Generic Testing Tool

Design of a generic testing tool for a hospitality system

ADRIEN ENAULT

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2011
Generic Testing Tool

Design of a generic testing tool for a hospitality system

ADRIEN ENAULT

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011
Supervisor at CSC was Alexander Baltatzis
Examiner was Stefan Arnborg

TRITA-CSC-E 2011:033
ISRN-KTH/CSC/E--11/033--SE
ISSN-1653-5715
Abstract

This thesis examines several solutions to create a generic testing tool to help developers in compare different algorithm in an hospitality system. The thesis examines several possibilities in order to make a tool that is easily extensible.

The tool was implemented in C++ and Java/JSP for the user interface part. The developed system sticks to the standard architecture of its environment.

A testing sequence is defined, and among its variants, the simplest possible sequence, testing one algorithm at a time, is found to be the most effective for speed, flexibility and because it allows to test a wide range of criteria. A system named fetcher system is designed to provide maximum extensibility and its performances are gauged, and found to be satisfactory in terms of extensibility. The use of pairs of key/value to store the results of the test is deemed to be more practical than use of plain text or custom database structure.
## Contents

1 Introduction ........................................... 1

2 A few important notions ................................. 3
   2.1 Structure of a hotel ............................... 3
      2.1.1 Property ..................................... 3
      2.1.2 Room types .................................... 4
      2.1.3 Allotments and rates .......................... 4
   2.2 Floating Inventory .................................. 5

3 Presentation of the environment ......................... 9
   3.1 Central Reservation System ........................ 9
   3.2 Architecture of the Amadeus CRS ................. 10
      3.2.1 Overall architecture ......................... 10
      3.2.2 Architecture of a back-end ................. 10

4 Specifications .......................................... 13
   4.1 Preliminary specifications .......................... 13
   4.2 First decisions ..................................... 14
      4.2.1 Flow of booking ............................... 14
      4.2.2 Architecture .................................... 15
      4.2.3 Business Object Models and Use cases ....... 16

5 Testing Algorithm ...................................... 19
   5.1 How does the test take place ....................... 19
      5.1.1 Flow of bookings .............................. 19
      5.1.2 Choice of algorithm to test .................. 20
   5.2 Sequence of a test .................................. 20
   5.3 Fetchers ............................................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.1</td>
<td>Presentation of the fetchers</td>
<td>23</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Examples of implementation</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Messages and Database</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>Organization of the tests</td>
<td>27</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Database</td>
<td>28</td>
</tr>
<tr>
<td>6.2</td>
<td>Messages</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>GUI and compilation of results</td>
<td>31</td>
</tr>
<tr>
<td>7.1</td>
<td>Managing the database</td>
<td>31</td>
</tr>
<tr>
<td>7.2</td>
<td>Presenting the results</td>
<td>33</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Combination</td>
<td>33</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Presentation</td>
<td>34</td>
</tr>
<tr>
<td>7.3</td>
<td>Architecture</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>Results</td>
<td>37</td>
</tr>
<tr>
<td>8.1</td>
<td>About the performances of the PSF system</td>
<td>37</td>
</tr>
<tr>
<td>8.1.1</td>
<td>About the flexibility of the PSF system</td>
<td>37</td>
</tr>
<tr>
<td>8.2</td>
<td>Presenting the results</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>45</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the software industry in general, innovation is often the key to success. Therefore a system must constantly be imbued with new features. And during their development, testing is critical, as the earlier a mistake is discovered, the cheaper it is to correct. It is made throughout their whole development and after, and must answer various questions such as: does the new functionality work correctly, does it make the system crash... Early benchmarks can also be done in order to determine if the functionality is really worth the investment, as well as to convince potential customers of the power of the system. [1]

Testing can be time and energy consuming. Developers may have to adapt existing testing tools to their needs with various results. These tools are not always very fitting for this task. They might be cumbersome to use, and they sometimes don’t present the results in a readable manner.

The internship was done in the hospitality section of the company Amadeus, which is currently developing a complete system for the hotel industry. This system is able to take bookings, manage inventories of rooms and services, the rates applied etc. The Inventory team of this section is developing a new and unique feature they call the Floating Inventory, whose principle will be explained later. And with it arose the need of a tool which could test this feature, but it came with a few challenges: first, the feature was not yet finished (actually, when the internship started, it was barely started, and only a simplified version was working when the internship finished). Second, no criterion of testing was clearly defined. They primarily wanted something to test the performances of their new feature, but were unsure about the details. Finally, the system was in commercial use, so testing the new features should not interfere too much with the rest of the system.
The goal of the internship was finally to design, code and test a testing tool able to help a developer of the Inventory team to easily test the criteria he wants on any part of the system they were working on. This implied the testing tool had to be generic and flexible to be able to test about every part of the system on about any criteria. It also had to be able to present the results clearly. The resulting tool was of course bound to be very specific to this particular system, but this thesis might contain a few interesting ideas for someone who is developing something similar.

