depending of the number of bookings, a class may be opened or closed. No bookings can be done when a class is closed.

The maximum booking limit sets the maximum number of bookings that can be done in a certain class. The sum of all maximums can not be larger than the cabin capacity. When the number of bookings equals the maximum limit, the class closes. To shut down a class the maximum limit is set to 0 (see classes H2, K2, S1 and M2 in the figure).

The minimum limit sets the number of seats which must be protected for this class from all the other classes. If a class C has a minimum of 3, and 1 booking has been made in it, 2 seats are protected from all the other classes.



C	S	Yld	MIN	MAX	Bkg	AV	OB%
C	J	409			11	14	4
C	C1	329		18	1	13	0
C	C2	301		18	1	13	0
C	C3	329		18	0	13	0
C	I1	14		17	0	13	0
C	U	14		17	2	13	0
C	R	327		0	0	0	0
C	I2	115		0	1	-1	0
C	D	232		0	3	-3	1
	Bkg: 19	WL Count: 0		WL Max:	Staff Standbys Cou		
M	Y	306			4	42	2
M	B2	70		96	1	42	2
M	S2	25		95	7	42	2
M	S3	34		78	0	42	2
M	S4	25		95	11	42	2
M	G1	14		57	0	41	0
M	G2	14		57	14	41	0
M	G3	14		57	0	41	0
M	G4	18		78	21	41	0
M	X1	9		57	0	41	0
M	X2	14		57	1	41	0
M	B1	220		38	1	38	2
M	H2	70		0	0	0	1
M	K2	60		0	0	0	1
M	S1	34		0	0	0	2
M	M2	50		0	1	-1	2

Simplified view of the Altea Invenotry GUI Flight Structure and controls view

C: Cabin identifier.
S: Class and subclass identifier.
Yld: Price for the subclass.
MIN: Minimum booking limit.
MAX: Maximal booking limit.
Bkg: Booking counter.
Av: Availability.
OB%: Allowed overbooking percentage.

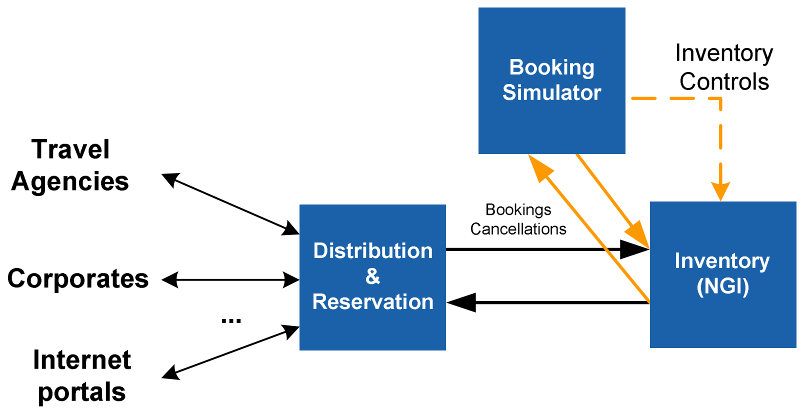
Booking simulator

Overview

The booking simulator aims to see how the choice of inventory controls (in particular booking limits imposed on booking classes) influences the total revenue of a flight.

To achieve this goal, the booking simulator:

- Simulates demand for a given flight using historical demand curves
- Submits this demand (booking requests and cancellations) and the booking limits chosen by the user to the inventory system. The inventory decides whether to accept or refuse the booking requests (depending on the booking limits imposed).
- Receives the answer of the inventory and calculates the total revenue according to the number of bookings accepted.



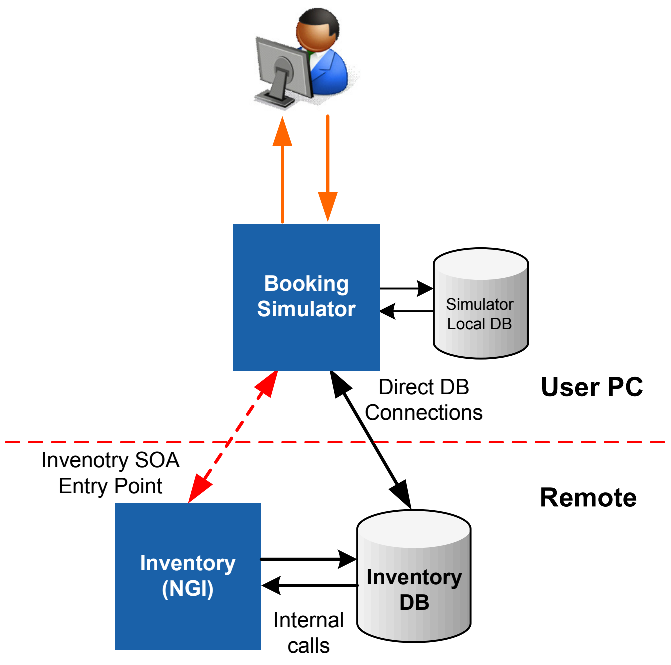
Interactions between the Simulator and the Inventory

Existing simulator

The booking simulator is the result of an internship which took place in 2004. It has been developed by Nicholas Brenwald, who is now working in the FCT team. Many interns have since worked on it adding new functionalities and improving the code.

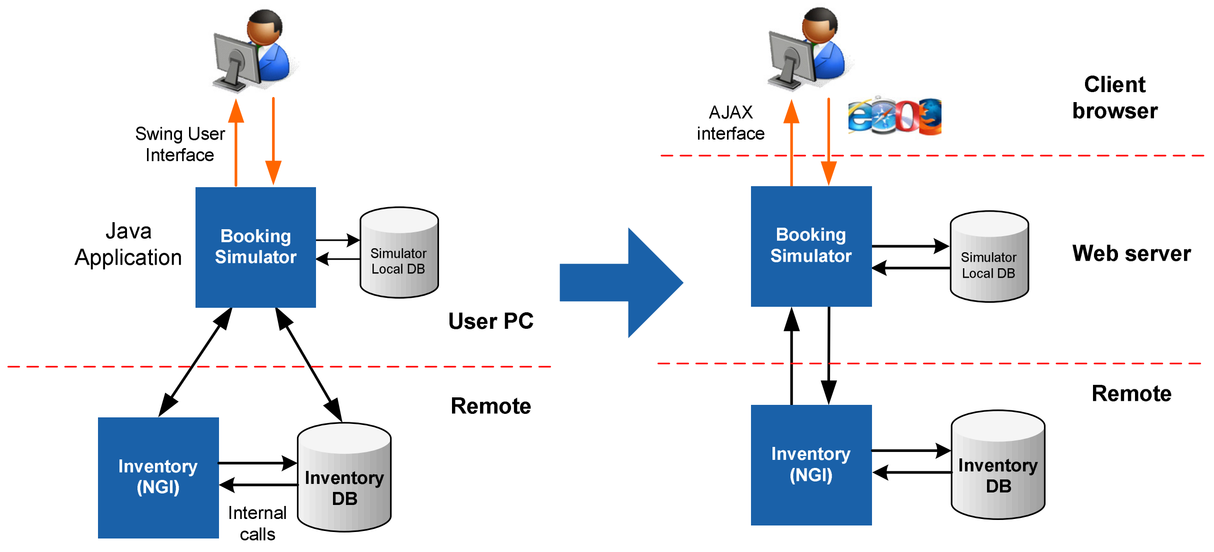
Cedric Baxa optimized the simulation by creating a batch framework that was ran in the inventory side. This was done in order to increase the performance of the simulation response time.

Valerie Seguin made a study of overbooking techniques. She implemented an overbooking module inside the simulator. She also added the functionality to send cancellations to the inventory. As the internship of Valerie was not focused on the simulator, just in the overbooking techniques study, she did not have time to implement all the functionalities.



Internship Objectives

The main objective of the internship is to port the existing simulator user interface from Java Swing to Web 2.0 technology.



The existing simulator is basically an application coded in Java. However, some functionality was implemented by PL/SQL scripts. This standalone application communicates with the user through a Java Swing frontend. In order to work properly, the simulator needs a local database to store some data. More technical details are going to be described later.

The problem with the original simulator is the time one has to take in order to make a first simulation in a fresh environment. There isn't any deployment procedure established as the simulator was an internship project.

To run it, it was necessary to download the source from a CVS repository and compile it. A MySQL database server had to be installed onto the users PC. It had to be filled manually with data extracted by an SQL script from a 3rd database. Once everything was ready one could start to simulate.

With the new AJAX version the usability increases considerably. A single thin browser is the only tool needed in order to perform simulations. Moreover, the historical data of the already departed flights can be shared. A single local database for each client is no longer needed.

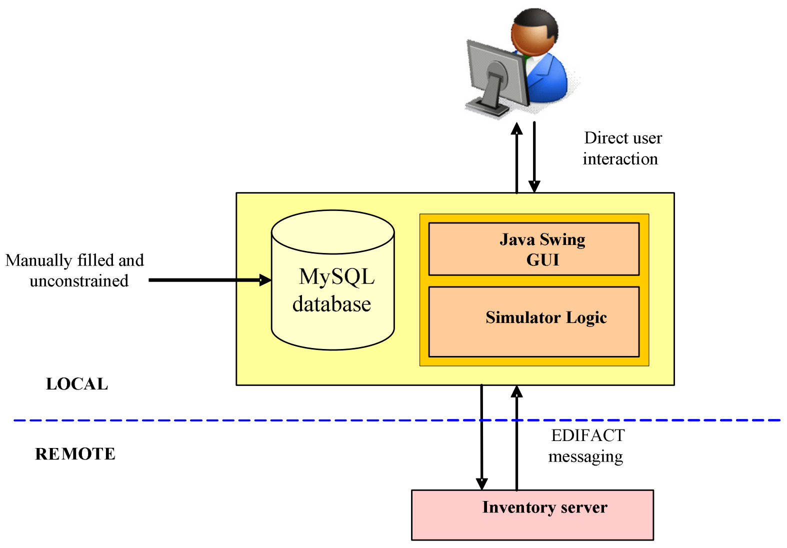
In order to achieve the goals, a migration plan was created. Before establishing the plan milestones, the original simulator had to be studied deeply to enable me to make a good estimation.

Existing simulator study

External Interactions

The structure of the simulator was described in documentation as follows (later on, we will see that it was not really like this). The Java application has 3 different interactions.

- On one side there is a Java GUI Frontend. This interface permits the interaction of the end user with the simulator.
- In the same enviornment of the main application there is a MySQL database. This is accessed through JDBC.
- The component object to study, the inventory, is acceccced through its main interface. This SOA interface is implemented with EDIFACT messaging. There is an EDIFACT message for each service.



- Apart from the Inventory connection through EDIFACT, there is also a direct connection to the inventory's Oracle database. There are some functionalities that were hard to perform using EDIFACT messages. To perform them, the database was targeted directly. Later, it will be explained why this connection was no longer useful and how we got rid of it.

EDIFACT

EDIFACT stands for Electronic Data Interchange For Administration, Commerce, and Transport. UN/EDIFACT is the international EDI standard developed under the United Nations. EDIFACT has been adopted by the International Organization for Standardization (ISO) as the ISO standard ISO 9735.

The EDIFACT standard provides

- A set of syntax rules to structure data,
- An interactive exchange protocol (I-EDI),
- Standard messages that allow multi-country and multi-industry exchange.

EDIFACT has a hierarchal structure where the top level is referred to as an interchange, and lower levels contain multiple messages which consist of segments, in turn consisting of composites. The final iteration is an element which is derived from the United Nations Trade Data Element Directory (UNTDED) and is normalized throughout the EDIFACT standard.

A group or segment can be mandatory (M) or conditional (C) and can be specified as being repetitive. For example, C99 indicates between 0 and 99 repetitions of a segment or group, while M99 signifies between 1 and 99 repetitions.

EDIFACT is used by Amadeus for intra-back-end communication and for GUI-to-server communication. In this particular project EDIFACT was used between the simulator and the inventory server.

When a simulator is run it starts by sending some bookings, cancellations and configuration messages to the inventory.

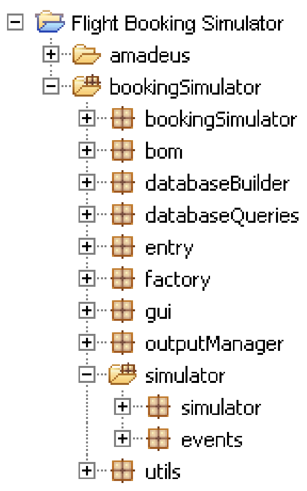
```
UNB+IATB:1+1ANGUD+1ASIGUD+080703:0943+682T4M8YQR0049+00BH83JB750048++E'
UNH+1+IEOTUQ:02:1:1A+GURQY8M4T20049'ORG+1A+:NCE1A0955++++FR:EUR:FR+A043
9PGSU'TVL+030708::030708::0+NCE+LHR+BA+341:G+++P'RPI+1+SS'STX+SEL'IRV++
++::DID:26281'DUM'UNT+8+1'UNZ+1+682T4M8YQR0049'
```

The message above corresponds to a booking of a seat in an ariplane. The name of the message is IEOTUQ (in red). Its version is 02 revision 1. The booked seat is marked in blue. Flight BA 341, segment Nice – London Heathrow, on July 3th of 2008 at cabin G class P. In orange SEL denotes a booking. CAN would denote a cancellation.

In order to generate and parse these messages each of them has an assigned grammar. The grammar varies for each message and version. The same message does not assure compatibility between versions.

Amadeus has a big database of all the messages it generates, receives or cooperates with. There is a specific tool, Visual Edifact, used for accessing this database.

Code analysis

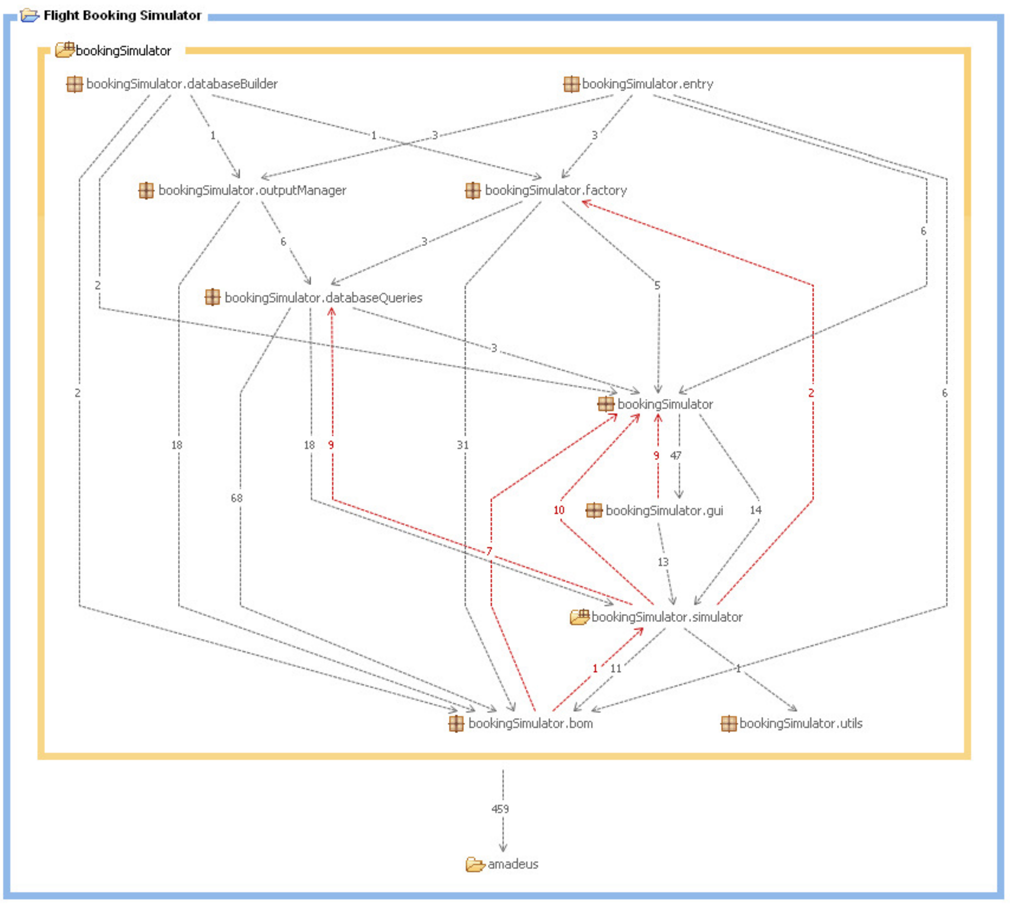


In order to detect the components that the simulator consists of, a source code analysis was performed. A static code analyzer was used. It gives the developer an idea of how the code is structured (layers, patterns, 3rd party components, etc).

Taking a look to the package tree the application seems well divided and structured.

Using STAN, a code analyzer, the dependency map below was drawn. Each connection between two nodes represents a call to a function. The number in each link represents the number of calls.

The red connections represent the couplings. They should not be present in a well designed code. At the moment there are 6 couplings with 38 methods involved.
According to the results, the application did not seem to follow any design model. Without a clear design model, it would not have been possible to cleanly extract the GUI and replace it for another implementation. This was the first milestone to be achieved.



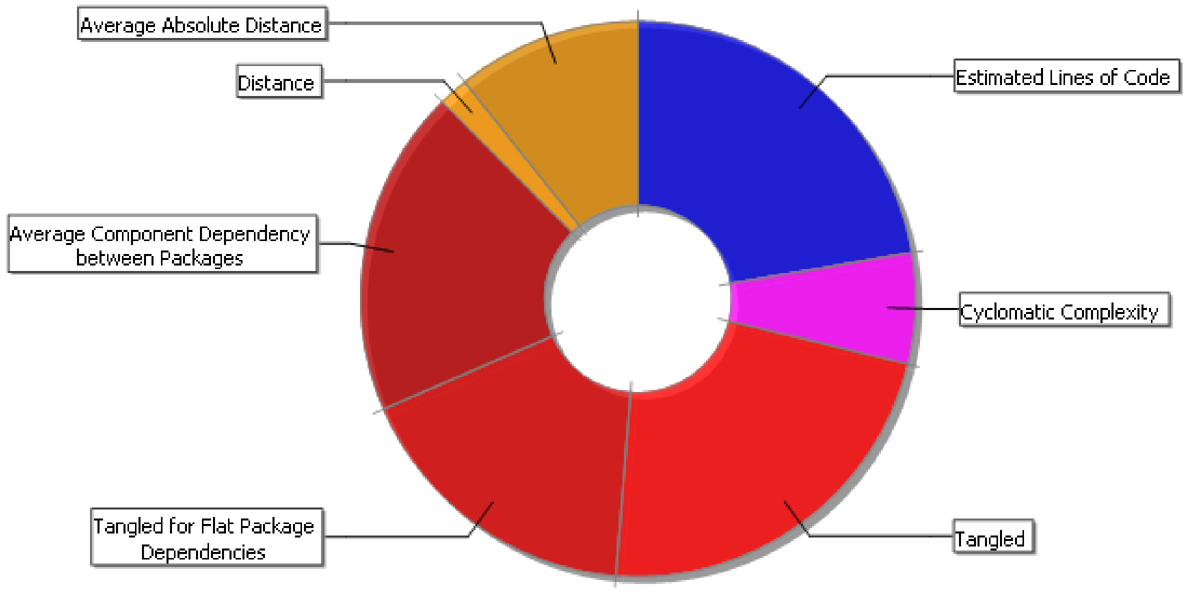
Code Pollution Analysis

Using the same piece of software a pollution diagram was generated. It gives an overview of the packages, classes, and code status. The code defects are painted in warm colours, such as red. The better the code, the colder the colour. Good code is one which has a blue and green chart only.

As we can see in the pie chart, the main problem was *tangling*. Tangling is produced by coupling between classes.

The second main problem is the *flat package dependency*. The *average component dependency between packages* is related to it as well. That is due to the package grouping of classes without a previous design. The packages do not follow a hierarchy and, as we can see in the diagram above there are many calls from one to another.

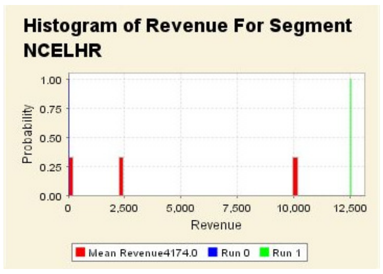
Another problem is the average absolute distance and the lack of response for a class. That denotes a bad organization of methods among classes. Too many different class method calls are needed to resolve a single functionality.

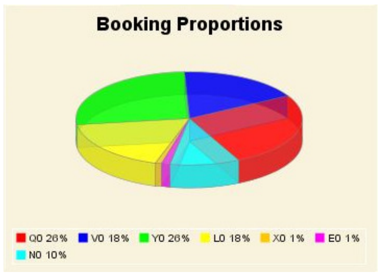


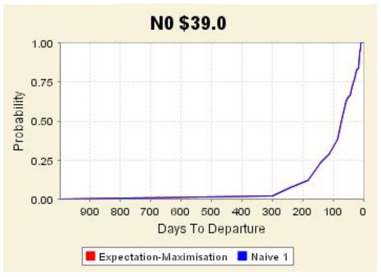
Functionalities

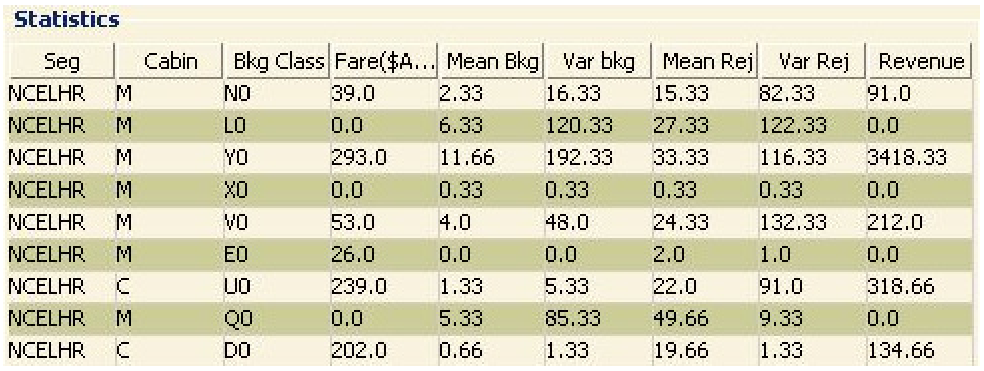
After running some simulations and making some tests, we discovered that there were some functionalities missing. The inventory max and min controls were not properly applied to a flight during its life. This was documented in the last intern's documentation, so, the fix/addition of this functionality became a requirement.

Apart from this, the inventory server seemed to respond to the bookings and cancellations and the frontend presented some charts with expected revenues.





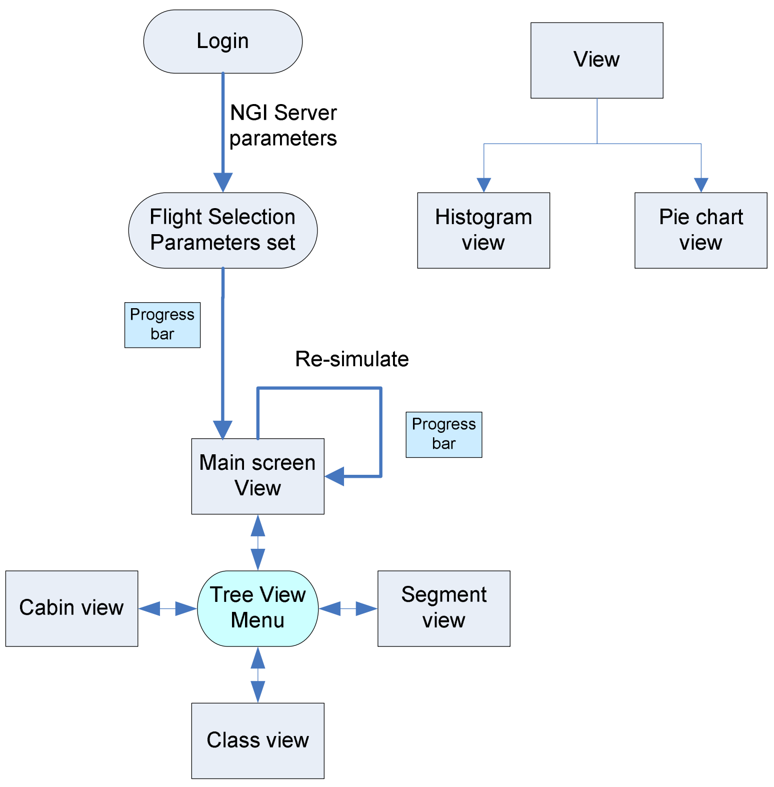




Statistics

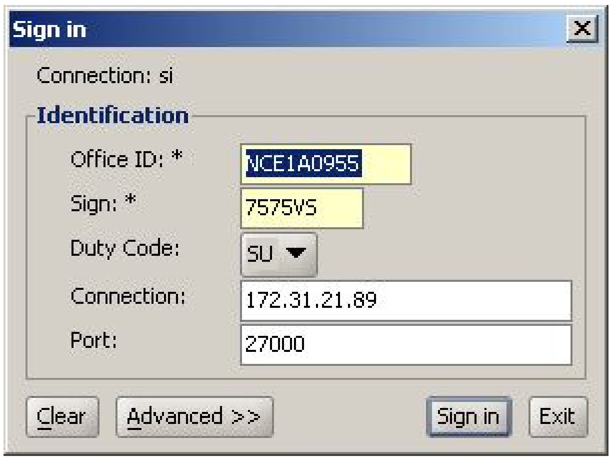
Seg	Cabin	Bkg Class	Fare($A…	Mean Bkg	Var bkg	Mean Rej	Var Rej	Revenue
NCELHR	M	N0	39.0	2.33	16.33	15.33	82.33	91.0
NCELHR	M	L0	0.0	6.33	120.33	27.33	122.33	0.0
NCELHR	M	Y0	293.0	11.66	192.33	33.33	116.33	3418.33
NCELHR	M	X0	0.0	0.33	0.33	0.33	0.33	0.0
NCELHR	M	V0	53.0	4.0	48.0	24.33	132.33	212.0
NCELHR	M	E0	26.0	0.0	0.0	2.0	1.0	0.0
NCELHR	C	U0	239.0	1.33	5.33	22.0	91.0	318.66
NCELHR	M	Q0	0.0	5.33	85.33	49.66	9.33	0.0
NCELHR	C	D0	202.0	0.66	1.33	19.66	1.33	134.66

Current navigational diagram



Windows details

Login window



Connection: si

Office ID: * NCE1A0955

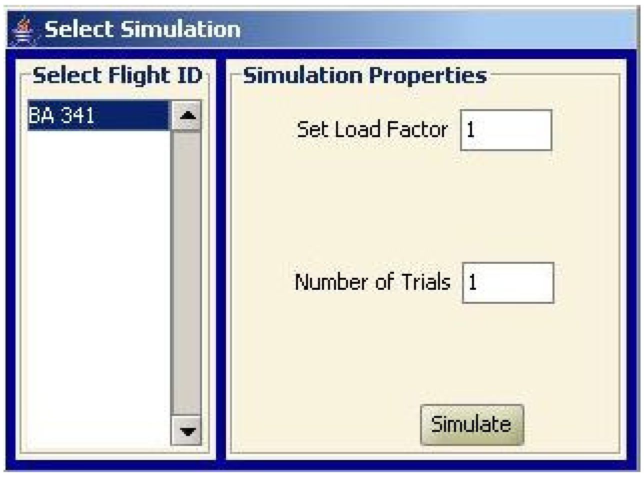
Sign: * 7575VS

Duty Code: SU

Connection: 172.31.21.89

Port: 27000

Flight Selection

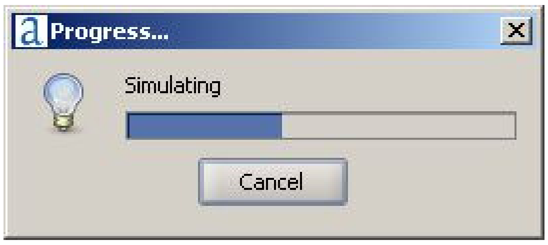


Select Flight ID: BA 341

Set Load Factor: 1

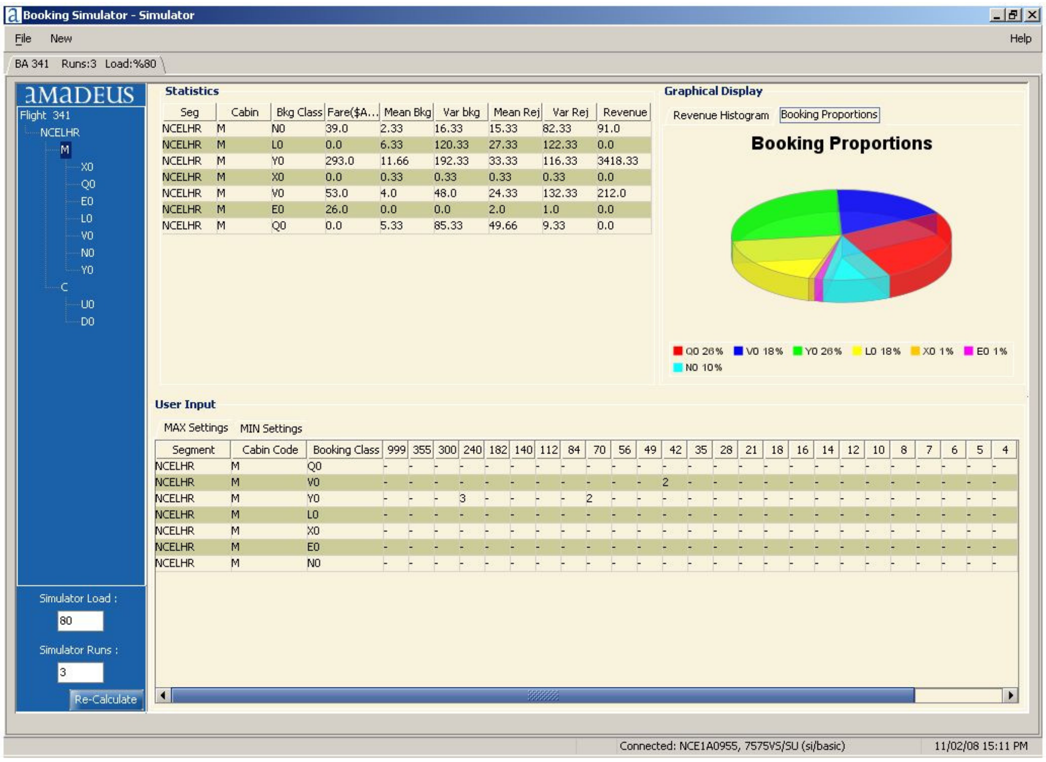
Number of Trials: 1

Progress bar



Simulating

Main view



BA 341 Runs:3 Load:%80

Statistics

Seg	Cabin	Bkg Class	Fare($A...	Mean Bkg	Var bkg	Mean Rej	Var Rej	Revenue
NCELHR	M	N0	39.0	2.33	16.33	15.33	82.33	91.0
NCELHR	M	L0	0.0	6.33	120.33	27.33	122.33	0.0
NCELHR	M	Y0	293.0	11.66	192.33	33.33	116.33	3418.33
NCELHR	M	X0	0.0	0.33	0.33	0.33	0.33	0.0
NCELHR	M	V0	53.0	4.0	48.0	24.33	132.33	212.0
NCELHR	M	E0	26.0	0.0	0.0	2.0	1.0	0.0
NCELHR	M	Q0	0.0	5.33	85.33	49.66	9.33	0.0

🟥 Q0 26% 🟦 V0 18% 🟩 Y0 26% 🟨 L0 18% 🟧 X0 1% 🟪 E0 1% 🟦 N0 10%

Segment	Cabin Code	Booking Class	999	355	300	240	182	140	112	84	70	56	49	42	35	28	21	18	16	14	12	10	8	7	6	5	4
NCELHR	M	Q0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	V0	-	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	Y0	-	-	-	3	-	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	L0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	X0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	E0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	N0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Simulator Load : 80

Simulator Runs : 3

Connected: NCE1A0955, 7575VS/SU (si/basic)   11/02/08 15:11 PM

Components

Basically the components of the simulator can be explained in the figure below. The GUI part is composed of java swing components attached to the Amadeus framework frontend. The data access layer contains connections to the relational databases and to the inventory. The business logic connects both parts and is in charge of performing the simulation logic.

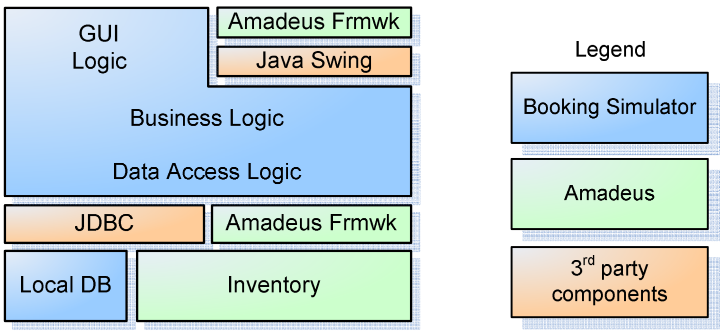


Figure X

Amadeus Framework

It is a java middleware component that deals with the presentation layer and with the data access layer. It allows the programmer to concentrate on the business logic (the important part of the application).

The Amadeus Framework handles authentication by providing an advanced login screen. The programmer has to implement its own desktop according to its needs.

As the user has to authenticate in order to connect to the desktop, the Amadeus Framework creates a connection object called a conversation. This conversation can be used by the programmer to send and receive message from the inventory.

JDBC

The database access part of the simulator consists of two parts: one that connects to the local MySQL database and another that connects to the inventory database.

The first uses the MySQL JDBC driver. The only drawback is that all the queries have to be isolated in the data layer. In the current code the GUI components access the database directly.

The second uses the Oracle JDBC driver. This dependency had to be completely eliminated. The inventory had to be accessed though the EDIFACT interface as the database structure can change without previously warning and could cause critical errors that would leave the simulator useless.

Simulator Logic

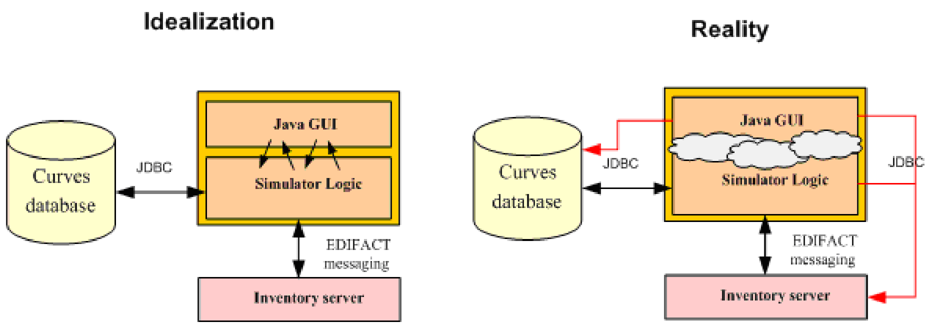
The simulator logic is the tangled part. It consists of Swing forms, SQL queries, simulation algorithms, statistical distributions and EDIFACT messages.

It is possible to distinguish the business core from the tangled simulator logic. It is the part that is purely simulation logic.

Conclusions

The conditions of the existing simulator are not the optimal to perform a clean GUI migration. The following problems were identified:

- The code does not follow any architecture or design pattern.
- The direct access to the Inventory should not be there.
- The calls from the presentation to the local DB should not be there.
- The inventory controls modification is not implemented
- There is not a centralized place where to configure the simulator
- There is not a logging framework



All these ments give a list of tasks that should, or could be done to ease the migration of the presentation layer.

Project planning

Main tasks

Once the objectives of the internship were clearly defined, a migration plan had to be drawn. This plan was very similar to any software engineering project plan with some differences. The requirements analysis is not so important in a frontend migration because the requirements were already defined in the previous simulator.

The timeline of the project was restricted to the length of the internship: 6 months. One person, me, will be working on the project. Moreover I was assigned a tutor, Patrick, that gave me technical advice on Amadeus middleware and other, more general aspects of the software development. I was also assigned a team leader that tracked my evolution on the project, Benoit. Periodically I had to inform him, the manager of the department about the advances of the project.

In France the number of weekly working hours varies depending on the contract. Amadeus employees work 37.5 hours per week. From February 4th until July 31st there has been 26 weeks. In this period there have been 5 holidays so in total the number of hours planned to accomplish the project is 940.

The main project milestones defined have been:

- Existing simulator study
- Development plan definition
- Development
- Testing
- Packaging Deployment
- Documentation

It was hard to say how much time these tasks would last. In order to concretize and make a time-estimation some tasks had to be done beforehand. The existing simulator was studied during 2-3 weeks and a requirement document was written. The existing simulator study is the one that was shown in the previous chapter. The requirements are presented in the next section.

Requirements

Front-End Portability

This was the main task to be performed. In order to cleanly migrate the frontend, a simulator redesign was required. To carry out the portability several tasks had to be planned. Before starting the developing process, some architectural decisions had to be taken.

- Suitable web server.
- Web developing framework or library.
- Chart generation libraries

Those technologies depend on the original simulator and may have some dependencies themselves. For example, the use of one technology for the web server could avoid using a developing framework. That will be studied and discussed further on.

Simulator Logic redesign

The simulator logic had to be redesigned before the frontend migration. This task was important to avoid errors. It was also good to follow the MVC (model, view, controller) pattern for portability, extensibility, robustness, debugging and other software properties.

The main tasks are the following:

- Apply architecture and design patterns
- Get rid of the inventory Oracle DB connection.
- Remove GUI direct calls to de local database.
- Refactor source code and create a configuration framework
- Creating a logging framework

Inventory Controls feature

The inventory controls were not sent to the inventory whilst running the original simulator. They had to be sent manually and can't change during the flight life. Implementing this functionality will make the simulator functional again.

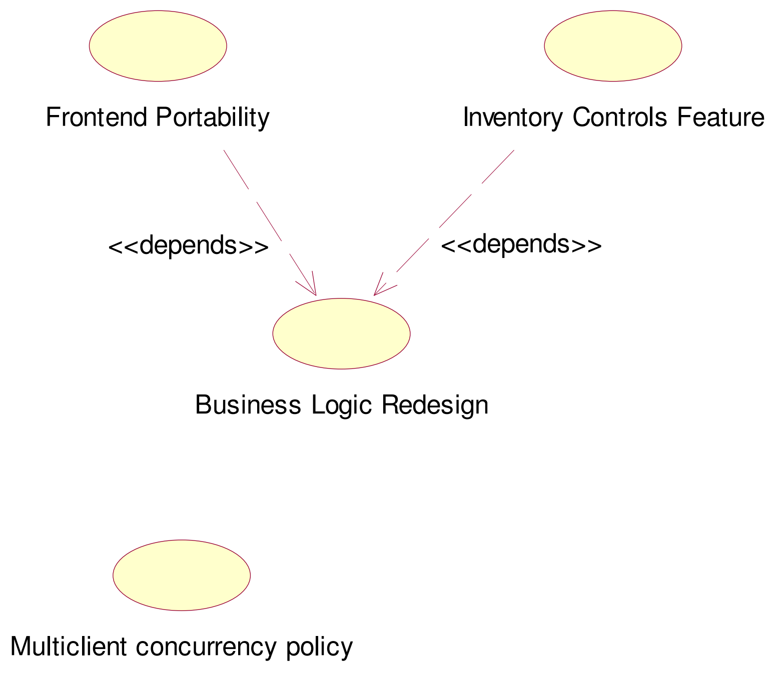
Multi-client concurrency policy

The web-server application will allow several users to interact with the simulator. This can lead to some concurrency errors because just one simulator can be run at the same time. This problem arises from the NGI server dependency. The original simulator actually changes the system date of the server. If many clients run parallel simulations, this could possibly lead to interference between those simulations. The original simulator did not support various simulations at the same time. That could be solved by locking the queries when a simulation is in process or developing some kind of queuing mechanism.

That was not the main point of the internship so that task became optional. In order to prevent errors a default locking policy was developed.

Requirement dependencies

Both the frontend portability and the control features depend on the business logic redesign.



The frontend migration could have been carried out without the redesign, but the simulator would not have been very extendible, scalable or reusable. There would have been serious problems for debugging and for adding or modifying functionalities.

The addition of the Inventory Controls feature could have been implemented before the redesign. We discarded this option because after implementing it, it would have had to have been adapted to the new design.

Timing

In the project there are basically 6 main steps; each of them can be divided into smaller, more manageable tasks:

- Simulator study
  - Rerun the old simuator: Check-out source code from CVS repository. Build Simulator from source code. Configure simulator and connection to Inventory. Install a mySQL instance. Fill the database with historical data of departed flights extracted from a production environment.
  - Study the functionallities of it and its internal design. Manual study and perform static source code analysis.

- Make a development plan
  - Requirements document elaboration and timing
  - Web framework study: Choose a technology to implement the presentation layer.
  - Design and architecture.
  - Preliminary visual design. High level graphical design of the screens.

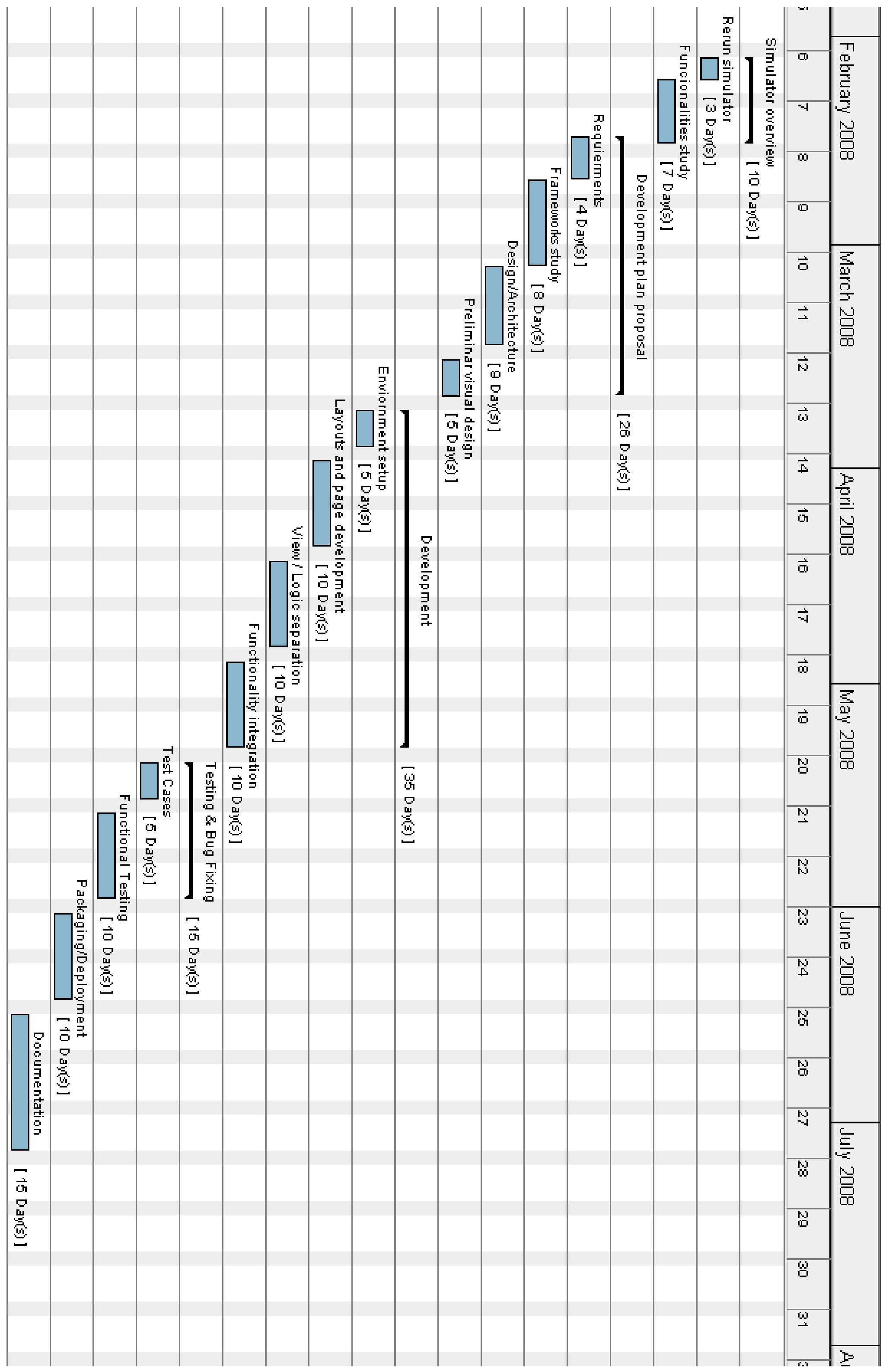
- Development process
  - Environment setup: Prepare the development tools as well as the development enviornment (servers and databases).
  - Layout and page development: code the presentation layer with its interaction logic.
  - Redesign and refactor.
  - Functionallity integration: connect the presentation with the business layer. Integrate the missing features identified during the simulator study.

- Testing and bug fixing
  - Test cases: Unit tests, non-regression tests
  - Functional tests: Test scenarios

- Packaging and deployment
  - Define installation procedures and create scripts.

- Documentation
  - Javadoc, Presentations, Installation document, Final Report

The tasks were sized in time in the Gantt Diagram of the next page. The project had a one month delay margin.


February 2008
March 2008
April 2008
May 2008
June 2008
July 2008
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
Simulator overview [10 Day(s)]
Rerun simulator [3 Day(s)]
Funcionalities study [7 Day(s)]
Development plan proposal [26 Day(s)]
Requirements [4 Day(s)]
Frameworks study [8 Day(s)]
Design/Architecture [9 Day(s)]
Preliminar visual design [5 Day(s)]
Development [35 Day(s)]
Enviornmment setup [5 Day(s)]
Layouts and page development [10 Day(s)]
View / Logic separation [10 Day(s)]
Functionality integration [10 Day(s)]
Testing & Bug Fixing [15 Day(s)]
Test Cases [5 Day(s)]
Functional Testing [10 Day(s)]
Packaging/Deployment [10 Day(s)]
Documentation [15 Day(s)]

Ajax Framework Study

Before defining the architecture of the new simulator there was a missing step. This one was to choose a web technology to develop the presentation layer. This technology is key to the portability. This decision defines the component organization and the new structure of the simulator.

If we chose a technology like php + mysql we would need a web server like apache that deals with php extensions and a mysql server. If we chose something like ASP.NET we would need the Microsoft Internet and Information Server with the .NET Framework. A Java web technology like JSP or JSF would require a Servlet container and and/or application server.

The main requirement is that the platform/framework/technology used must be AJAX based. A study of AJAX frameworks was carried out. Afterwards the most suitable frameworks were selected and studied in detail and a decision was taken.

What is AJAX?

According to Wikipedia: *“AJAX (Asynchronous JavaScript and XML), or Ajax, is a group of inter-related web development techniques used for creating interactive web applications. A primary characteristic is the increased responsiveness and interactivity of web pages achieved by exchanging small amounts of data with the server "behind the scenes" so that the entire web page does not have to be reloaded each time there is a need to fetch data from the server. This is intended to increase the web page's interactivity, speed, functionality, and usability.”*

Summarizing, AJAX is not a programming language neither a single technology. We could define it as a technique or a mash up of techniques. It is possible to program raw AJAX applications. Raw means to program it from the scratch: dealing with the connections and its protocols from the javascript side and handle directly DOM elements. Even if it is possible, it is a tedious not very productive task. Depending on the need of each developer/application, an implementation of AJAX will have to be chosen in order to develop a rich internet application. Also called as RIA, the rich internet applications are the web sites developed under AJAX technologies.