The back ends of the system were written in C++ and had access to an Oracle SQL database, the GUI was written in JSP/Java and used the framework Webwork. XML messages are used for the communication between the GUI and the back ends as well as between different back ends. The architecture of the system will be detailed in the first parts.
Chapter 2

A few important notions

In order to understand the subject at hand, it is important to see how hotels are organized and represented in the system. In this part, we will also explain the principles and the challenges of the Floating Inventory as being able to test it was the main motivation for the design of the testing tool.

A Central Reservation System (CRS) is a very complex system which can be divided in several components [2], each of whom are dedicated to a task. Among these components, we are going to talk about two: the inventory, which is responsible for answering to availability requests among other things (has a particular hotel enough rooms for this booking? how many?), and the rates, which is mainly responsible for computing the price of a reservation. There are many other components, such as the configuration component, where a user can define the structure of a hotel, but the testing tool never interacted with them.

2.1 Structure of a hotel

2.1.1 Property

The first thing we are going to see is the central concept of property. It corresponds to a physical hotel containing several sets of rooms of different types. A property has an address, it offers different services, applies its own rates. A property is represented in the system by an alphanumeric string of six to eight characters.

Technically, a hotel is part of a chain and a brand. A chain is a big entity of the hospitality industry. It generally owns several brands. A brand is basically a set of hotel who all share the same name. For instance, Hilton Worldwide is a chain which owns several brands, Hilton being the most known of them [3]. Both the chain and
the brand are represented by an alphanumeric string of two to three characters. Hence, a hotel in particular can be uniquely referred to by a code similar to this example: CHA/BRA/HOTEL1.

In real life, some hotels are not part of a brand or a chain, for instance when they are owned by small proprietors, but they are generally not customers for hotel reservation systems, who are mostly designed for big companies.

2.1.2 Room types

A property itself is structured. It has several room types, each one having a capacity. Two rooms of the same room type will of course generally share common characteristics. The criteria used to differentiate the different room types may vary. It might the type of the bed (double, simple), the view the room has (street, garden, sea), the floor it is on or the size or anything else. There are a few parameters all rooms of a room type share. For instance, the number of guests they can accommodate, or the number of extra beds they can have.

It is the finest granularity possible in the inventory: it is impossible to distinguish two rooms of the same room type. The rooms of a given room type in a hotel is basically represented by the capacity (total number of rooms available for booking) and the availability (number of rooms which have not yet been booked).

A room type is represented by an alphanumeric string of 3 characters. SNG for single bed room or DBL for double bed room for instance, but the structure of a hotel can of course be far more complex than that.

One of the main challenge is that the structure may (will) change over time. For instance, a hotel on the coast may offer less beds during the winter than during the summer, and a hotel intended for business trips may offer less rooms during the week end.

2.1.3 Allotments and rates

As is, the system would not be neither very useful nor very complex, as it does not give to the proprietor the possibility to manage to who he sells what or at which price. There come the notions of allotments and rates. Theses notions are closely related.

A rate is a set of rules which lets the system compute the price of a given reservation. It takes in account several criteria, the obvious ones being the room type, the number of guests, but may also take in account the number of rooms still
available, for instance, as an almost full hotel is generally going to sell its rooms at a higher price. It can also take in account the period of the year (is it the high season?) or of the week.

A property is divided in several allotments, which can overlap or not. They define a set of rooms to which a given rate apply. Or more precisely, all the booking of a given rate will arrive in the same allotment. We are not going to see too much in details what are the different types of allotments. They just define the maximum part of a hotel you want to with a rate.

For instance, let’s say a hotel wants to make a special offer to attract customers. He is going to create a rate with a set of rules which gives lower prices than the usual one, but he does not want to sell his whole hotel at this price. So he will bind this rate to an allotment which, for instance, contains at most a tenth of his hotel. Therefore, he can configure a special offer, but is sure it won’t sell his whole hotel. Another example: you want a preferential rate which applies only to groups, but again, you may not want to be able to sell your whole hotel at this rate.

The structure of the allotments of a hotel can be very complex, as you can define the capacity of these allotments on a room type basis, some are included in others, some cannot overlap and some can, and some are valid only for reservations made a few days in advance.

As the structure is bound to change over time or to be modified, managing and computing the availability of the allotments was one of the biggest challenge the inventory team of Amadeus faced.