There are thousands of AJAX implementations. These depend on the programming language, the abstraction level, operative system, browser, programming paradigm etc. We could group them in the following types:

- Direct AJAX
- AJAX Components
- Server-driven Frameworks

Direct Ajax Frameworks

These frameworks require the expertise of the developer with HTML, CSS and Javascript. The framework API deals directly with HTML elements. Normally are used to enrich dynamic web applications that have not been designed to support AJAX and for simple solutions.

The most common direct Ajax frameworks are also so-called JavaScript libraries.

Ajax Components Frameworks

These frameworks offer pre-built components, such as tabbed panes, calendars, menus, etc, which automatically create and manage their own HTML. Components are generally created via JavaScript, XML tags or by adding special attributes to normal HTML elements. These frameworks are generally larger, and intended for web sites rather than web applications.

Some component frameworks require the developer to have extensive HTML/CSS/Ajax experience and to do cross-browser testing. For example, grids, tabs, and buttons may be provided, but user input forms are expected to be authored directly in HTML/CSS and manipulated via Ajax techniques. Other frameworks provide a complete component suite such that only general XML and/or JavaScript abilities are required.

Server-Driven Frameworks

Components are created and manipulated on the server using a server-side programming language. Pages are then rendered by a combination of server-side and client-side HTML generation and manipulation. User actions are transferred to the server through asynchronous connections, server-side code manipulates a server-side component model, and changes to the server component model are reflected on the client automatically.

These frameworks offer familiarity for server-side developers at the expense of some degree of power and performance.

There are Ajax frameworks that handle the presentation layer completely within the browser. These offer greater responsiveness because they handle many more user interactions without server involvement.

On the other hand in a server-driven model, some UI interactions may cause many network requests. Furthermore, server-dependent Ajax frameworks will never be able to offer offline support.

AJAX framework categories comparison

	Direct Ajax	Ajax Components	Server-Driven
Client/Server Side	Client	Client / Server	Server
HTML, CSS, JScript skills needed	High	Medium	None
Abstration level	Low	Medium	High
Direct *XMLHttpRequest* manipulation	Yes	No	No
Cross-browser testing needed	Yes	Yes	No
Reusability	Low	High	High
Developing Time	High	Medium	Low
Data-Transmision Overhead	Low	Medium	Medium/High
Client-side Overhead	Low	Medium/High	Depend
Maintainability Cost	High	Medium	Low
Need of an intelligent server	No	No	Yes
High level language programming	No	No	Yes
Personalization of components	High	Medium	Low

Analysing the table it was possible to make a better decision on what kind of platform was the most suitable to use.

Direct Ajax frameworks are the most efficient both in consumed resources and resources needed to deploy the application. On the other hand, a high level of technical skills, are needed to develop this way. The maintainability cost is very high and the reusability of the code is practically inexistent.

In contrast Server-driven frameworks need more server-side resources but are very user-friendly to use. The applications are coded in a high level language. This enhances the reusability and maintainability of the code. The learning curve for this kind of framework is much lower for an experienced Java Swing developer.

Between both categories there are the Ajax-components. They are normally used to add Ajax functionalities to static/non-Ajax web applications. Some are closer to the direct-Ajax platforms others are closer to the server-driven ones. Normally basic to medium skills in HTML, JS and CSS are needed to use these frameworks. Thus, the learning curve is higher and the maintainability as well. The reusability is high because of the isolation of components. Normally these platforms are a mix of technologies and do not follow standards.

As discussed, the set of frameworks that suits better the migration needs is the server-driven. The original simulator was developed using the Java programming language. Most of server-driven frameworks use java as the main language so it enhances the plug-ability of the GUI. The most important requirements needed for the application are reusability, maintainability and productivity. Server-driven is the category that best balances these requirements.

Server-driven Framework Implementations

Basically there are two kinds of server-side frameworks. The firsts handles the presentation completely from within the browser. No server interaction is needed for the presentation logic. This gives a high grade of responsiveness but makes the client very heavy because big libraries are loaded into the client browser. These are so-called thick-clients. The bandwidth overhead is low because a connection is not needed for each user interaction. On a server failure, presentation logic would keep working.

On the other hand there are the server-dependent frameworks. Few libraries are loaded on the client converting it into a thin-client. Every interaction with the UI needs a connection to the server and it decides what to do. The view state is stored and handled in the server. The set of operations to be performed is broader because the server provides a more advanced programming environment than a web browser.

In order to keep the dependencies of the simulator and not to add new ones, the technologies studied have been limited to the Java based ones. Seven frameworks have been chosen: ThinWire, GWT, Echo, WingS, ZK, ICEFaces and Flex. Flex and GWT handle the presentation from the browser and the others are server-dependent.

GWT: Google Web Toolkit

A very popular framework built originally for developing Google's applications. Applications are coded in Java and transformed into web pages using a compiler. It is basically a Java-to-JavaScript compiler that generates HTML and JS code. The code generated does not necessary need to interact with the server. The applications can be deployed in a static web server. The programming capabilities are low because the compiler is reduced to a subset of java 1.4 objects.

The applications developed are distributed. They need another layer to provide the data source, for example a web service. That implies security issues and the addition of new components.

The GWT community is large and the product is stable. By default there are not many widgets but there are well known extension libraries that provide a large amount of them. The productivity is high but the learning curve is low.

Flex

It is the Adobe RIA (Rich Internet Applications) builder. It is based on the flash technology. The coding is done through a visual editor that generates xml code. It is compiled and converted into flash applications.

The developing is very fast and is targeted to the presentation layer. The code on the server side has to be done using another technology like a Servlet, PHP or ASP. Basically the Flex application retrieves a XML file with all the data needed from a web service provided by the technology chosen.

There is a large community and there exist plenty of built-in widgets and controls. The implementation would be done with JSeamless. It is a Java wrapper for the Flex frontend.

Echo2, ThinWire, WingS

These three frameworks have much in common. All are server-dependent based on events and action listeners. The model is very similar to Java Swing, in the event handling and the layout and component managing. The coding is done entirely in Java. The application is deployed into a Servlet container. Functional and eye-candy applications are easy to develop with this kind of frameworks. The programming environment is uniquely Java-based with all the capabilities of the language.

The drawback is that these frameworks use a swing-like event driven model but this is not really compliant with the standard. The separation of the view and the controller (MVC pattern speaking) model has to be done manually, is not forced by the framework.

The differences of each framework are not big. The study in this case is the quantity of widgets, the documentation, the community and support. There is a lack of them in some cases due of the immaturity of the framework.

ICEFaces, ZK

Both are based on template models. There exists a separation of the view and the controller. The view is an XML file that is handled as an object by a java controller. In ZK the template view is based on the XUL (Mozilla standard) components. In the ICEFaces case the templates are based on the Java Server Faces (J2EE Standard).

The reusability of the code is high in both view and controller side. That's because they have plenty of widgets and controls ready to use.

JSF is supported by Sun. It has lots of documentation and a big community. There exists visual plug-ins for the major IDEs. These allow the programmer to design impressive interfaces, drag-and-dropping elements into the layout.

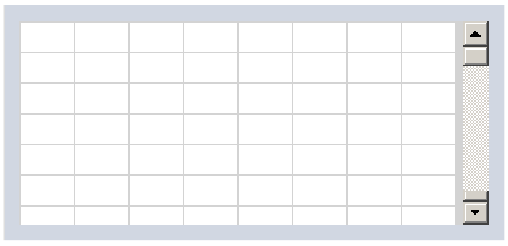
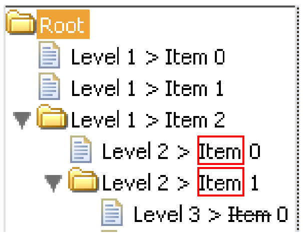
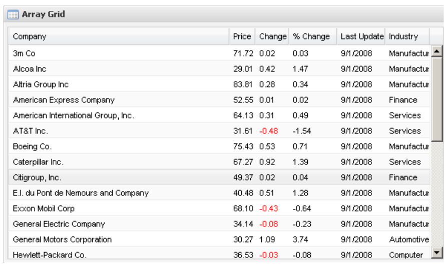
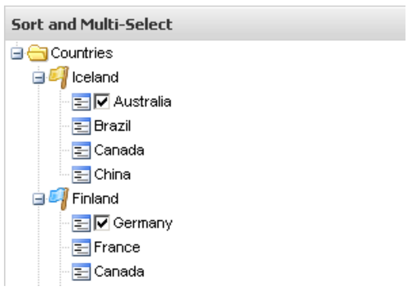
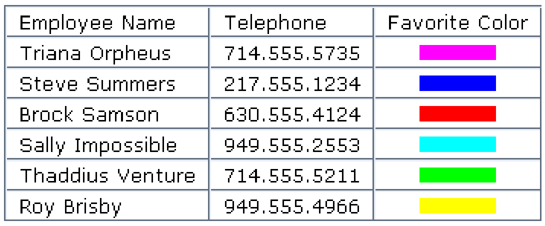
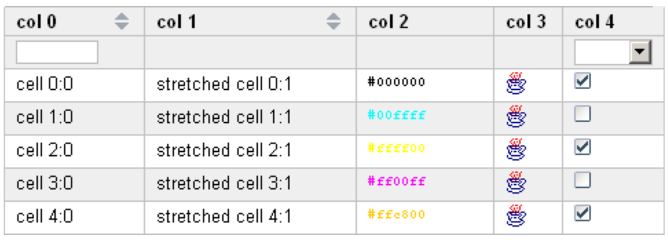
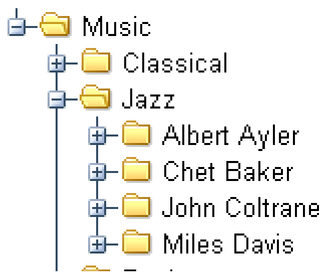
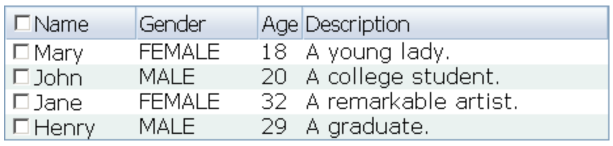
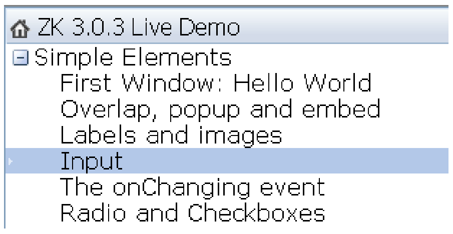
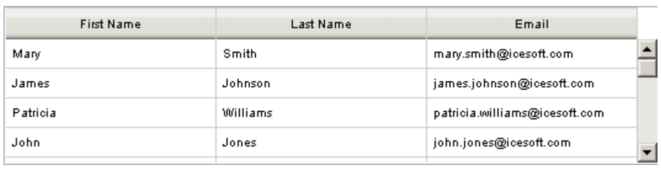
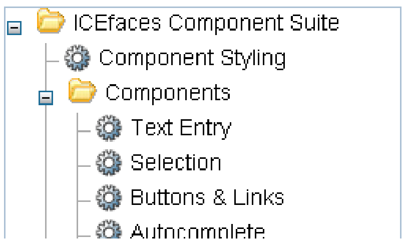
On the other hand ZK is more immature and does not have any visual editors.

Comparative tables

Framework	Developing languages	Deployment	Debug	Browser plugins needed
GWT	Subset of Java 1.4	Java to JS compiler, no servlet cnt. needed	Browser plugin	No
Flex	XML, JS + Server Side Tech.	Depend on the server side tech.	Eclipse, Flex Builder	Flash Player
Echo 2	Java >= 1.4.2	Servlet container	Java IDE	No
WingS	Java >= 1.5	Servlet container >2.3	Java IDE	No
ThinWire	J2EE >= 1.3	Servlet container	Java IDE	No
ZK	Java, XUL, ZUML	Servlet container >2.3	Java IDE	No
ICEFaces	Java, JSF, XML	Servlet container	Java IDE + Plugins	No

Framework	Application Server Needed	View-State Storage Side	Exposed code to client	Documentation available	Quantity of available controls/widgets
GWT	No	Client	Yes	High	High (not native)
Flex	Depend	Client	Yes, but compiled	High	High
Echo 2	Yes	Server	No	Few	Normal (not native)
WingS	Yes	Server	No	Few	Normal
ThinWire	Yes	Server	No	Several	Normal/High
ZK	Yes	Server	No	Several	High
ICEFaces	Yes	Server	No	High	High

Framework	Weight	Window style / Layout	Embedded browser support	Security Control	Licence
GWT	Heavy (>350kb)	Docked	Special	Low	Apache v2
Flex	Heavy (Flash bin + data)	Free	If supports flash	None	MPL v1.1 & Propertary
Echo 2	Light (N/A)	Docked/Floating	N/A	N/A	Mozilla v1.1
WingS	Light (N/A)	Docked	N/A	N/A	LGPL v2.1
ThinWire	Light (35kb shared JS)	Docked/Floating	Yes	High	LGPL
ZK	Light (N/A)	Docked	Yes	External Component	GPL
ICEFaces	Light (N/A)	Docked	Yes	Highest	Mozilla v1.1

Framework	Table	Tree
ThinWire		
GWT		
Echo2		Not provided
Wings		
ZK		
ICEFaces		

ThinWire tree:

- Root
  - Level 1 > Item 0
  - Level 1 > Item 1
  - Level 1 > Item 2
    - Level 2 > Item 0
    - Level 2 > Item 1
      - Level 3 > Item 0

GWT table:

Array Grid

Company	Price	Change	% Change	Last Update	Industry
3m Co	71.72	0.02	0.03	9/1/2008	Manufactur
Alcoa Inc	29.01	0.42	1.47	9/1/2008	Manufactur
Altria Group Inc	83.81	0.28	0.34	9/1/2008	Manufactur
American Express Company	52.55	0.01	0.02	9/1/2008	Finance
American International Group, Inc.	64.13	0.31	0.49	9/1/2008	Services
AT&T Inc.	31.61	-0.48	-1.54	9/1/2008	Services
Boeing Co.	75.43	0.53	0.71	9/1/2008	Manufactur
Caterpillar Inc.	67.27	0.92	1.39	9/1/2008	Services
Citigroup, Inc.	49.37	0.02	0.04	9/1/2008	Finance
E.I. du Pont de Nemours and Company	40.48	0.51	1.28	9/1/2008	Manufactur
Exxon Mobil Corp	68.10	-0.43	-0.64	9/1/2008	Manufactur
General Electric Company	34.14	-0.08	-0.23	9/1/2008	Manufactur
General Motors Corporation	30.27	1.09	3.74	9/1/2008	Automotive
Hewlett-Packard Co.	36.53	-0.03	-0.08	9/1/2008	Computer

GWT tree:

Sort and Multi-Select

- Countries
  - Iceland
    - ✓ Australia
    - Brazil
    - Canada
    - China
  - Finland
    - ✓ Germany
    - France
    - Canada

Echo2 table:

Employee Name	Telephone	Favorite Color
Triana Orpheus	714.555.5735	🟪
Steve Summers	217.555.1234	🟦
Brock Samson	630.555.4124	🟥
Sally Impossible	949.555.2553	🟦
Thaddius Venture	714.555.5211	🟩
Roy Brisby	949.555.4966	🟨

Wings table:

col 0	col 1	col 2	col 3	col 4
cell 0:0	stretched cell 0:1	#000000		✓
cell 1:0	stretched cell 1:1	#00ffff		☐
cell 2:0	stretched cell 2:1	#ffff00		✓
cell 3:0	stretched cell 3:1	#ff00ff		☐
cell 4:0	stretched cell 4:1	#ffc800		✓

Wings tree:

- Music
  - Classical
  - Jazz
    - Albert Ayler
    - Chet Baker
    - John Coltrane
    - Miles Davis

ZK table:

Name	Gender	Age	Description
☐ Mary	FEMALE	18	A young lady.
☐ John	MALE	20	A college student.
☐ Jane	FEMALE	32	A remarkable artist.
☐ Henry	MALE	29	A graduate.

ZK tree:

ZK 3.0.3 Live Demo

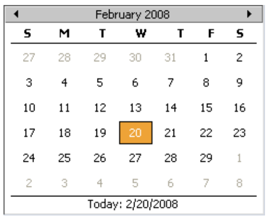
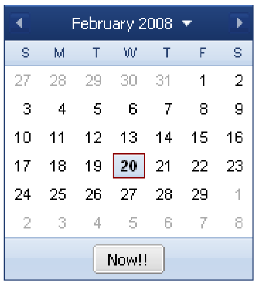
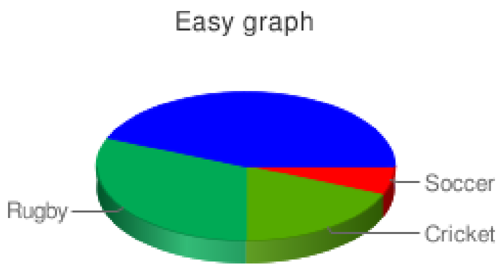
- Simple Elements
  - First Window: Hello World
  - Overlap, popup and embed
  - Labels and images
  - Input
  - The onChanging event
  - Radio and Checkboxes

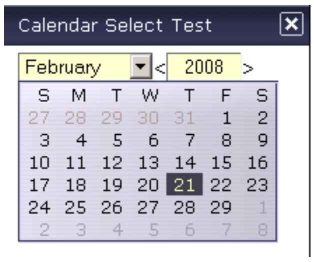
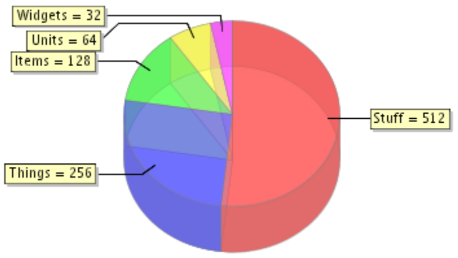
ICEFaces table:

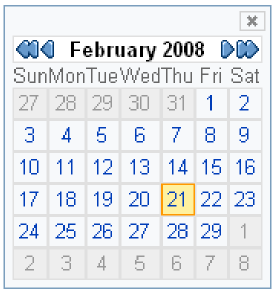
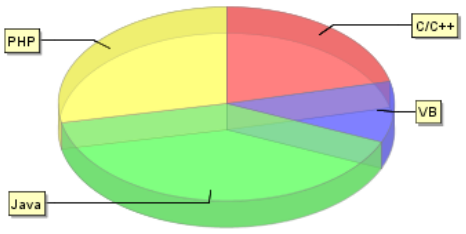
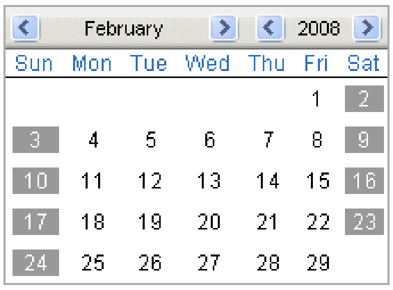
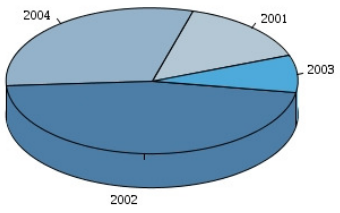
First Name	Last Name	Email
Mary	Smith	mary.smith@icesoft.com
James	Johnson	james.johnson@icesoft.com
Patricia	Williams	patricia.williams@icesoft.com
John	Jones	john.jones@icesoft.com

ICEFaces tree:

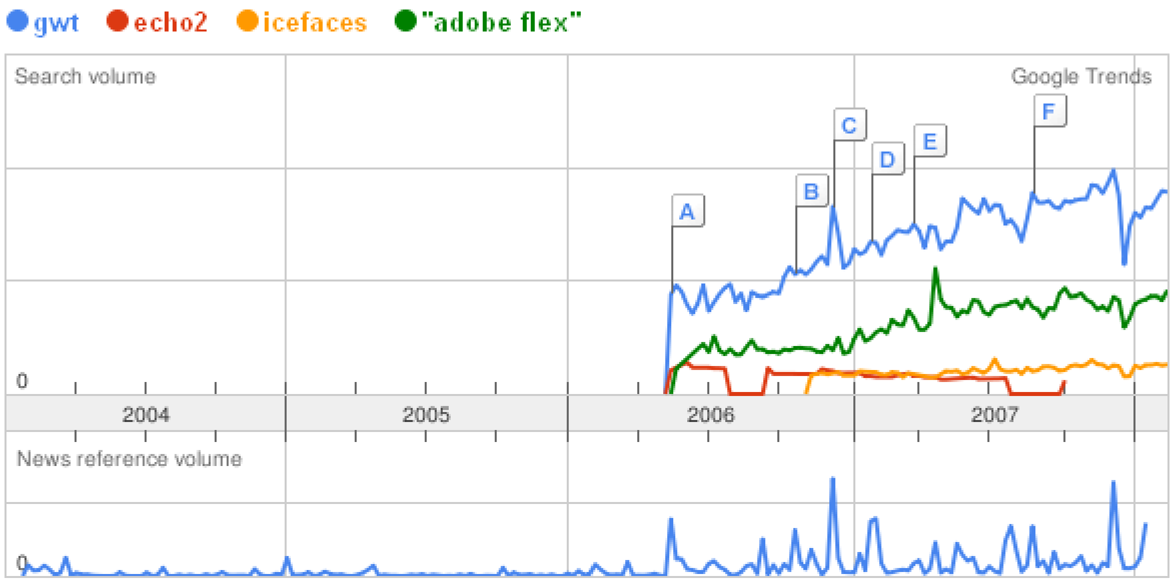
- ICEfaces Component Suite
  - Component Styling
  - Components
    - Text Entry
    - Selection
    - Buttons & Links
    - Autocomplete

Framework	Date Picker	Chart
ThinWire		Not provided
GWT		

Echo2		 Widgets = 32
		Units = 64
		Items = 128
		Stuff = 512
		Things = 256

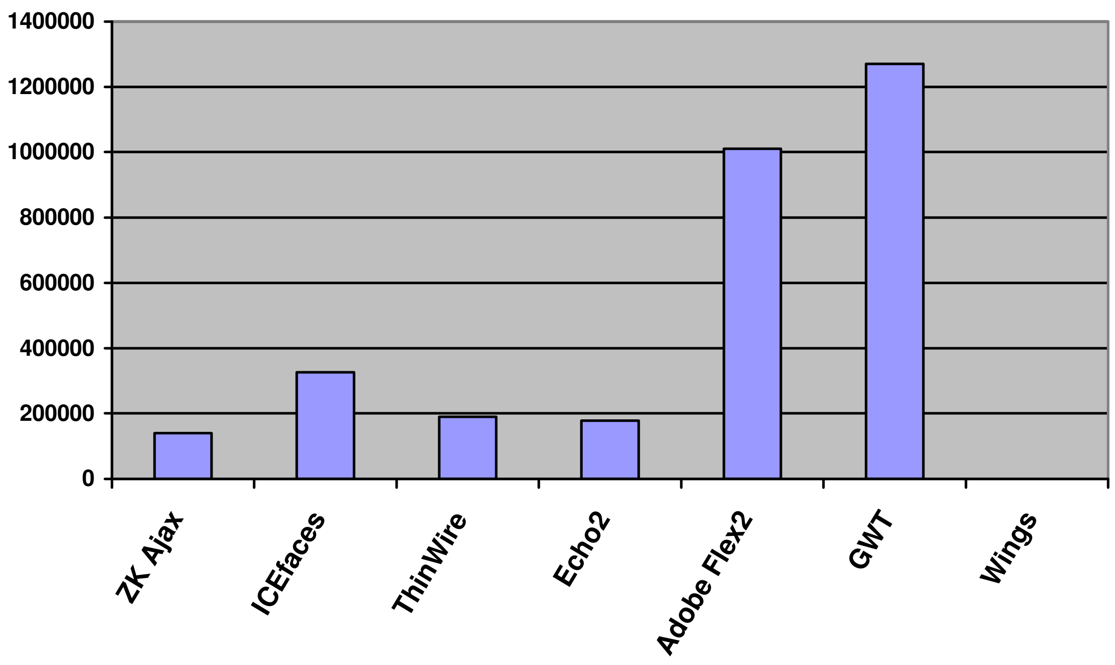
Wings		Not provided
ZK	Not provided	
ICEFaces		

Statistics from Google Trends



These statistics should not be taken as a decisional factor just as information about how market looks. The Y axe represents the number of queries in Google for the correspondent keyword. The frameworks that are not shown do not have enough results to appear.

Statistics of Google Search engine results



The Y axe represents the number of results in google for the corresponding query.

These charts give an idea of the most popular platforms in the market. Polularity usually means more doumentation, examples and support.

Choosing a Framework

Why were GWT and FLEX rejected?

As explained above GWT and Flex are two frameworks that handle the presentation in the client side. That makes the browser manage much information and may reduce responsiveness. Moreover it makes the application less scalable.

Another drawback is that these are distributed applications that need a server side framework apart of the technology itself.

Coding in GWT is done entirely in java, but with Flex there are several new scripting languages and templates such as ActionScript.

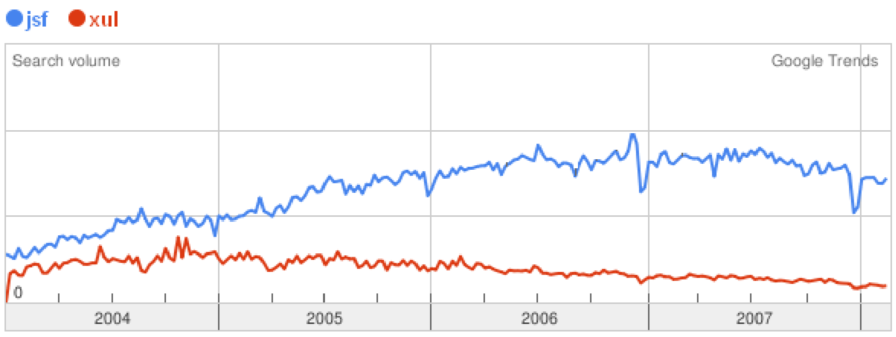
These are the reasons why we choose not to use these frameworks.

The best of each category

Frameworks that use templates are ZK and ICEFaces. The other three, Echo, ThinWire and WingS are event based and coded purely in java.

The advantage that templates offer is that they force the developer to separate the view from the controller. The templates are coded in XML. ZK follows the XUL Mozilla standard and ICEFaces follows the JSF Sun standard. This enhances the reusability of the widgets/controls produced. That is why these frameworks have a larger set of built-in available widgets to use. On the other hand the layout of the pages is also managed using templates. This makes it depend directly on HTML and developer defined tags. In ICEFaces, everything is XML compliant but one may find HTML, JSP, JSF and ICEFaces tags in the same template.

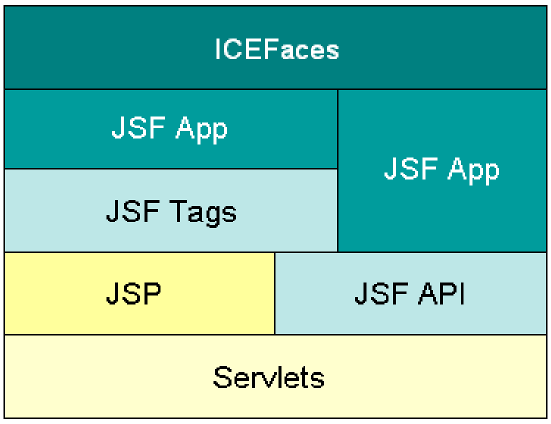
Between ZK and ICEFaces, the second wins the duel. JSF is broadly extended standard on the java community. In contrast, XUL is not that extended in web developing. Mozilla Firefox uses it to handle its presentation layer and it has not popularly spread in other fields. The chart below gives an approximate image of the popularity of both.



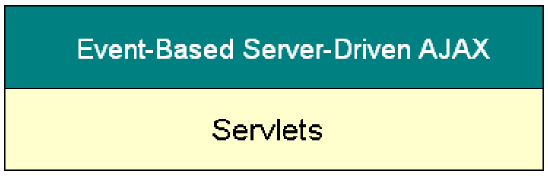
Templates are difficult to debug and to check for its correctness. Are also difficult to maintain cause the lack of extensiveness.

In the side of purely java coding style frameworks, the developer does not have to deal with XML templates. The code is written entirely with the same language. That eases the maintainability of the application. The separation of the view and the controller, the MVC pattern, is not forced by the technology but it is the responsibility of the developer whether to use it or not.

All these frameworks are built over a Servlet layer belonging to a Servlet container. Taking a look to the component diagram of ICEFaces one can see that it deals with a lot of components. The layers have been added as long as the web developing has been advancing until the arrival of web 2.0 with AJAX and rich components.



The same architecture diagram on an event-based framework is much simpler. By the other hand this component has been coded from the scratch thinking in the web 2.0 needs directly. It has not been an extension of a previous technology that could not satisfy the current needs.



These diagrams do not take in consideration the request and responses below the Servlet that are coded in HTML, CSS, XML or JSON and JavaScript. This gives a picture of the complexity of such a framework.

Between the three event-based platforms: Echo, Thinwire and WingS, it is hard to decide which one is the best, taking in consideration our needs. Both three are quite recently emerged frameworks. Taking a look to the previous comparison table it is possible to see that the documentation of both is not very big. There is a lack on Echo2 and WingS. Regarding the quantity and quality of built in widgets ThinWire wins the prize. Echo2 also have many widgets but are part of an external library. ThinWire also have support for embedded devices and the security has been taken into consideration in the design of the framework. The most popular of the three is ThinWire followed by Echo2 and further WingS.

ICEFaces vs ThinWire

The framework that fits our needs in the templates caregory is ICEFaces. In the event-server-driven model ThinWire emerges.

ICEFaces	ThinWire
Complex platform	Simple platform
Up-Down Layer Overhead	Low Overhead
Maturity	Immaturity
Tons of documentation	Some documentation
Big community	Small community
Follows JSF standard	Does not follow any standard
Uses templates	Does not use templates
Deals with Java, Servlets, XML, HTML, JSP, Faces and ICEfaces	Deals with Java, Servlets and ThinWire components
High learning curve	Low learning curve
Can be ported to another JSF based framework	Cannot be ported
Harder to portate from a Swing-like application	Easier to portate from a Swing-like application
Needs more server resources	Needs few resources
Depends on many libraries	Does not depend on many libraries
Have a visual GUI editor	Have an immature visual GUI editor

Comparing both there is a shock between productivity and maturity. Finally, with the advise of the members of the team, ThinWire has been choosen.

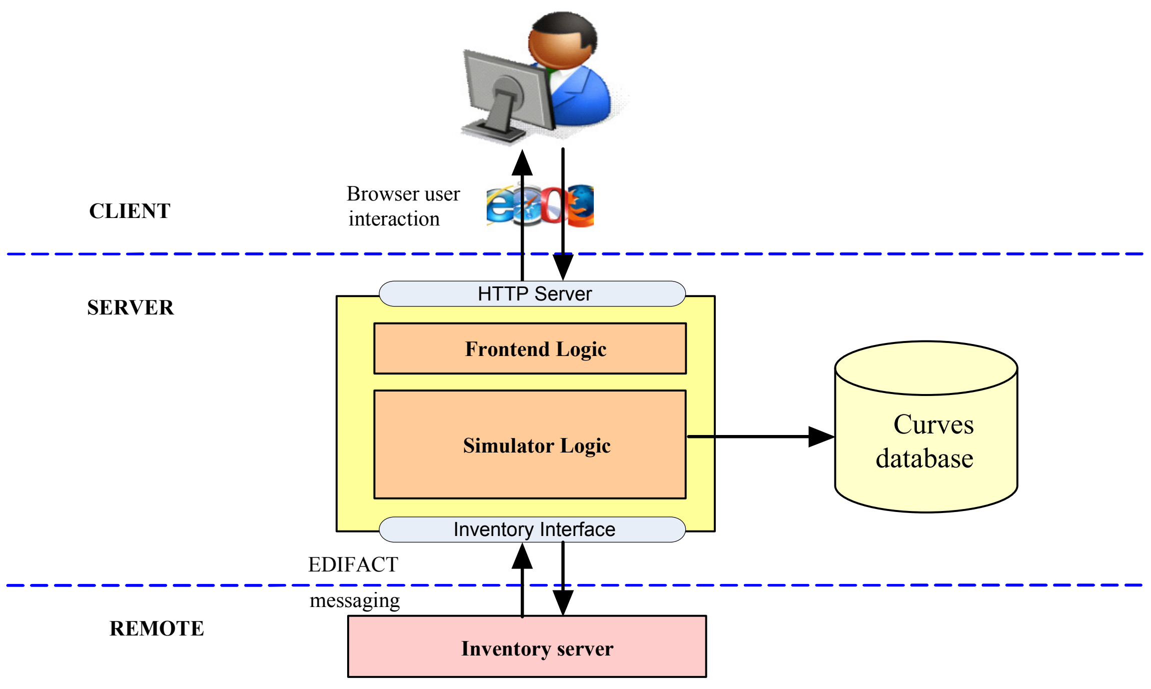
The main reason is that migrate a swing application into a swing-like event and component one is much easier than reenginer all the logic of the presentation that would have to be done with ICEFaces.

Appart of it, as an internship or final project, is more interesting to work in a cutting-edge technology reather than a well-know broadly-used one.

Architecture plan

Review

The Architecture that we were aiming for is represented in the following diagram.



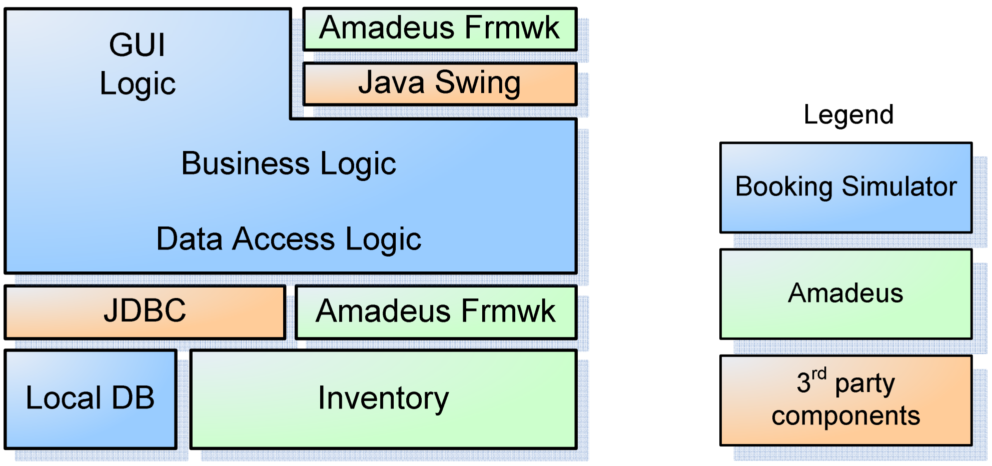
Implementation

That can be achieved with the chosen technologies that are described in the table.

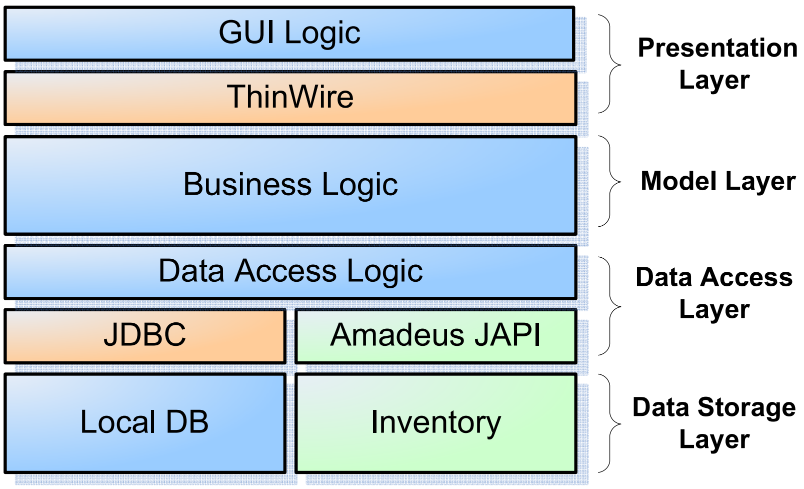
Component	Technology
Java Virtual Machine	Sun Java Virtual Machine version 1.6
HTTP Server	Tomcat Servlet Container 6.0.16 with Servlet Specifications version 2.5
Frontend Framework	ThinWire 1.2 RC2
Local Database	MySQL 5.0 and Oracle
Local Database Interface	JDBC MySQL and Oracle Java Driver
Inventory Interface	JAPI Amadeus Driver
Charts library	JFreeChart
Testing	JUnit 4.1

Components view

The components that are involved in the original simulator, separated by layer, are the following.



The new component architectural plan is to separate the piece of software in 3 layers as in the diagram above. In the fourth layer there are represented the data storage services.



ThinWire is the Ajax Framework chosen for implementing the web-interface. It is located on the presentation layer, interacting with the GUI logic.

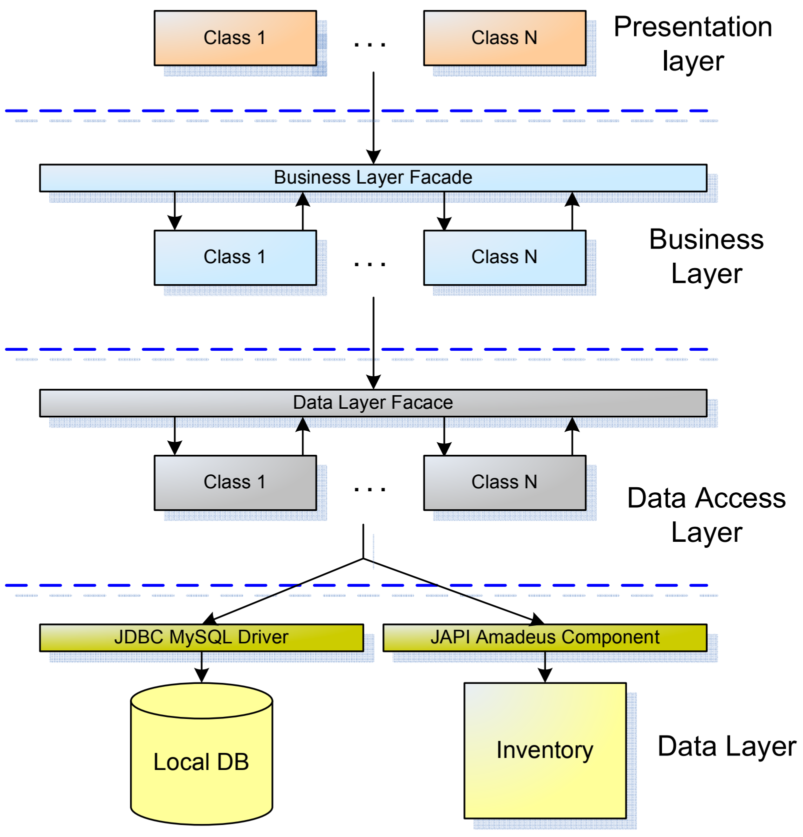
In the model layer there are the model classes that build up the simulator logic. Consequently the functionality tests have to be located in the same layer, so as not to produce unwanted dependencies.

In the data access layer there are classes dedicated to query the data from, and to the different storages. Accessing the local database is done through the JDBC interface. To access the inventory the Amadeus JAPI middleware library was used. This approach is different from the original simulator that was using the Amadeus Framework for handling the communication. Amadeus Framework will no longer be used because it was only providing UI support.

Communication between layers

In order to have the application correctly separated into 3 layers the communication between them has to be engineered. In the diagram above a communication between layers map has been drawn.

That design enhances the portability, changeability and maintainability. The only disadvantage is that it increases the overhead slightly. However its impact is minimal.



Highlighted aspects of the 3-layers separated by façade controllers:

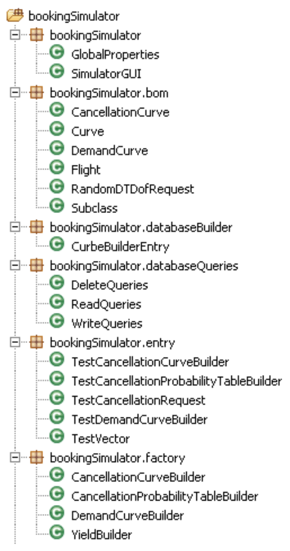
- Entry point in the presentation layer
- Up to Down communication
- Isolation of layers through façade controllers
- Free communication of classes in the same layer
- Common classes named, helpers shared with all the layers (not on the diagram)
- Conceptual isolation of the responsibility of each layer

The data transmission between layers will be handled with the creation of a BOM (Business Object Model) that will represent all the treated data. In the original simulator, there is a minimal BOM and the transmission is done with various Java simple structures.

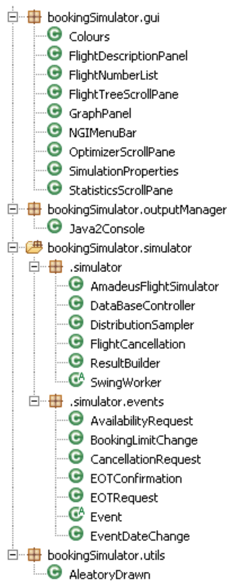
Design plan

Redesign

The following figures give a snapshot of the state of the original simulator. The packages are represented as brown squares and the classes are green circles.



```
bookingSimulator
  bookingSimulator
    GlobalProperties
    SimulatorGUI
  bookingSimulator.bom
    CancellationCurve
    Curve
    DemandCurve
    Flight
    RandomDTDofRequest
    Subclass
  bookingSimulator.databaseBuilder
    CurbeBuilderEntry
  bookingSimulator.databaseQueries
    DeleteQueries
    ReadQueries
    WriteQueries
  bookingSimulator.entry
    TestCancellationCurveBuilder
    TestCancellationProbabilityTableBuilder
    TestCancellationRequest
    TestDemandCurveBuilder
    TestVector
  bookingSimulator.factory
    CancellationCurveBuilder
    CancellationProbabilityTableBuilder
    DemandCurveBuilder
    YieldBuilder
```



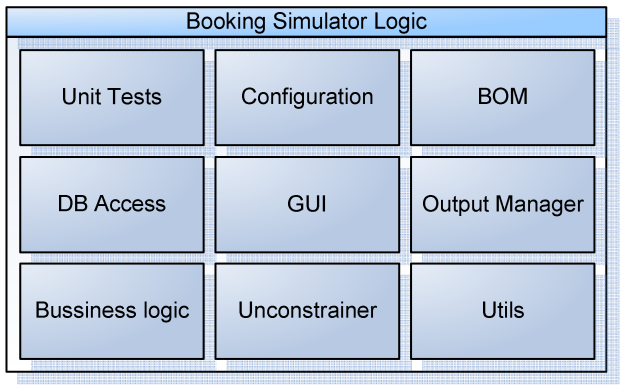
```
  bookingSimulator.gui
    Colours
    FlightDescriptionPanel
    FlightNumberList
    FlightTreeScrollPane
    GraphPanel
    NGIMenuBar
    OptimizerScrollPane
    SimulationProperties
    StatisticsScrollPane
  bookingSimulator.outputManager
    Java2Console
  bookingSimulator.simulator
    .simulator
      AmadeusFlightSimulator
      DataBaseController
      DistributionSampler
      FlightCancellation
      ResultBuilder
      SwingWorker
    .simulator.events
      AvailabilityRequest
      BookingLimitChange
      CancellationRequest
      EOTConfirmation
      EOTRequest
      Event
      EventDateChange
  bookingSimulator.utils
    AleatoryDrawn
```

As the full class structure is a bit confusing this table maps the packages into conceptual units.

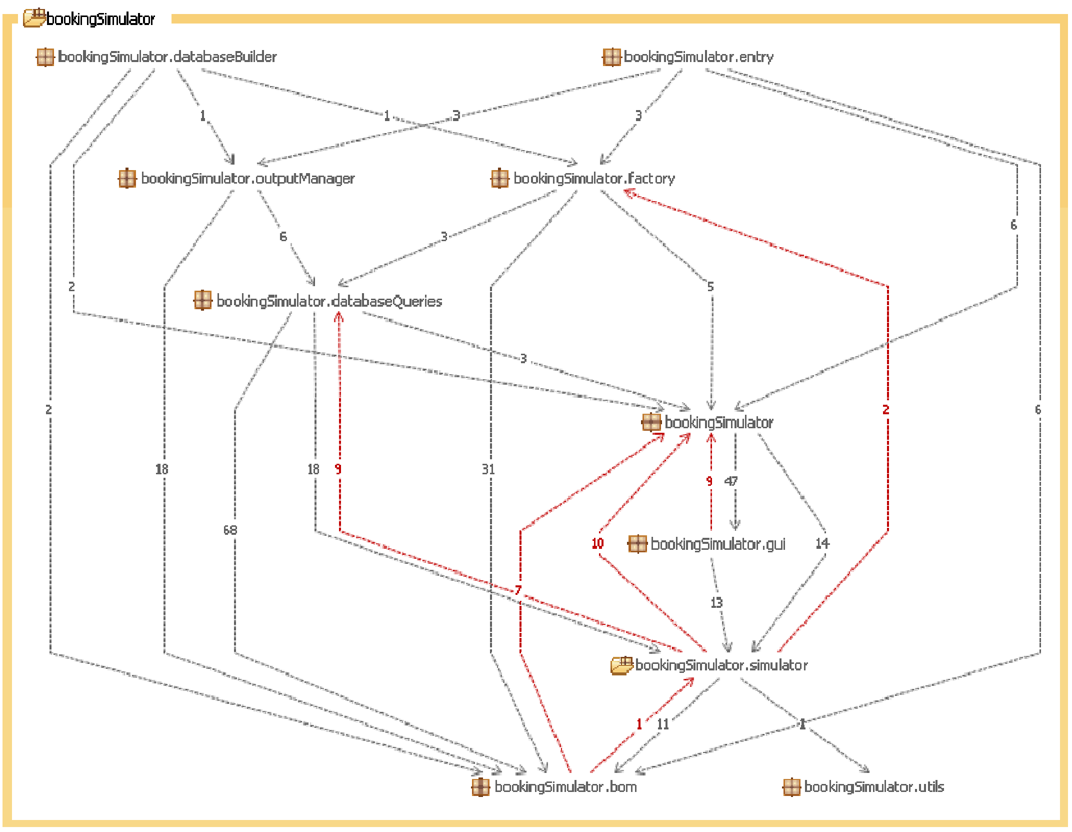
Package	Conceptual unit
bookingSimulator	Configuration
bom	BOM
databaseBuilder	DB Access
entry	Unit Tests
factory	Unconstrainer
gui	GUI
outputManager	Output Manager
simulator	Business Logic
utils	Utils

These conceptual units are easier to understand because the name reflects the function of the mapped packages:

- Configuration: The environment variables and database parameters are coded there.
- BOM: Business Object Model, classes that encapsulates the information
- DB Access: Classes that manage the connection to the local DB and to the Inventory.
- GUI: Views and Amadeus Framework integration
- Output Manager: Interface to print debugging data to the console
- Business Logic: Simulation algorithms
- Unconstrainer: Unconstraining algorithms
- Utils: Helpers to perform different tasks
- Unit tests: Test that check the functionality of the business logic.



All the packages were tangled and coupled, as shown by the diagram below. The red lines represent the couplings. The number identifying each line is the number of calls from one package to another.



This created the need for a redesign of the package structure: *GUI* to be separated and put in to a special package.

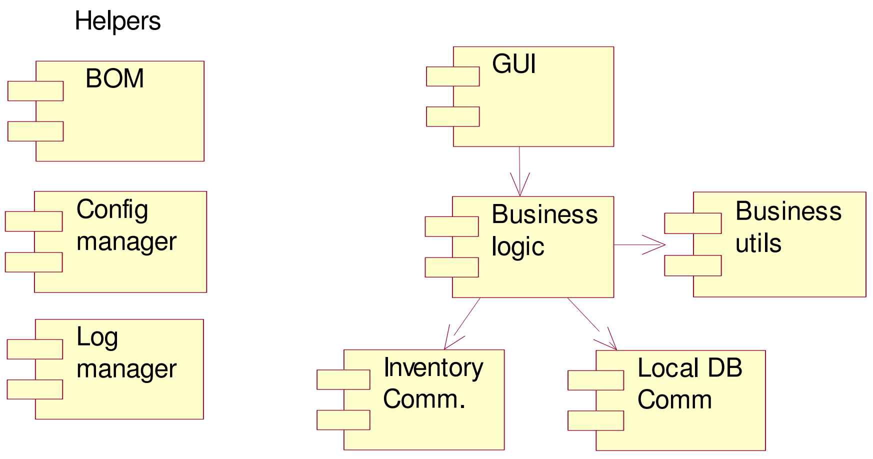
The *Business Logic* to be merged with the *Unconstrainer Logic*. Together with the *Utils* package it will form the *Business Layer*.

In the *Data Layer,* two main packages appear from the *DB Queries* original one. One to be in charge of managing the *EDIFACT messages* handled with *JAPI.* The other is to manage the local database using the JDBC interface implemented with the *Java MySQL driver*.

The Helpers are the packages accessible from all the classes and allow the communication between layers. This is the function of *BOM*. The *Configuration* package is where all the parameters are set and factorized. Is a place to centralize the managing the settings of the application. The O*utput Manager* is the package in charge of printing the information that the application outputs into different channels. The channels might be the standard output, log files, presentation layer, among others.

Program exceptions to be treated uniformly throughout the application. This was not done in the original simulator, and so makes debugging hard.

The diagram located in the next page gives an overview of the new restructuration.



All the packages have the dependency of the Helpers.

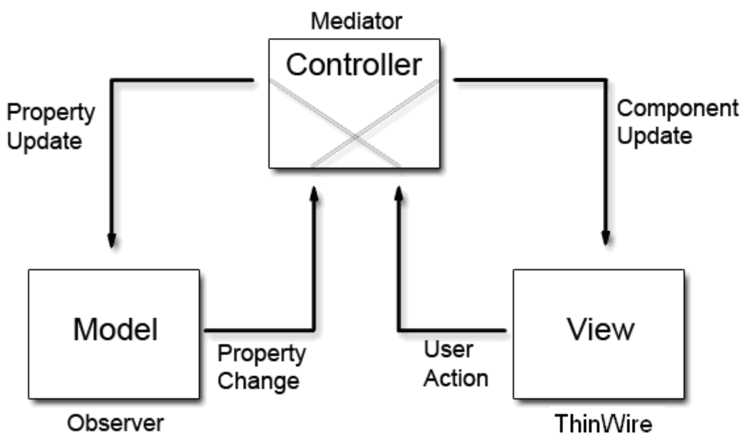
In order to restructure the package, several tasks had to be performed. The bullets below unify all the tasks that had to be done.

- Separate the data layer into inventory and local database access. **(1)**
- Place all the queries to the inventory and to the local database found in the presentation and business layer in the corresponding package. **(1)**
- Translate the direct SQL queries to the inventory into EDIFACT messages. **(1)**
- Create the business layer façade **(1)**
- Merge the simulator logic and the unconstrainer logic and make a unified package called Business Logic. **(2)**
- Create the data access layer façade **(2)**
- Extract the common functions in the Business logic and place them in the Utils package. **(3)**
- Unify the exception treating procedure for all the application. **(3)**
- Unify the Output Manager. Replace the standard outputs. **(3)**
- Create the configuration manager and factorize the configuration variables. **(3)**

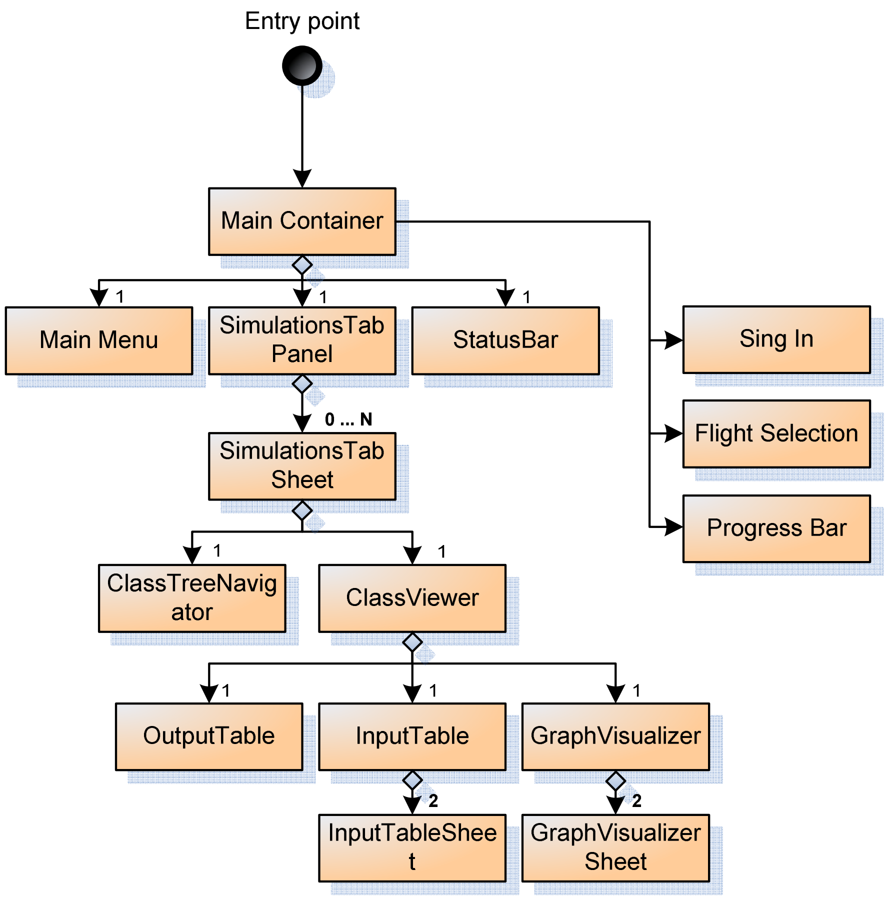
All these tasks lead to a clean design ready to extend and migrate. There are tasks that are more essential than others, illustrated in the following way: **1** essential, **2** necessary, **3** recommended

Presentation layer design

The presentation layer will follow the MVC pattern in order to handle the view events.



The ThinWire elements organization will be done according this hierarchy.



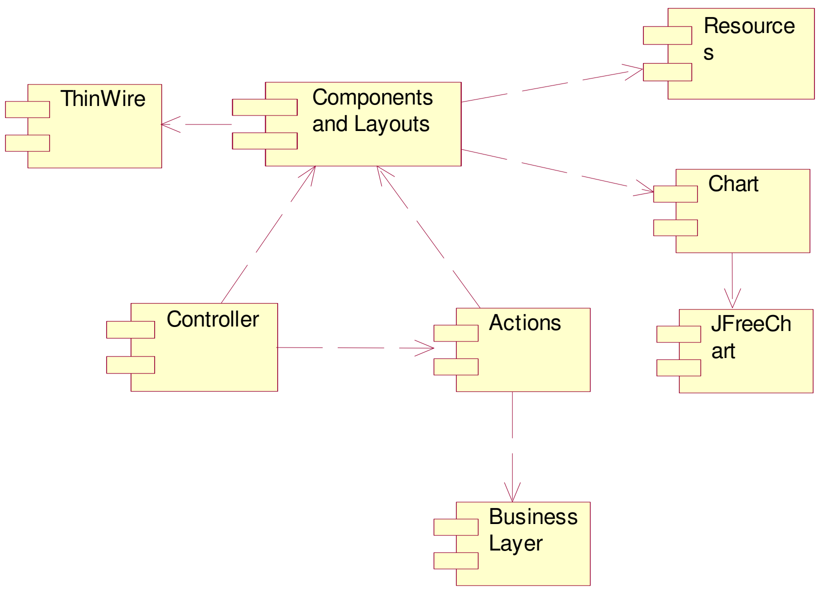
Every elements has to be assigned to a ThinWire component from the suite of components it provides.

Control	ThinWire Implementation
MainContainer	Panel
MainMenu	Menu
SimulationsTab Panel	TabFolder
SimulationsTab Sheet	TabSheet
ClassTree Navigator	Tree
Class Viewer	Panel
OutputTable	GridBox
InputTable	TabFolder
GraphVisualizer	TabFolder

Control	ThinWire Implementation
InputTableSheet	TabSheet
GraphVisualizer Sheet	TabSheet
Sign In	Dialog
Flight Selection	Dialog
Progress Bar	Dialog

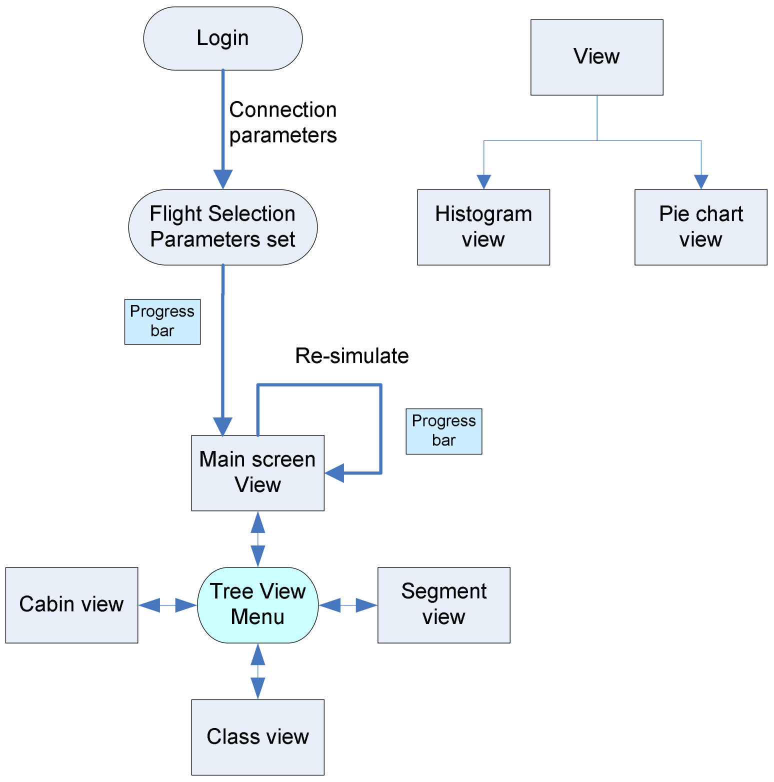
The diagram below is the structure of packages of the presentation layer. On one hand, there is the GUI package that is composed of views and resources. A *View* is where the components of the previous diagram and its interactions are placed. *Resources* are used to store graphical components such as images, style sheets, xml files.

On the other hand there is the controller package. It is in charge of the communication with the business layer. It isolates and factorizes the functions. This is the package that handles the view state.



This design tries to be compilant with the MVC presentation pattern model.

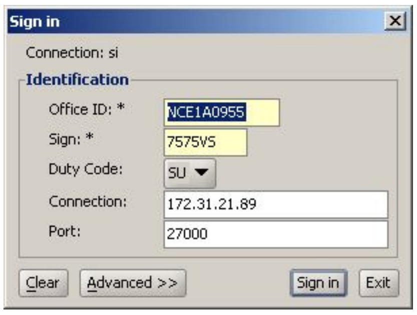
Current functionallities



Making a simplification of the simulator there are the following views:

> The connection screen is named: *SignIn*
> The simulation selection screen is named: *FlightSelection*
> The progress bar is named: *ProgressBar*
> The output statistics table is named: *OutputTable*
> The input settings table is named: *InputTable*
> The navigation tree is named: *ClassTreeNavigation*

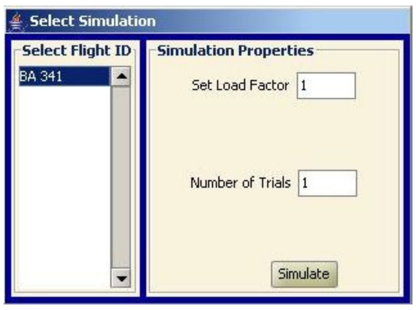
The names are equivalent with the ones in the Presentation Layer design on page 9.



Connection: si

Identification

Office ID: *	NCE1A0955
Sign: *	7575VS
Duty Code:	SU
Connection:	172.31.21.89
Port:	27000



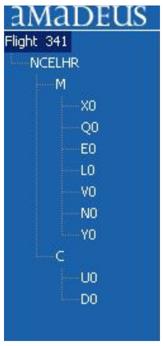
Select Flight ID

BA 341

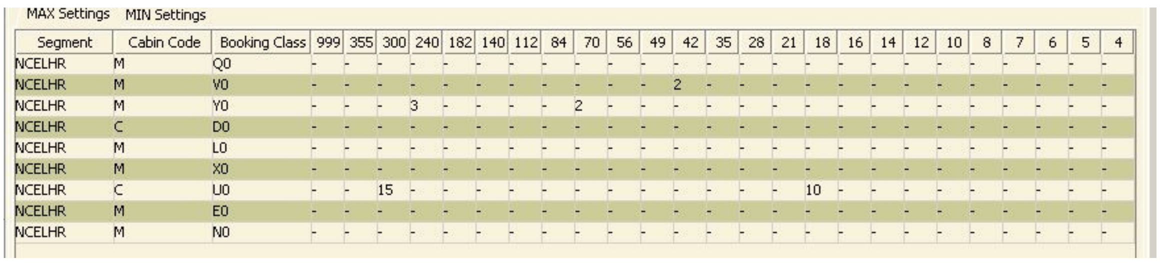
Simulation Properties

Set Load Factor 1

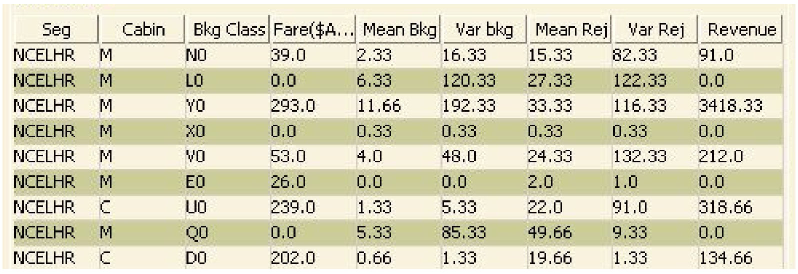
Number of Trials 1



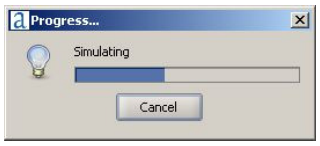
Jordi Planadecursach
July 31st 2008



Segment	Cabin Code	Booking Class	999	355	300	240	182	140	112	84	70	56	49	42	35	28	21	18	16	14	12	10	8	7	6	5	4
NCELHR	M	Q0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	V0	-	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	Y0	-	-	-	3	-	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	C	D0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	L0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	X0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	C	U0	-	-	15	-	-	-	-	-	-	-	-	-	-	-	-	10	-	-	-	-	-	-	-	-	-
NCELHR	M	E0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NCELHR	M	N0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-



Seg	Cabin	Bkg Class	Fare($A...	Mean Bkg	Var bkg	Mean Rej	Var Rej	Revenue
NCELHR	M	N0	39.0	2.33	16.33	15.33	82.33	91.0
NCELHR	M	L0	0.0	6.33	120.33	27.33	122.33	0.0
NCELHR	M	Y0	293.0	11.66	192.33	33.33	116.33	3418.33
NCELHR	M	X0	0.0	0.33	0.33	0.33	0.33	0.0
NCELHR	M	V0	53.0	4.0	48.0	24.33	132.33	212.0
NCELHR	M	E0	26.0	0.0	0.0	2.0	1.0	0.0
NCELHR	C	U0	239.0	1.33	5.33	22.0	91.0	318.66
NCELHR	M	Q0	0.0	5.33	85.33	49.66	9.33	0.0
NCELHR	C	D0	202.0	0.66	1.33	19.66	1.33	134.66



SignIn

Parameters: Office ID, Sign, Duty Code, Connection IP, Port
Buttons: Clear, SignIn, Exit

These parameters are necessary to establish a connection. The current parameters have to be saved in order to remember them the next sign up.

Different sign in procedures might be implemented. The connection will be done directly to the server without passing through the SI service.

FlightSelection

Parameters:

Load factor, range from 1 to 100
Number of trials
Flight Selection, it has to be filled with all the different flights in the local database that are able to be simulated.

Button: Simulate

ProgressBar

Button: Cancel, cancels the current simulation

The progress bar has to be updated meanwhile the simulations run. This can be done using server-push, the server notifies the simulation state to the view. It can also be done from the view, asking the state of the simulation to the model. The first option is more realistic but creates a not needed dependency.

OutputTable

Table containing these fields: Segment, Cabin, Booking class, Average fare, Mean of bookings, Variance of bookings, Mean of rejected, Variance of rejected, Revenue.

That table is filled with data obtained from the simulation. It can only be showed after it. Depending on the selection done in the ClassTreeNavigation the information about the revenue of a class has to be shown or not.

InputTable

Two input tables containing the booking limits for each class. The tables are separated with tabs. One set the booking minimums and the other the booking maximums. These values can be fixed along the DCP timeline.

ClassTreeNavigation

Tree control that allows to walk along the different classes and cabins of a concrete flight.

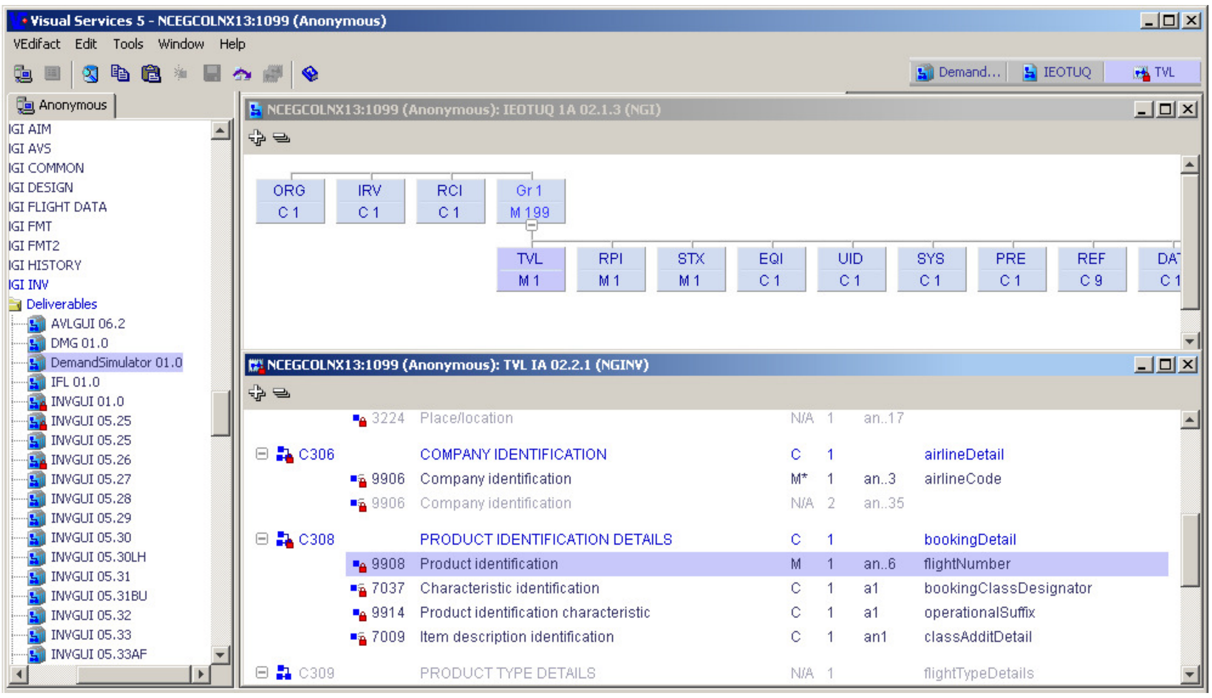
Level 1: Airline Code + Flight Number
Level 2: Segment
Level 3: Cabin code
Level 4: Class code

The tree controls the OutputTable results.

Implementation

Working with Visual EDIFACT

Visual EDIFACT is an Amadeus tool that allows searching definitions in the EDIFACT corporate message repository. It allows querying all the versions of the messages stored and the descriptions of each field.



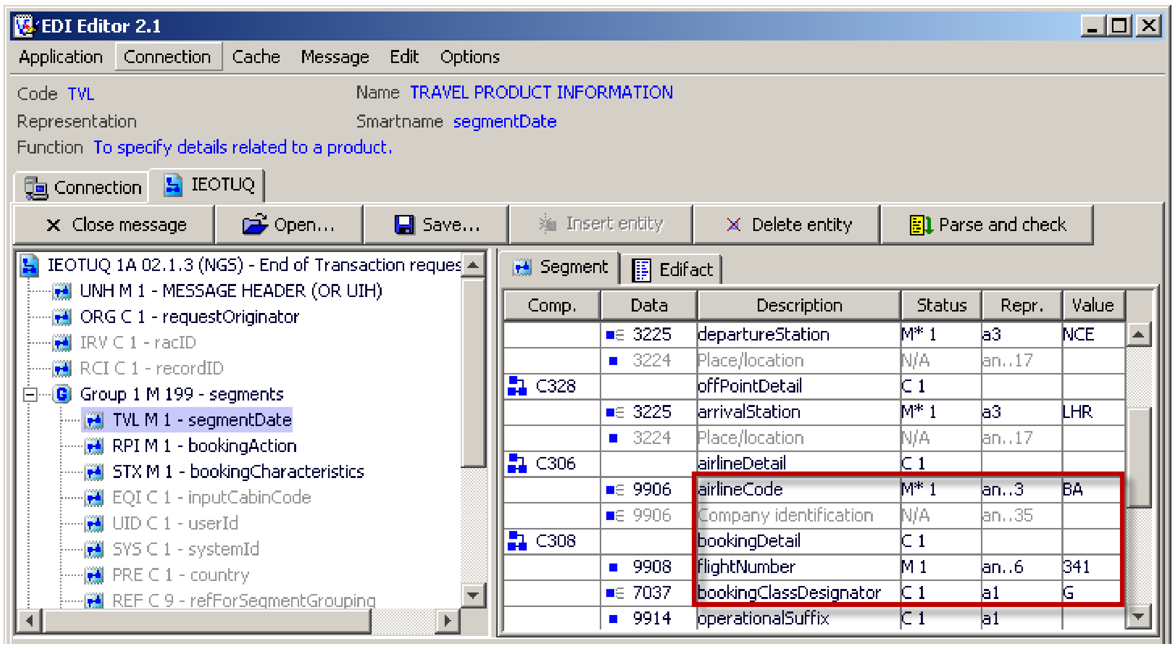
	3224	Place/location	N/A	1	an..17	
C306		COMPANY IDENTIFICATION	C	1		airlineDetail
	9906	Company identification	M*	1	an..3	airlineCode
	9906	Company identification	N/A	2	an..35	
C308		PRODUCT IDENTIFICATION DETAILS	C	1		bookingDetail
	9908	Product identification	M	1	an..6	flightNumber
	7037	Characteristic identification	C	1	a1	bookingClassDesignator
	9914	Product identification characteristic	C	1	a1	operationalSuffix
	7009	Item description identification	C	1	an1	classAdditDetail
C309		PRODUCT TYPE DETAILS	N/A	1		flightTypeDetails

Screenshot of Visual EDIFACT showing the structure of the message IEOTUQ and the flightNumber field selected.

Another interesting feature is that it is possible to store the grammar definitions in an XML file. This XML file is then used to generate stubs and skeletons for different programming languages. This makes the programmer not having to deal directly with EDIFACT thus the abstraction that C++ or Java classes provide.

A complementary tool is EDI Editor. It transforms a plain EDIFACT message, plus the definition of the grammar it has been encoded with, into a readable format. The values can be modified visually and re-encoded. This is very useful when one has to manually send messages to a server in order to try some functionality.

Amadeus has developed this tools because all the communications are based with EDIFACT. This is compulsory knowledge to know when one gets into the Amadeus subsystems.



Code TVL
Name TRAVEL PRODUCT INFORMATION
Representation
Smartname segmentDate
Function To specify details related to a product.

IEOTUQ 1A 02.1.3 (NGS) - End of Transaction reques
UNH M 1 - MESSAGE HEADER (OR UIH)
ORG C 1 - requestOriginator
IRV C 1 - racID
RCI C 1 - recordID
Group 1 M 199 - segments
TVL M 1 - segmentDate
RPI M 1 - bookingAction
STX M 1 - bookingCharacteristics
EQI C 1 - inputCabinCode
UID C 1 - userId
SYS C 1 - systemId
PRE C 1 - country
REF C 9 - refForSegmentGrouping

Comp.	Data	Description	Status	Repr.	Value
	3225	departureStation	M* 1	a3	NCE
	3224	Place/location	N/A	an..17	
C328		offPointDetail	C 1		
	3225	arrivalStation	M* 1	a3	LHR
	3224	Place/location	N/A	an..17	
C306		airlineDetail	C 1		
	9906	airlineCode	M* 1	an..3	BA
	9906	Company identification	N/A	an..35	
C308		bookingDetail	C 1		
	9908	flightNumber	M 1	an..6	341
	7037	bookingClassDesignator	C 1	a1	G
	9914	operationalSuffix	C 1	a1	

Booking Simulator Messages

The booking simulator deals with 8 different types of messages divided into 4 queries and 4 responses. Each query has its corresponding response. Normally the queries end with the letter Q and the responses with R.

IEOTUQ
End of Transaction Request Message: Allows making a booking or a cancellation of a determined class belonging to a flight date. It returns a IEOTUR message that contains the success of the message.

IINVRQ
Stands for Inventory Request: Given a flight-date returns a IINVRR message containing the flight structure for it. It has all the classes structure with its corresponding inventory controls.

IFLIUQ
Stands for Inventory Flight Update Query: Permits changing the inventory controls retrieved with IINVRQ. Returns a IFLIUR messages containing illogicalities in the controls, if there are.

ADMREQ
Allow to change the internal clock of the inventory server. This message is used to change to simulate that the messages has been sent in the past. Returns an ADMRSP confirmation.

The grammars for the usage of the message inside the Java application has been generated with Angel, the Amadeus grammar generator. It generated the stubs and skeletons to set the data to the outgoing messages and retrieve it to the ingoing ones.

OTF Framework

The OTF Framework is the core of Amadeus. It is where all the messages are processed. It consists of a distributed application with different modules. In order to test the simulator I just needed the FLD (flight date) and INV (inventory) modules.

I had to download the code from the CVS and build the framework in the local Linux machines. After that I had to fill the database of the framework with some testing data. This data was basically flight dates with its flight structure and pricing. Once running I was able to target this instance of the inventory from the simulator.

To check whether a message has been sent successfully to the Inventory or not, the logs of the OTF framework have to be checked.

Deployment scripts

After finishing the code part everything had to be put in place. The binary generated by Eclipse (the development tool used) was a WAR file. This is a compressed file that contains a Servlet definition. Inside this Servlet, there are all the simulation interactions. This file is meant to be put in the tomcat directory. Once it is put there, when the first tomcat instance is run, it is automatically deployed and put online.

This simplified a lot the deployment task. The installation script takes the code from the CVS, compiles it and puts it in the Tomcat directory.

There is start-up and a shut down script too. The start-up script launches the tomcat instance. If it is the first time, it will deploy the simulator Servlet into the corresponding context. After, the simulator is accessible from the web. The shut down script cleans the simulator environment and stops the tomcat server.

Apart from that, all the specific parameters can be tweaked in the simulator configuration files and the tomcat configuration files.

Test Cases

The framework used for automated testing of the Booking Simulator is JUnit 4. It can easily be integrated in to every Java project and there is plenty of support from the community. Most development IDEs (i.e. eclipse) provide integration tools for JUnit.

The main problem with the test cases is that the simulator interacts with non static components such as the Inventory. Moreover in order to perform simulations random numbers are used.

The random numbers issue has been solved by applying a fixed seed into the random number generation engine. For testing purposes it is possible to fix it. For simulation purposes the time is used as a seed.

The problem of the variance of the answers of the Inventory server can be solved by hard-coding the response of the messages in Test Tool Server. This response has to be validated manually before assuming it is a valid one. In order to validate the correctness some automated test cases can be defined as well.

Two kinds of tests have been defined, automatic and manual. The automatic tests (so-called unit tests), as the name suggests, can be automated. They do not need human interaction to indicate whether a test has been successful or not. These tests can be assembled together in a suite in order to create a non-regression framework. The tests are executed periodically and indicate if there have been any regressions in the program.

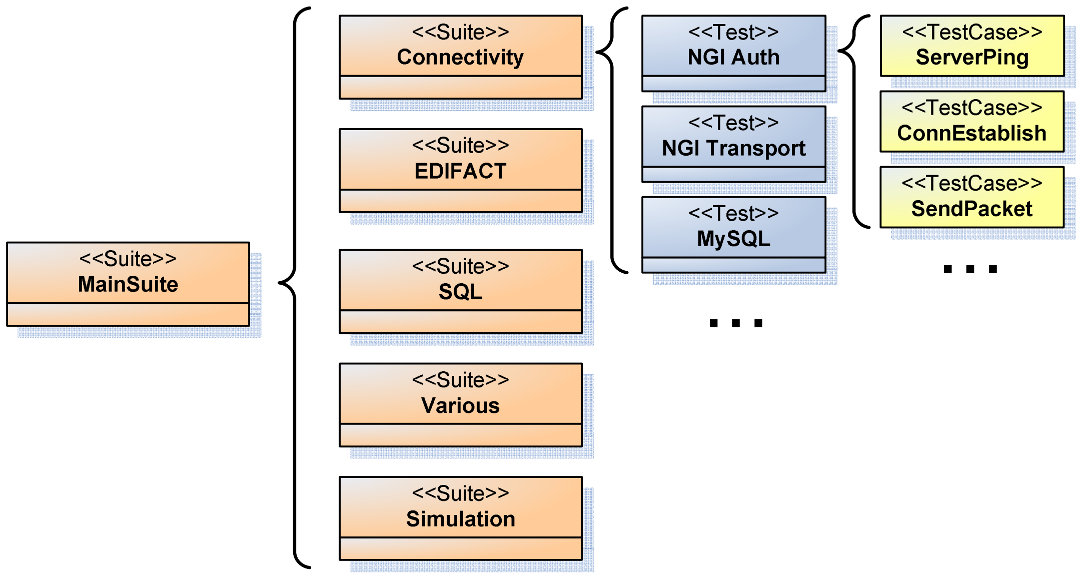
The manual functional tests assure the correctness of behaviours that can not be checked by a computer and thus the results have to be interpreted by an analyst. Even a person that is familiar with the subject would have problems to determine if the results of the simulation are as expected. This has been solved by defining some tricky scenarios and describing its expected results. If one wants to assure the correctness of the simulator, one has to simulate these scenarios and compare the obtained results with the described ones.

Both unit tests and manual scenarios are included in the appendix.

Unit tests organization

The automatic tests are implemented using the Java Unit Test library. In the JUnit framework there are 3 kinds of objects: Suites, Tests and Test Cases. It is a way to have everything sorted hierarchally. A Suite is the main unit and contains different Tests. A Test is the conceptual unit of something we want to assure. It contains various Test Cases. The correctness of each one assures the correctness of the Test itself.

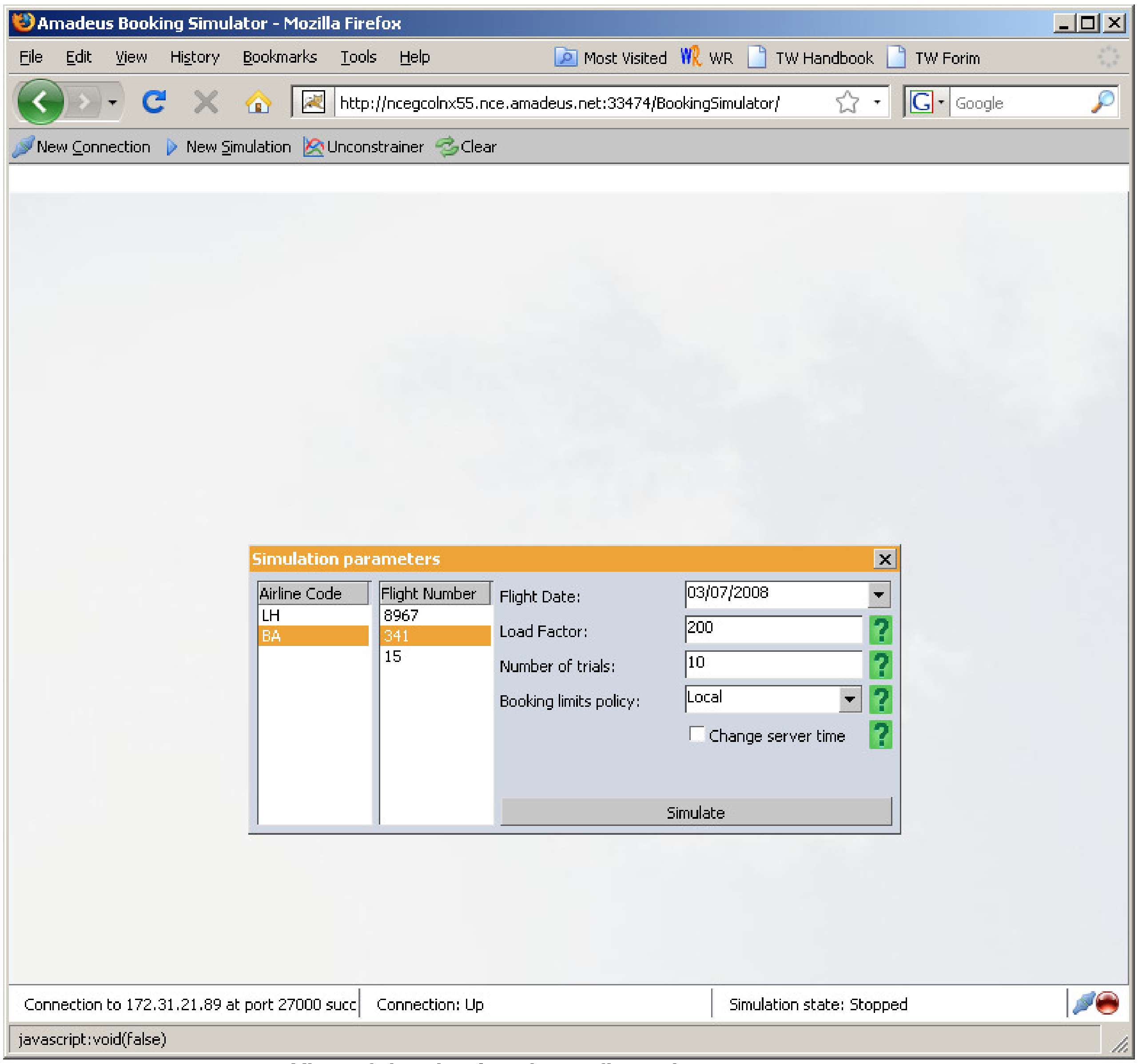
In the booking Simulator the tests are organized as following:



Results

The migration was performed succesfuly. The functionallities were kept and in some cases extended.

New simulation screen


Amadeus Booking Simulator - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Most Visited WR TW Handbook TW Form
http://ncegcolnx55.nce.amadeus.net:33474/BookingSimulator/
Google
New Connection New Simulation Unconstrainer Clear
Simulation parameters
Simulate

Airline Code	Flight Number
LH	8967
BA	341
	15

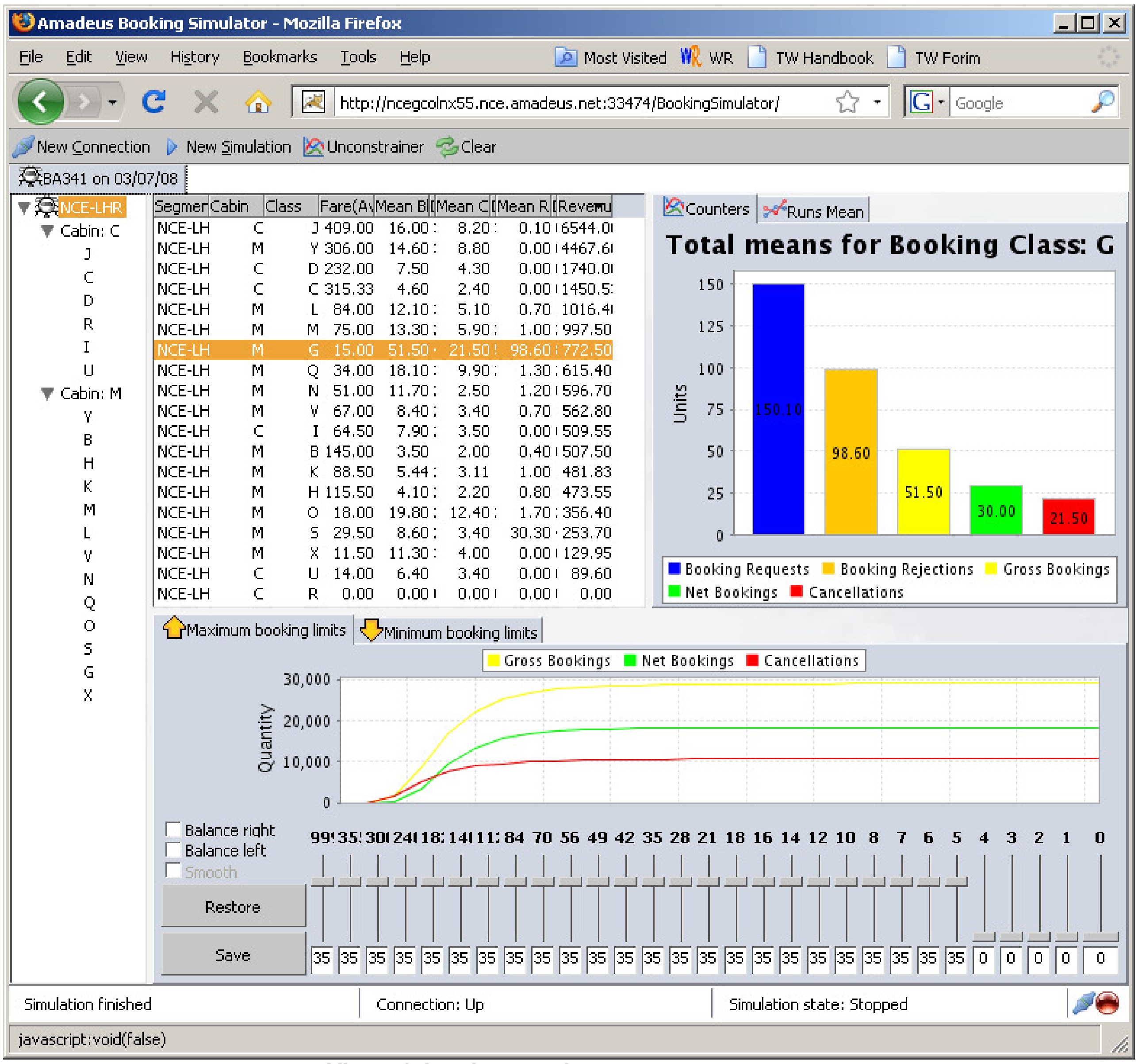
Flight Date: 03/07/2008
Load Factor: 200
Number of trials: 10
Booking limits policy: Local
☐ Change server time

Connection to 172.31.21.89 at port 27000 succ | Connection: Up | Simulation state: Stopped

javascript:void(false)

View of the simulator's configuration screen

Results screen



Segmer	Cabin	Class	Fare(A	Mean Bl	Mean C	Mean R	Revenu
NCE-LH	C	J	409.00	16.00	8.20	0.10	6544.0
NCE-LH	M	Y	306.00	14.60	8.80	0.00	4467.6
NCE-LH	C	D	232.00	7.50	4.30	0.00	1740.0
NCE-LH	C	C	315.33	4.60	2.40	0.00	1450.5
NCE-LH	M	L	84.00	12.10	5.10	0.70	1016.4
NCE-LH	M	M	75.00	13.30	5.90	1.00	997.50
NCE-LH	M	G	15.00	51.50	21.50	98.60	772.50
NCE-LH	M	Q	34.00	18.10	9.90	1.30	615.40
NCE-LH	M	N	51.00	11.70	2.50	1.20	596.70
NCE-LH	M	V	67.00	8.40	3.40	0.70	562.80
NCE-LH	C	I	64.50	7.90	3.50	0.00	509.55
NCE-LH	M	B	145.00	3.50	2.00	0.40	507.50
NCE-LH	M	K	88.50	5.44	3.11	1.00	481.83
NCE-LH	M	H	115.50	4.10	2.20	0.80	473.55
NCE-LH	M	O	18.00	19.80	12.40	1.70	356.40
NCE-LH	M	S	29.50	8.60	3.40	30.30	253.70
NCE-LH	M	X	11.50	11.30	4.00	0.00	129.95
NCE-LH	C	U	14.00	6.40	3.40	0.00	89.60
NCE-LH	C	R	0.00	0.00	0.00	0.00	0.00

```
99 35 30 24 18 14 11 84 70 56 49 42 35 28 21 18 16 14 12 10 8 7 6 5 4 3 2 1 0
35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 0 0 0 0 0
```

Simulation finished | Connection: Up | Simulation state: Stopped

javascript:void(false)

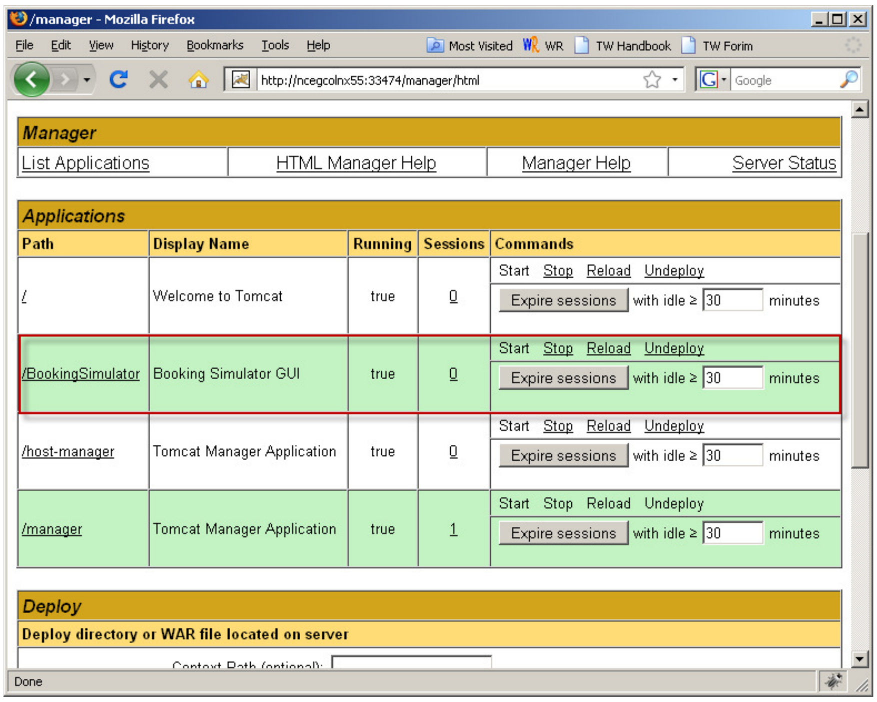
View of the simulator's results screen

In the left there is the flight structure with its classes. In the bottom the bookings limits can be modified with sliders. The curves above the sliders show the historical demand curves that help to set up the limits. The main table contains the counters (bookings, cancellations, rejections, revenue) for each class. The charts provide a graphical representation of the revenues and the distribution of the bookings.

Usability enhanced

The new version of the simulator is more usable. The installation process has been eliminated. The only thing that the revenue analyst needs is a thin browser. Beforehand the user had to deal with many components, such as middleware binaries, the mysql database, etc.

Managing the simulator server is very easy, the server administrator jobs has been drastically simplified. Some scripts were created for launching and shutting down the application. If the application fails it can be restarted from the web. It does not require direct access to the server. This is done through the tomcat servlet administration web GUI.


/manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Most Visited WR TW Handbook TW Forim
http://ncegcolnx55:33474/manager/html
Google
Done

Manager

List Applications	HTML Manager Help	Manager Help	Server Status

Applications

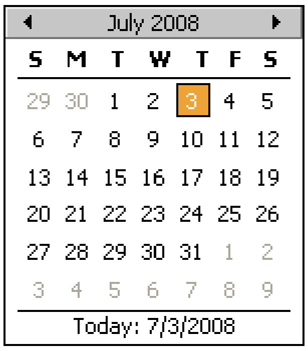
Path	Display Name	Running	Sessions	Commands
/	Welcome to Tomcat	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/BookingSimulator	Booking Simulator GUI	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/host-manager	Tomcat Manager Application	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/manager	Tomcat Manager Application	true	1	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes

Deploy

Deploy directory or WAR file located on server

Context Path (optional):

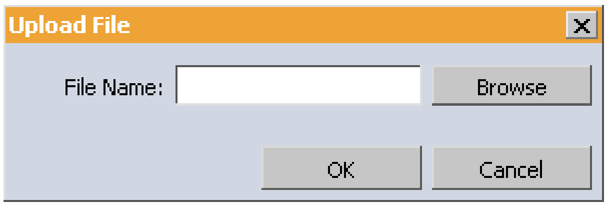
Apart from the installation of the simulator there have been some other enhancements. Smarter controls/components have been introduced like a calendar selector for a flight date and an automatic file up-loader. Before, the date had to be typed as the Amadeus standard for a flight date: YYMMDD and the historical data uploading was done manually on the server side.



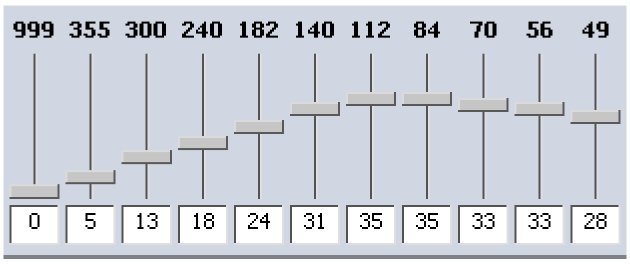
July 2008

S	M	T	W	T	F	S
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Today: 7/3/2008



One of the major UI enhancements was the implementation of a new way to set the maximums and the minimums. Before, it was done manually with text fields. If one wanted to shut down a class one had to set a 0 to all the text fields. In the new versions there are some sliders to do that. Moreover they adapt automatically to the users input.



Several simulations can be done from the same window. Each one is saved in a tab and one can navigate from one to another in order to make comparisions. This was not possbile before.

The last relevant new feature is the sortable table colums. It is interesting for the analyst to sort the classes by obtained revenue, or by number of bookings.

Portability and reusability enhanced

These are the properties that a good architecture and design implies. Thanks to the new 3-layers architecture, it would now be very easy to plug in another frontend. A modification in the business logic does not impact the data or the presentation layer.

Beforehand the local database was running on MySQL. Migrating to Oracle has been really easy thanks to the isolation of the data layer.

The low level design with a high level of abstraction also improves these software properties. The presence of inheritance and interfaces makes the code easy to extend and adapt. New functionalities can be created easily.

Maintainability enhanced

A good architecture and design also help in the maintainability. The main improvements are:

- Exception handling: The exceptions are correctly treated in the layer to which they correspond. The error codes have been factorized. Special screens have been designed for printing exceptions to the user. The end user can easily view any exceptions thrown, either using the simple error view, or the advanced view if they need to see the full stack trace.
- Logging interface: The logs are handled with the Java Log Handling interface that the JVM offers by default. It allows defining different granularity in the log files (Debug, Fine, Info, etc). There is a log for every kind of service: Server,

Simulator, EDIFCAT, Oracle/MySQL, etc. This makes it is easy to detect where the problems come from.

- Configuration interface: All the parameters of the simulator can be configured without compiling any code. The variables are stored in text files. When there is a parameter change the simulator automatically detects it and changes. For example the local database connection parameters can be defined in the db.conf file, where the logs are stored can be configured in the log.conf.

Performance Enhanced

The business logic has also been optimized for example, the message scheduling algorithm, reducing the number of sent messages. Also the way of performing bookings has been substantially improved. Before, two messages were required to make a booking: The availability request and the End of transaction request. Now it is possible to make it directly with one.

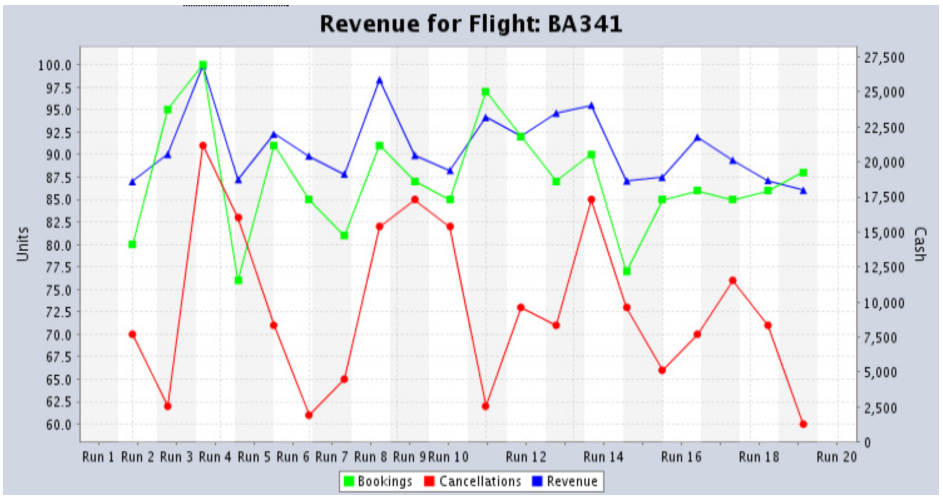
These changes resulted in an improved simulator performance, because the bottleneck in the simulator is the inventory response delay: As the simulator now sends fewer messages, it runs much faster (2-3 times faster).

Comparison table with a load of 250%:

	Time per run	Message per second	Data exchange speed
Before	5m 02s	3.8 msg	4.15 kb / s
Now	1m 46s	10.62 msg	4.07 kb / s

Copy of the performance manual test found in the appendix:

Flight: BA 341 16/06/08 **Load:** 250 **Runs:** 20 **Booking policy:** Local (Random)
Server: APL DEV **Date:** 23/06/08 at 10:14



Time:

Total: 35m 25sec
Messages per second: 10.62
Kbytes per second: 4.07
Time per run: 1m 46s
Time per load % unit: 0.45 s

Messages:

Packet	Count	Total size	Average size
All	22584	8664 Kb	393 B/msg
Sent	11292	3934 Kb	357 B/msg
Received	11292	4730 Kb	429 B/msg
IINVRQ	517	104 Kb	206 B/msg
IINVRR	517	1627 Kb	3223 B/msg
IFLIUQ	516	1221 Kb	2224 B/msg
IFLIUR	516	73 Kb	145 B/msg
IEOTUQ	10259	2609 Kb	260 B/msg
IEOTUR	10259	3030 Kb	302 B/msg

10259 IEOTUQ divided into 8800 sells (bookings) and 1459 cancellations.

Message description:

IINVRQ: flight Structure Retrieval
IFLIUQ: Booking limits clear / change
IEOTUQ: Make Booking / Cancellation

Conclusions

At the end of the internship, the project was presented to the RMS team. The binaries: installation and documentation have been delivered to all members of the team. When the RMS product will be finished, the revenue analyst and testers will be able to work with it and the simulator will be exploited to its full potential.

The system has met all of the functional and non-functional requirements. Working with a cutting-edge tool has made me realize the importance of investing in new technologies to achieve cost reductions.

I am proud of the results that ThinWire has given, making the application robust and responsive. I have liked the idea of programming a web application in a single language. The idea of following the Swing event and component model is really good and intuitive. I think a very good improvement that could be done in ThinWire would be the compliance with the Java Abstract Windows Toolkit (AWT). Then applications that are already compliant with this standard (as Swing applications) could be migrated easily to a web interface.

The project also gave me lots of skills and experiences. First of all, this project helped me understand how to work in a big international development company with development sites spread across the world. Before my internship with Amadeus I had been working for some low-scale companies. Issues like having 9 different test environments for each application (development, integration, user acceptance, production...), daily non-regression tests for all functionalities were new things for me and it was a great experience to understand why they were needed and how they worked. I have worked both at high and low level: High level by designing the development plan, low level by having to code all the program and writing installation and configuration scripts in Linux machines, as well as the deployment and servers setting-up.

Beside the technical skills, I also learned the organizational issues of a big company: The communication between different sites of a worldwide company, between departments, top-down communication, the operational issues, and much more things that made this internship a really great experience that I recommend to everybody.

To end with, I had the opportunity to work in a multicultural environment that was an enriching experience. As well as getting to know people from different nationalities, I had the opportunity to improve my English and to learn French.

Glossary

Airline Yield Management System	Also known as Revenue Management System. Calculates and provides the flight controls to Altéa Inventory and gathers statistics. This system may also be called the Optimizer.
Airport Code	The 3-letter IATA identifier for an airport.
Altéa Inventory	The new inventory system developed by Amadeus to replace legacy airline inventory systems such as BABS and QUBE RS13. It includes the Inventory, Schedule, Reference Data, Seat, Reaccommodation, and MIB servers.
Altéa Inventory GUI	This is the Graphical User Interface used by users to communicate with the Altéa Inventory sub-systems.
Amadeus System	The Amadeus computer system that travel agents use to display schedules, availability, book seats, price the itinerary, and generate the tickets. Also referred to as the Amadeus reservation system, central system or distribution system.
Booking Class	Usually defined by a one-letter code that identifies the kind of ticket restrictions that apply, for example, Advance Purchase and Non-Refundable. Therefore, booking classes are attached to segments. Subclasses are defined by additional rules and constraints, such as the location of the requestor and the flight designator.
Class Availability	Defined at the segment/subclass level, it is a Seats Equivalent availability calculated from the class controls only. This availability is used to cap the sum of Cumulative UPR and Net Revenue Availability in the calculation of Segment Availability under Revenue Controls for all (sub)classes with MAX controls. For flights under Class Control the Net Class Availability will simply be a net version of Segment Availability.
Departure Airport	The airport from which the aircraft last departed using the same flight number.
Destination Airport	The ultimate intended termination airport of a flight.
EDIFACT	Electronic Data Interchange for Administration, Commerce and Transport is an agreed message structure for exchanging data between systems.
Effective Capacity	Based at a leg-cabin level, it is defined as the Operational Capacity plus the Total Adjustment plus the regrade counter for the cabin. The Effective Capacity is the leg-cabin capacity used by <<Re-Calculate availability>> (e.g., to calculate the Availability Pool).
Flight Designator	An airline code, a flight number, and possibly a suffix that allows a user to identify a flight.
Flight Owner	The operating carrier for the flight.
GDS	Global Distribution System, offering services from different providers: airlines, car, rail, or cruise companies. See Amadeus System.
GUI	Graphical User Interface. A windows-based application. See Altéa Inventory GUI.
IATA International Air Transport Association.	An association of international airlines that provides services to airlines, i.e. assigns airline codes, and authorizes agreements between airlines and travel agents for international ticketing.
Inventory Server	The Inventory server actor is the Altéa Inventory subsystem responsible for the inventory of flights on their operating dates.
Leg	A non-stop journey between a departure airport and an arrival airport.
Market	A market is a group of geographical information defined in MIB and used in the definition criteria in Reference Data.
Marketing Carrier	When describing a flight, segment, airline, carrier, this refers to a selling company under its name but with another airline operating.
NGI	New Generation Inventory, code name for Altéa Inventory.
No-show	Passenger with a booking who does not come to the check-in.
Operating Carrier	The carrier that physically controls the flight.
POS	Point of Sale

Route	A sequence of legs.
Segment	A saleable journey, involving a single flight designator, between a board point and an off-point.
Yield	An estimate of how much revenue an airline gets from a sale of a ticket.

Appendix A: Manual test cases

The target of the manual test is to be able to detect bugs in the simulator that cannot be detected automatically. To detect that kind of flaws the human interaction is needed. The manual tests are organized with scenarios and expected results. To pass a manual test the scenario has to be followed and the obtained results compared with the expected ones.

Parameters

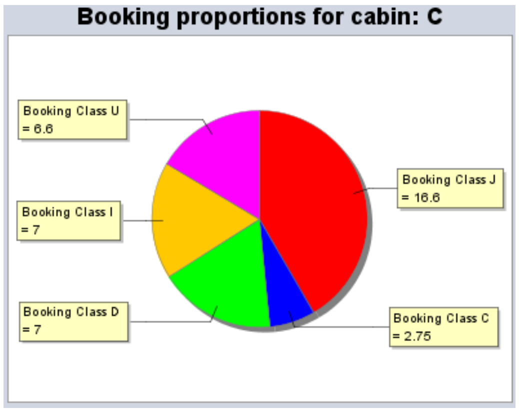
The *flight date* is the main parameter. This is the flight to simulate. The flight date structure can vary on many aspects so that we consider a static flight structure object of the test.
The *trial-number* is a parameter that will not vary the results of the simulation. It is just a value to make the results more reliable. So that, 5 trials numbers is a suitable amount of simulations to be done per each configuration.
The *load* is a key value to charge the flight. It is object to changes.
The *limits policy* and the *limits* itself is the key parameter. It can be set many different ways so that the group of cases is needed.
The *historical demand data* and is the main source of information. In the majority of the test this data is fixed.

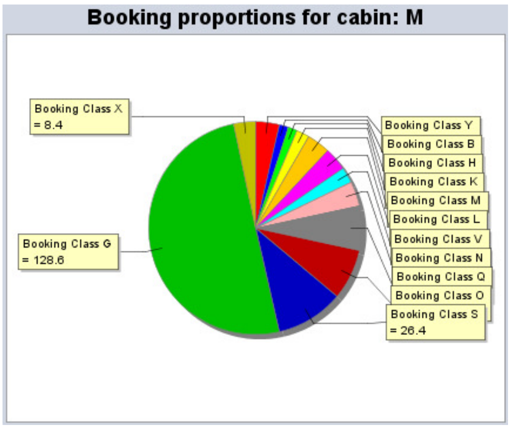
Scenarios

Booking proportions

A simulation is done to check the distribution of the classes booked. No limits are applied and the load is 200% to assure that all the classes are full. To see that the proportions are correct the historical data has been counted.

Flight: BA 341 10/06/08 **Load:** 200 **Runs:** 5 **Booking policy:** Unlimited





Comparison simulated data / historical data

		Bookings historical	% historical	Bookings simulation	% simulation	% Error
Cabin C	Class C	340	6.84	2.7	6.77	1.06
	Class D	981	19.73	7	17.54	11.10
	Class I	934	18.79	7	17.54	6.63
	Class J	1990	40.03	16.6	41.60	3.93
	Class U	726	14.60	6.6	16.54	13.26
	Total	**4971**	**100.00**	**39.9**	**100.00**	0
Cabin M	Class G	18321	47.58	128.6	50.31	5.75
	Class S	4617	11.99	26.4	10.33	13.86
	Class O	2768	7.19	19.6	7.67	6.68
	Class Q	2166	5.62	17.4	6.81	21.02
	Class N	1367	3.55	9.6	3.76	5.80
	Class V	1138	2.96	6	2.35	20.57
	Class L	1781	4.63	9	3.52	23.87
	Class M	1595	4.14	9.2	3.60	13.10
	Class K	513	1.33	5	1.96	46.84
	Class H	653	1.70	3.6	1.41	16.94
	Class Y	1717	4.46	9.4	3.68	17.52
	Class X	1086	2.82	8.4	3.29	16.53
	Class B	482	1.25	3.4	1.33	6.27
	Total	**38507**	**99.21**	**255.6**	**100.00**	0.79

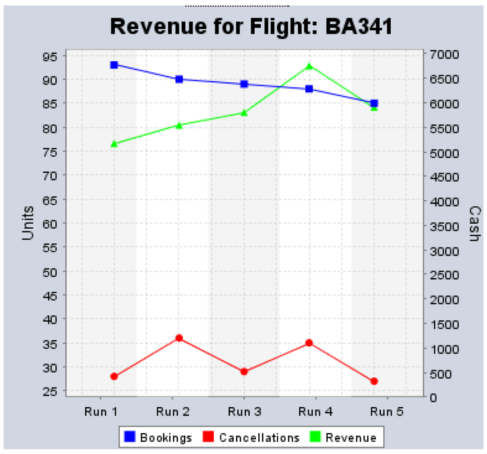
Conclusions

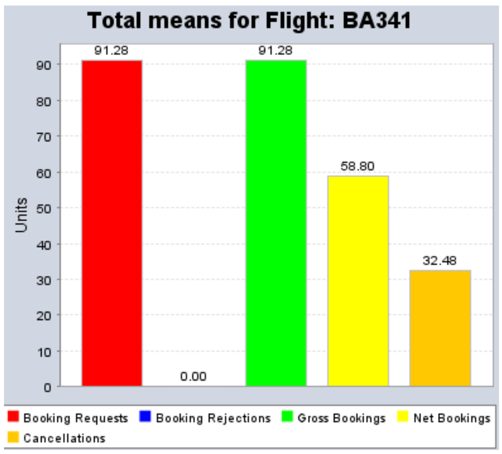
The results are the expected and the booking class proportions follow the initial distribution. That information is extracted from the error ratio calculated from the differences of the historical and simulated tan per cents.

Load factor change

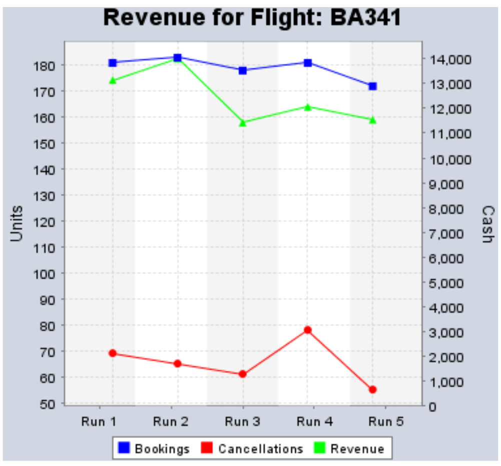
The aim of this test is to change the load percentage to see the increasing of the rejections. The booking limits are set to unlimited. When all the cabins become full the rejections will start to come up.

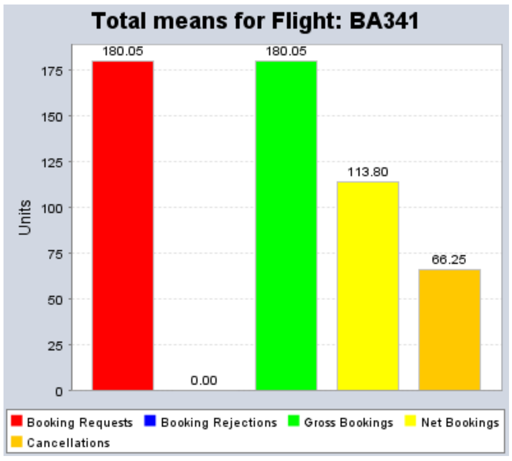
Flight: BA 341 10/06/08 **Load:** 50 **Runs:** 5 **Booking policy:** Unlimited



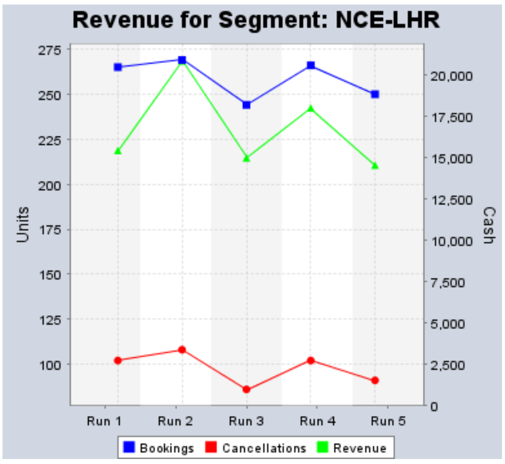


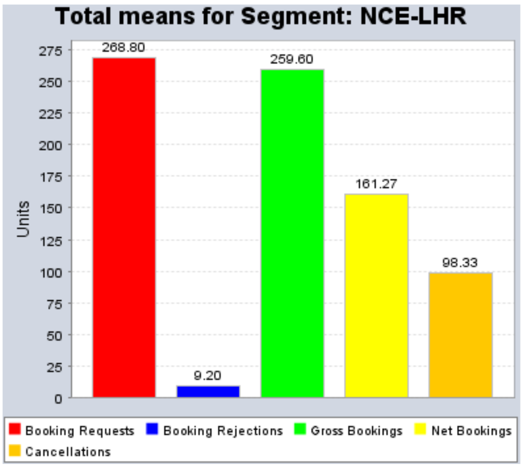
Flight: BA 341 10/06/08 **Load:** 100 **Runs:** 5 **Booking policy:** Unlimited



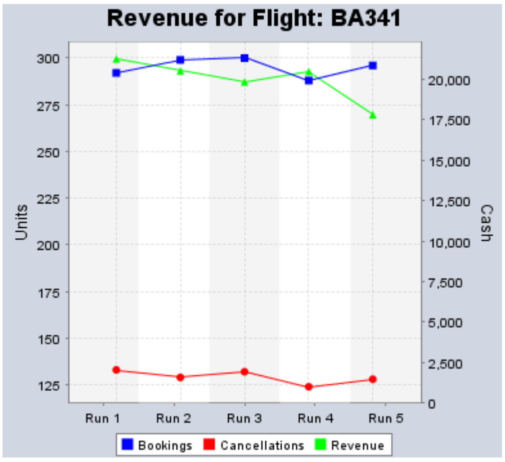


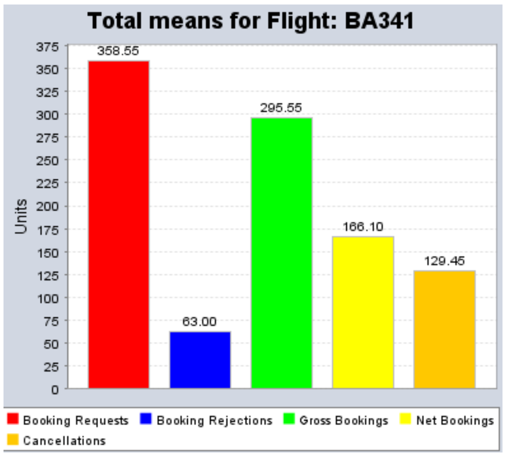
Flight: BA 341 10/06/08 **Load:** 150 **Runs:** 5 **Booking policy:** Unlimited





Flight: BA 341 10/06/08 **Load:** 200 **Runs:** 5 **Booking policy:** Unlimited





Conclusions (table)

Load factor	Requests	Rejections	Gross Bookings	Net Bookings	Cancel-lations	Real load
50%	91	**0**	91	59	32	32%
100%	180	**0**	180	114	66	63%
150%	268	**9**	260	161	98	89%
200%	358	**63**	296	166	129	92%

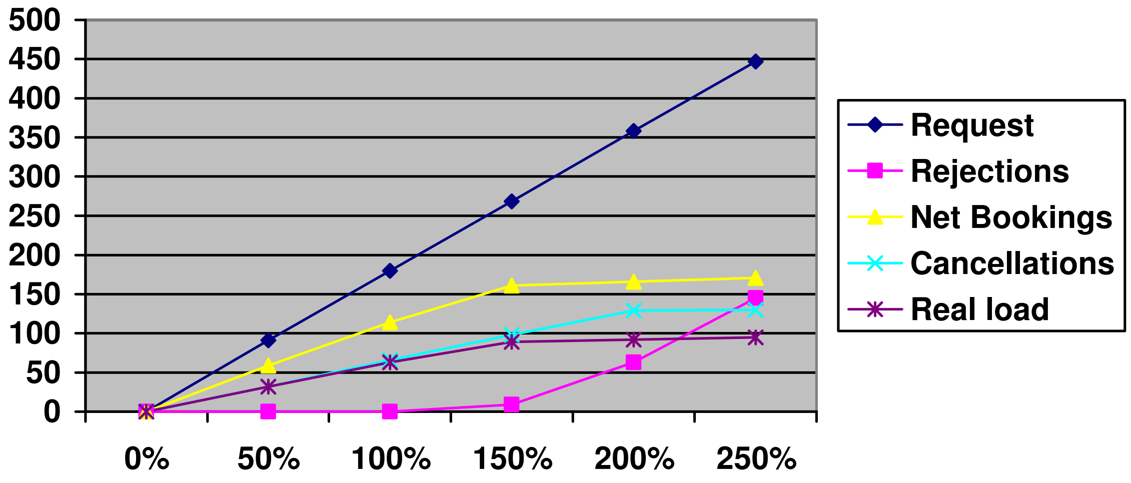
Increasing the load, the rejection rate increases as well. It doesn't increase that much because there are many cancellations and these keep some seats avaiable.

The capcaity of the flight is 179 places divided into 2 cabins C (35 seats) and M (144 seats). Taking a closer look is possible to observe that the cabin C is never filled at 100% while the cabin M is always filled. That's because the cabin C is a bussiness cabin and the fares are higher.

Conclusions (chart)

Analysing the data obtained changing the load factor one realizes that everything has sense.

- The **requests** are a fixed value that depends directly from the load factor, so, it is linear.
- The **rejections** start increasing when the cabins start to be full. When the net bookings are close to the availability (179 seats) the rejections start increasing.
- The **net bookings** stop growing linearly when they are close to the availability.
- The **cancellations** curve keeps similar and proportional to the net bookings.
- The **real load** is a logarithmic curve.



Cancellation proportions

In order to validate the correctness of the cancellations let's make a comparison between the historical data and the simulation results. The data from the previous two scenarios has been used.

Flight level cancellations

If we count the bookings that has been cancelled from the raw data and we divide it per the total of bookings we obtain a cancellation rate of 70%

Load factor	Gross Bookings	Cancellations	Cancellations Rate	Error Rate (over 70)
50%	91	32	35%	50%
100%	180	66	36%	48%
150%	260	98	37%	47%
200%	296	129	43%	38%

With the flight level cancellation study is not possible to detect what is happening but apparently, the cancellation rate seems to be much lower than the original 70%. The error rates are very high.

Booking class level cancellations

Let study the cancellations for each class in a load of 200% and 10 runs

		Cancels Historical	Bookings Historical	Cancels Historical	Cancels Simulate	Bookings Simulate	% Cancel	% Error
C	C	297	637	46.62	1	3	33.33	28.51
	D	798	1779	44.86	3.5	8	43.75	2.47
	I	735	1669	44.04	3	7.5	40.00	9.17
	J	1323	3313	39.93	6	18.5	32.43	18.79
	U	314	1040	30.19	4	8	50.00	65.61
	Total	**3467**	**8438**	**41.09**	**17.5**	**45**	**38.88**	**5.37**
M	B	367	849	43.23	1.5	3.5	42.86	0.85
	G	13947	32268	43.22	45.5	130.5	34.87	19.32
	H	488	1141	42.77	2.5	5.5	45.45	6.27
	K	373	886	42.10	2	3	66.67	58.36
	L	1270	3051	41.63	6.5	12	54.17	30.14
	M	1165	2760	42.21	5.5	9.5	57.89	37.15
	N	1138	2505	45.43	2.5	10.5	23.81	47.59
	O	1289	4057	31.77	13	18.5	70.27	121.17
	Q	1204	3370	35.73	6	12	50.00	39.95
	S	2909	7526	38.65	8	23.5	34.04	11.93
	V	895	2033	44.02	3	6.5	46.15	4.83
	X	482	1568	30.74	3.5	7.5	46.67	51.82
	Y	1311	3028	43.30	8.5	9.5	89.47	106.65
	Total	**26471**	**64675**	**40.93**	**108**	**252**	**42.85**	**4.69**

The error rate is the correlation between the simulated information and the historical data. Is calculated like:

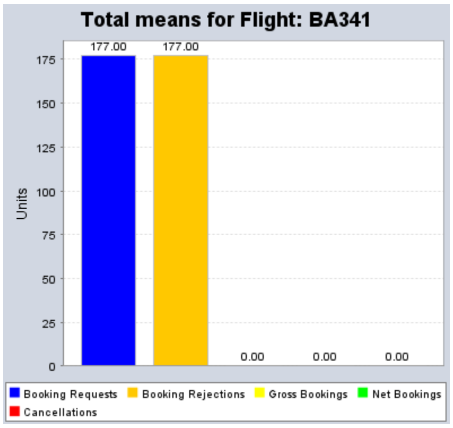
$$h \equiv historical\_cancel\_ratio$$
$$s \equiv simulated\_cancel\_ratio$$
$$error = 100 \cdot \frac{|h - s|}{h}$$

The total error ratio is around 5% and it can be considered as acceptable. There are some high error-ratios due the randomness. Ones compensate others and give a meaningful average.

All the classes shuted down

Shut down all the classes. The MAX Limit is set to 0 for all the classes. Any booking should be produced.

Flight: BA 341 16/06/08 **Load:** 200 **Runs:** 5 **Booking policy:** Local
Booking MAX: All classes 0 **Booking MIN:** All classes 0

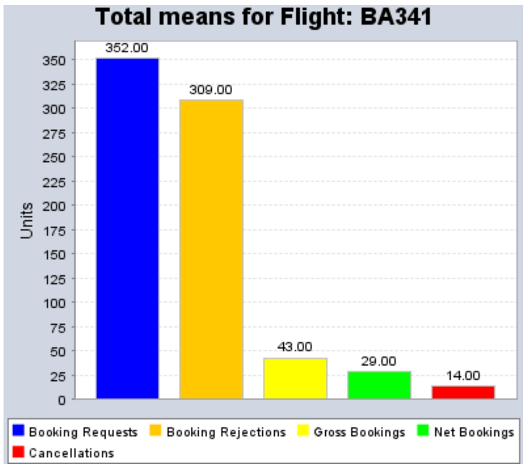


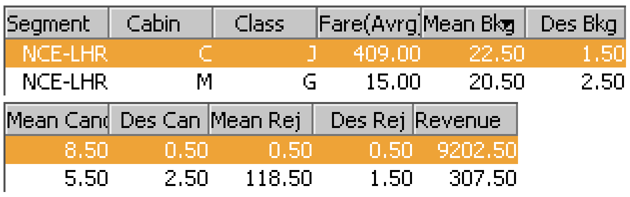
The result of the test is satisfactory. No bookings neither cancellations has been done.

One class per cabin

Shut down all the classes except one for each cabin. The bookings should be just done to these classes. A lot of rejections should appear because the classes cannot accept bookings. The bookings should not be greater that the sum of the maximum of each class.

Flight: BA 341 16/06/08 **Load:** 200 **Runs:** 5 **Booking policy:** Local
Booking MAX: All classes 0; **J,G=15** **Booking MIN:** All classes 0





Segment	Cabin	Class	Fare(Avrg	Mean Bkg	Des Bkg
NCE-LHR	C	J	409.00	22.50	1.50
NCE-LHR	M	G	15.00	20.50	2.50

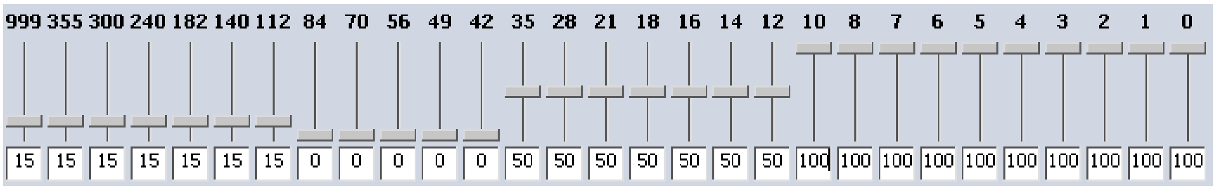
Mean Can	Des Can	Mean Rej	Des Rej	Revenue
8.50	0.50	0.50	0.50	9202.50
5.50	2.50	118.50	1.50	307.50

The results are satisfactory. The Class G has been filled completely (never more than 15 bookings). The class J has been booked according to the demand curves. The limit of 15 has not been exceeded. A total of 29 net bookings have been accomplished (of a maximum of 30 allowed).

Limits change during the time

All the classes are shut down except one. This one has different values depending of the time.

Flight: BA 341 16/06/08 **Load:** 200 **Runs:** 1 **Booking policy: Local**
Booking MAX, MIN: All classes 0 exepct class G that has been set like the following.



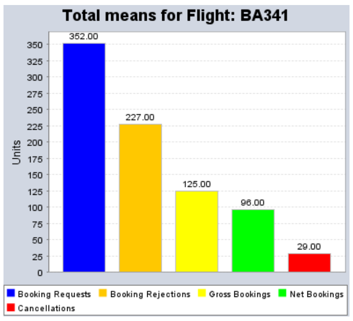
999	355	300	240	182	140	112	84	70	56	49	42	35	28	21	18	16	14	12	10	8	7	6	5	4	3	2	1	0
15	15	15	15	15	15	15	0	0	0	0	0	50	50	50	50	50	50	50	100	100	100	100	100	100	100	100	100	100

The booking limit change messages have been sent at this time. The first message is sent at the DTD 182 because is when the first booking is performed. The simulator can detect when there has been a change on the booking limit and optimize the schedule.

```
Time: 999     Message: CleanBookings Accepted: true

Time: 182     Message: BookingLimitChange   Accepted: true
Max: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:15|X:0|
Min: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:0|X:0|
                                        . . .
Time: 70      Message: BookingLimitChange   Accepted: true
Max: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:0|X:0|
Min: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:0|X:0|
                                        . . .
Time: 28      Message: BookingLimitChange   Accepted: true
Max: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:50|X:0|
Min: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:0|X:0|
                                        . . .
Time: 8       Message: BookingLimitChange   Accepted: true
Max: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:100|X:0|
Min: J:0|C:0|D:0|R:0|I:0|U:0|Y:0|B:0|H:0|K:0|M:0|L:0|V:0|N:0|Q:0|O:0|S:0|G:0|X:0|
```

All the message the simulator sends are saved in a log. This table has been written extracting data from this log.



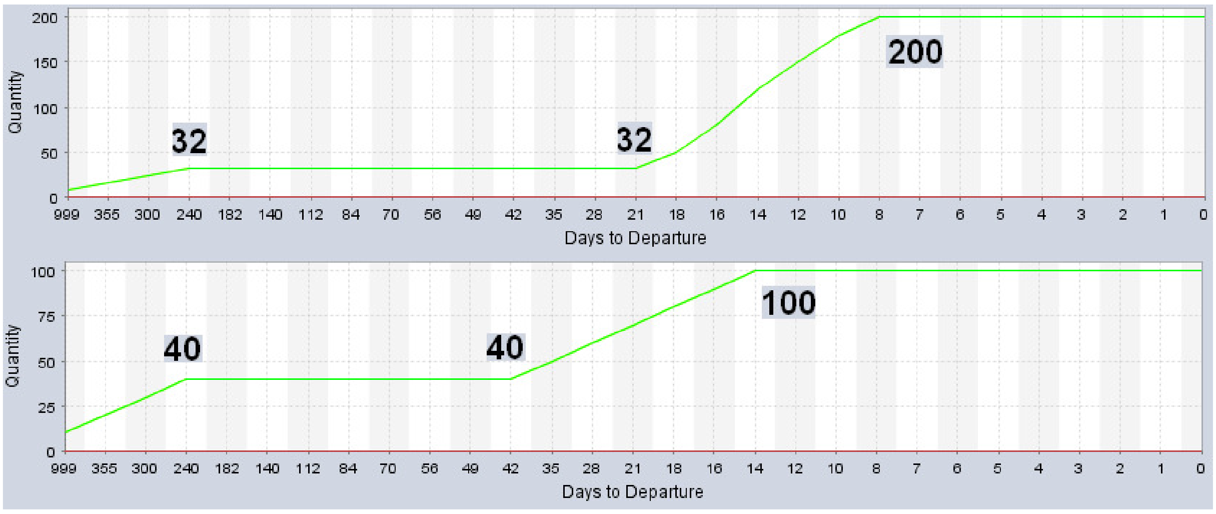
	999-83	84 - 34	35 - 9	10 - 0
Gross Bookings	12	0	53	56
Cancellations	1	0	14	10
Net Bookings	11	0	38	47
Cum Net Booking	**11**	**11**	**50**	**95**
Availability	4	-11	0	4

In the first period there are no rejections. The bookings are below the limit of 15. In the second period the class is shuted but there were already some bookings. In the third period the demand is cuted to 50. In the forth period the demand is freed again and reaches 95 bookings. The response of the simulator seems to be the expected.

Historical data source

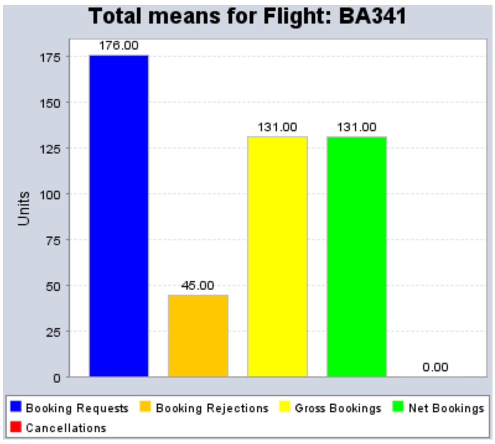
The demand data is reset to 0 for all the classes except 2. The cancellations are eliminated.

Flight: BA 341 16/06/08 **Load:** 100 **Runs:** 5 **Booking policy:** Unlimited
Demand data: No demand except for the classes M-X and M-G described below.

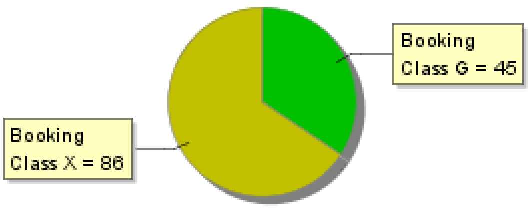


Results

Both classes are in the same cabin. The cabin has an availability of 131 seats. The load is 100% so 176 bookings will be done (availability of all the cabins). The number of rejections equals the availability of the other cabin: 45.



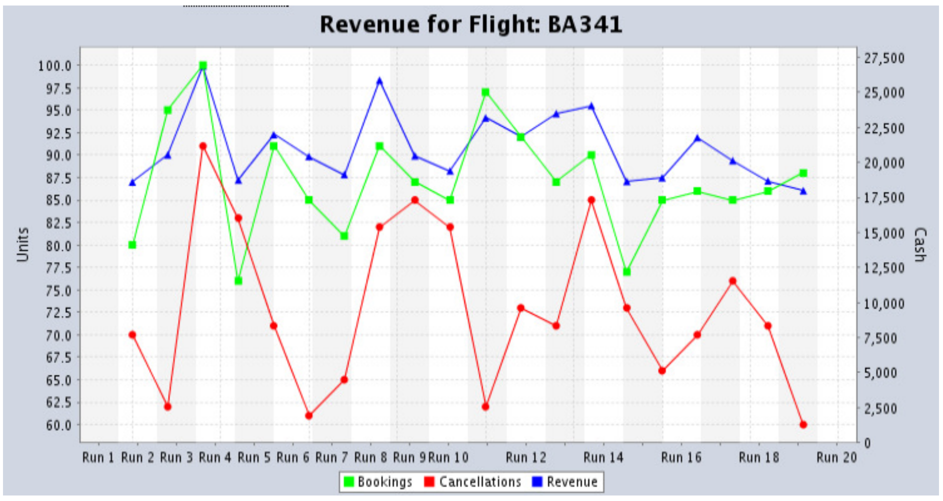
DTD Range	M-X	M-G
999-0	86	45
999-42	14	8
41-21	4	5
8-0	81	33



Performance

This test is just to have some statistics of simulation speed. The time of the simulation will be calculated as well as the size of the messages sent. The local limits are randomized to send a change of the limits for each DCP.

Flight: BA 341 16/06/08 **Load:** 250 **Runs:** 20 **Booking policy:** Local (Random)
Server: APL DEV **Date:** 23/06/08 at 10:14



Time:

Total: 35m 25sec
Messages per second: 10.62
Kbytes per second: 4.07
Time per run: 1m 46s
Time per load % unit: 0.45 s

Messages:

Packet	Count	Total size	Average size
All	22584	8664 Kb	393 b/msg
Sent	11292	3934 Kb	357 b/msg
Received	11292	4730 Kb	429 b/msg
IINVRQ	517	104 Kb	206 b/msg
IINVRR	517	1627 Kb	3223 b/msg
IFLIUQ	516	1221 Kb	2224 b/msg
IFLIUR	516	73 Kb	145 b/msg
IEOTUQ	10259	2609 Kb	260 b/msg
IEOTUR	10259	3030 Kb	302 b/msg

10259 IEOTUQ divided into 8800 sells (bookings) and 1459 cancellations.

Message description:

IINVRQ: flight Structure Retrieval
IFLIUQ: Booking limits clear / change
IEOTUQ: Make Booking / Cancellation

Appendix B: Automated test cases

Connectivity

The connectivity tests are necessary to assure the transfer of information between all the sources. The main two sources are the local database and the NGI connection. The authentication and codification of messages are assured.

NGI Connection

Test Case	Description	Assertions/Fails	Dependencies
Normal Server	Tries to open a connection to a server providing host, port, sign, office id and duty code. Sends a Flight Structure Request Message.	Asserts the connection is opened. Asserts there isn't error in the message transmission.	EDIFACT / Flight Structure Test.
Test Server	Tries to open a connection to a test server providing host and port. No authentication is provided. Sends a Flight Structure Request Message.	Asserts the connection is opened. Asserts there isn't error in the message transmission.	EDIFACT / Flight Structure Test.

MySQL Connectivity

Test Case	Description	Assertions/Fails	Dependencies
Connection	Connects to the local database with the parameters configured on the local database. Obtains a statement from the database.	Asserts the statement is created correctly.	None

MySQL Database Structure.

Test Case	Description	Assertions/Fails	Dependencies
Structure	Open a connection and tries to make a SELECT for each table in the database. The SELECT has all the expected fields of each table.	Fails on exception. For example, if a field on the select doesn't match the structure of the table.	Connectivity / MySQL Connectivity Test.

EDIFACT

The simulator involves different EDIFACT services. Each one is tried independently and its impact on the NGI is checked.

Booking Clean

Test Case	Description	Assertions/Fails	Dependencies
Send Message	Sends a flight Structure Request. If there are no bookings a successful booking is done. A Reset Bookings Message is sent. Sends a flight Structure Request.	Assures that the bookings before the reset are positive and after are equal to zero.	EDIFACT / Flight Structure and End of Transaction Tests.

Booking Limit Change

Test Case	Description	Assertions/Fails	Dependencies
Send Message	A fixed max and min is set for all the classes. Then a Flight Structure Request is sent and checked if the INV limits have changed. Special emphasis in unlimited and 0 limit.	Fails on an illogicality sent by NGI. Assures the expected values are the real ones.	EDIFACT / Flight Structure Test.

Date change Request

Test Case	Description	Assertions/Fails	Dependencies
Send Message	Reset the flight bookings and the limits. Makes a date change request to N days before today. A booking is made.	Fails on exception. Manual assertion.	EDIFACT / End of Transaction Test.

Is not possible to automatically check the time and date of the NGI server. The trick is to connect directly to the NGI Oracle database and check manually. The test is to check that the date of the last booking corresponds to the change of date we have done.
The SQL query is:

```
SELECT creation_date_and_time FROM inv_booking_status ORDER BY ASC LIMIT 1;
```

That might not work because the database structure might be changed.

End of Transaction (Sell, Cancel)

Test Case	Description	Assertions/Fails
Sell (Booking)	Clean bookings. Make one booking for each class. Save the booking DIDs. Query the Flight Structure.	Assert equals the number of successful bookings computed, and the ones obtained using the flight structure message. Checks that every DID for each booking is different and doesn't not exist in a previous booking.
Cancellation	Cancels all the bookings created previously. Query the flight structure.	Assert that all the cancellations are successful. Asserts that the sum of bookings using the flight structure request message is zero.
Over Book	Reset bookings and limits. Does random bookings to all the classes. It stops when the number of successful bookings equals the number of rejections. Normally the number of messages sent is around: availability * 2. Saves the successful bookings DIDs.	Fails if error or if the number of message that are sent exceeds the limit of: availability * 3 messages. This is a limitation measure in order to keep the test under control.
Over Cancel	Flight Request Message. Count the bookings. For each DID in the list send 2 cancellations. Count the successful and the failed cancellations. Flight Request Message.	The number of bookings at the beginning has to equals the number of successful and failed cancellations. At the end the number of bookings has to be 0.

All the test cases depend on each other, are prepared to be executed sequentially. Moreover they depend on the *Connectivity / NGI Connection* test and on the *EDIFACT / Booking Clean* and *Flight Structure* tests.

Flight Structure

Test Case	Description	Assertions/Fails	Dependencies
Send Message	Sends a Flight Structure Query Message and creates its corresponding Flight (using the classes defined in the BOM).	Checks the correctness of the created BOM. Assures there are (at least one) segments, legs, seg cabins, leg cabins, booking classes, subclasses. In each subclass check the bookings and the existence if the properties of it (nestings, yields, etc).	Connectivity / NGI Connection

SQL

The queries to the local database are tested separately and its impact is checked.

Building BOM from Local Database

Test Case	Description	Assertions/Fails	Dependencies
Demand Data	Creates a Flight Object and fills it from data extracted from the Demand Data table on the local database. Load the demand curves for each booking class.	Asserts the correctness of the BOM. Check that the curves are correctly loaded.	Connectivity / SQL DB Structure
Raw Data	Creates a Flight Object and fills it from data extracted from the Raw Data table on the local database.	Asserts the correctness of the BOM.	Connectivity / SQL DB Structure

Generate Demand Curves from Raw Data

.

Test Case	Description	Assertions/Fails	Dependencies
Generate curves	Takes the RAW Data as input stream and generates the demand curves and cancellations for each booking class.	Check that the curves (demand and cancellation) are correctly generated and consistent. By default the curves shall not be unconstrained.	SQL / Building BOM from local DB

Load/Save Booking Limits

Test Case	Description	Assertions/Fails	Dependencies
Load booking	Sends a Flight Structure Query message. For each booking class each correspondent booking limit time vector (min and max) is loaded from the database.	Assures that the min and max limit exists for each DCP.	EDIFACT / Flight Structure
Save bookings	Modifies the min and max limits for each class and for each DCP randomly. Save it to a local database. Load it again.	Assure equals the randomly chosen limits and the ones loaded at the end of the process.	Load booking (sequential)

Various

Tests such as booking scheduling, cancellations matrix correctness, probability drawn, dates transformation, etc, are placed here.

Date Format Tests

Test Case	Description	Assertions/Fails	Dependencies
Formatting	Tests the date conversion functions located in the utils package. The formats ara the Java date object, Amadeus flight data format (ddmmyy) and readable format (dd/mm/yyyy). The test has some hard-coded dates and its equivalents.	Assert equals the transformations and the hard-coded equivalents.	SQL / Building BOM from local DB

Random drawn

Test Case	Description	Assertions/Fails	Dependencies
Randomize check	Draw random numbers using the simulator randomize functions. Discretisize the values and put them in a hash map.	Check that all the values follow a normal distribution. Sometimes can fail.	None

Cancellation table matrix

Test Case	Description	Assertions/Fails	Dependencies
Generate curves	Load the BOM from raw data. Generate the probability cancellation curve from the cancellation curves.	Check the saneness of the probability matrix. Find the diagonal.	Building BOM from local DB / Raw Data

Booking/Cancellation/Limits Schedule

Test Case	Description	Assertions/Fails	Dependencies
Schedule	Perform a simulation and cancel it before sending messages, so, the planning of the message is done.	Assure the planning has as many bookings as defined in the load of the flight (taking in consideration the rounding assuming a margin of error of 2%)	Simulation

Simulation test

A full simulation with different booking limits policies is done. With the results obtained some automated simple checks are done. The synchronization between the local and remote results is checked here.

Full Simulation

Test Case	Description	Assertions/Fails	Dependencies
Booking Policy Unlimited	Runs a simulation with the standard parameters and the booking limits policy set to unlimited.	Assert no errors. Manual correctness checked.	Everything
Booking Policy None	Runs a simulation with the standard parameters and the booking limits policy set to none.	Assert no errors. Manual correctness checked.	Everything
Booking Policy Local	Runs a simulation with the standard parameters and the booking limits policy set to local.	Assert no errors. Manual correctness checked.	Everything

Simulation Automatic

Test Case	Description	Assertions/Fails	Dependencies
Simulate	Runs a simulation with 1 run, 200% load factor and booking limits unlimited. Send a Flight Structure Request Message.	Compare the requests, bookings, cancellations, rejections counted by the simulation logic with the same results obtained from the inventory.	Everything

Appendix C: Local database schema

Local curve database

booking_limits_max, booking_limit_min, demand_curves

These three tables have the same structure. First two store the limits imposed by the client. Last table contains the demand curve for each booking class. The demand curve table is generated from the raw_data and yield_data tables.

Field Name	Format	Description
airline_code	Varchar(3)	Code of the airline Ex: BA=British Airways
flight_number	Int(3)	Number of flight
board_point	Varchar(3)	Departure airport code
off_point	Varchar(3)	Arrival airport code
cabin_code	Varchar(1)	Cabin code (Business, First, Tourist)
booking_class	Varchar(2)	Booking class identifier
dmd1	Int(11)	Value on dcp 1
…	…	…
dmd29	Int(11)	Value on dcp 29

raw_data

This table contains the booking and cancellations done for a flight in a concrete date. It also provides the information of when the booking was done and when it was cancelled, if it was.

Field Name	Format	Description
airline_code	Varchar(3)	Code of the airline Ex: BA=British Airways
flight_number	Int(3)	Number of flight
board_point	Varchar(3)	Departure airport code
off_point	Varchar(3)	Arrival airport code
cabin_code	Varchar(1)	Cabin code (Business, First, Tourist)
booking_class	Varchar(2)	Booking class identifier
subclass	Int(1)	Subclass identyfier (not used)
creation_dtd	Int(3)	Booking creation days to departure
cancellation_dtd	Int(3)	Booking cancellation days to departure
flight_date	Varchar(9)	Date of the flight depature
nature	Int(1)	1 = Booking, -1 = Cancellation

yield_data

This table contains basically the revenue made by the airline for a concrete cabin of a flight. That is stored in the field *yield_value*. This is a simplification because in the real case the yields used should depend on the *flight_date*, *class*, *subclass* and *dtd*.

Field Name	Format	Description
airline_code	Varchar(3)	Code of the airline Ex: BA = British Airlines
flight_number	Int(3)	Number of flight
board_point	Varchar(3)	Departure airport code
off_point	Varchar(3)	Arrival airport code
cabin_code	Varchar(1)	Cabin code (Business, First, Tourist)
yield_value	Int(6)	Revenue of the booking