### 2.2 Floating Inventory

Now we can define a hotel and its capacity, its rates and the parts of the hotel to which it can apply. But the system is still very rigid. Let us say we have a businessman who wants to spend a night in a hotel which has two types of single bed rooms: the ones with a wonderful view on the Baltic sea, and the other ones with a view on the cute little garden of the hotel. Our businessman does not care: he just wants to sleep. With the system as it is, he is forced to choose, in which case he might choose, for instance, the garden view, because it is cheaper and he cares for his business expenses. If this room type is full, (actually, to be more precise, if the the allotment which corresponds to the rate which corresponds to the booking), he may want to look if the sea view type is also full or if he can book one. This is not practical and the businessman might not think to look for another room type.
And if there are a lot of similar room types, it might soon become discouraging. The only solution would be to merge these two room types, but then you could not charge your customers who want the sea view.

This is where the so-called Floating Inventory feature comes in. It lets the hotel proprietor define some kind of meta room types. In the previous example, you would still have the garden view and the sea view single bed room types, but you would also have a single bed meta room type which regroups both of them. And let us say you also have garden view and sea view double bed room types, you could regroup the sea view double and single bed room type in another meta room type, for the lonely meditative customers who absolutely want a view on the Baltic sea, even if it means taking a room with a double bed. Then, you have two distinct meta groups who share a room type.

Once the booking has been made, the room type assigned to it might still change if necessary. It is "floating", hence the name. Indeed, in order to manage the Inventory in an optimized way, bookings made on a meta group should not be directly assigned a room type, as shown in the example of the figure 2.1.

![Figure 2.1. A floating booking might still change of room type once booked so the system can take more bookings](image)

The hotel of the figure 2.1 has a very simple structure. Two room types, Single bed Sea View (SSV) and Single bed Garden View (SGV), a capacity of one room for each and a meta room type, Single bed, which encompasses them. The first booking targets the meta room type. Let us say the system first puts him in the SGV room, because this way, the hotel still have his most expensive room available. A second booking over a period which overlaps the period of the first booking, made by a cost-wary customer which want the SGV room. The system then has to move the first booking in the SSV room (which is compatible with the first booking’s customer’s wishes), or it would lose the booking. The problem is very simple in this example, but in a more complex structure, with a dozen of room types and meta room types which can overlap one another, this becomes very difficult.

The Floating Inventory, whose principle is quite simple, but the implementation
2.2. FLOATING INVENTORY

quite complex, comes with a question. With a standard inventory, if there is enough room in a hotel for a given booking, you are assured that the customer will be able to keep his room for the length of his stay. As shown in the example of the figure 2.2, a floating inventory might take more bookings if the customer accepts to change of room during his stay.

![Figure 2.2](image)

**Figure 2.2.** If the customer accept to change of room during his stay, the third booking can still be placed

The example described in the figure 2.2 takes place in the same hotel than the last example. If the first booking targets the SSV room type for the first night and the second targets the SGV room type for the second night, the third booking targeting the S meta room type for the first two nights can still be placed if the customer accepts to change of room during his stay.
Chapter 3

Presentation of the environment

A significant part of the internship was dedicated to the understanding of the environment where it took place. As the final results were heavily influenced by this environment, it is important to give the context. We will talk about the system and its role, then give an overview of its architecture.

3.1 Central Reservation System

A Central Reservation System (CRS) [2] is, as its name suggest, a mean to centralize all the bookings and the management made on the hotels of a company. Ultimately, all the reservations customers make on any channel arrives on the CRS. For instance, should a customer make a reservation on a websites specialized in low cost travels, in a travel agency or directly at the hotel, the CRS will be made aware of it.

Thanks to this centralization, a company using the system can easily perform large scale resource management. Such a system will be necessarily divided in several components which will interact.

Generally, messages coming from the exterior, such as a booking request, of course, but also a request to add or modify a hotel in the system, are standardized, to ensure several system can dialog together. The OTA specification is a widely accepted standard [4] (developed by OpenTravel Alliance), and was the most commonly encountered standard during the internship. It defines a certain number of couples of messages (query - response) for the travel agencies, and were used by the whole system to communicate with other systems. But the communication between the GUI and the back ends also used EDIFACT based messages (actually, they were XML messages wrapped in an EDIFACT message), EDIFACT being a
message format such as the XML format, more compact but less readable, developed by the UN [5]. The system often used custom XML internal messages for the communication between the back ends.

Although their meaning are theoretically defined by the OTA standard, the different interpretations and uses of these messages among the different systems in the world often made necessary the creation of a specific component in order to interact with another system, at least for the Amadeus system. This flexibility is made on purpose, and is supposed to reflect the flexibility of the industry [6]. This can only add to the complexity of a CRS.

3.2 Architecture of the Amadeus CRS

3.2.1 Overall architecture

The Amadeus CRS has a fairly simple and standard global architecture: a GUI dialogs with a set of back-ends which in turn dialog with a database [8]. Each back-end is able to perform a small set of actions, and therefore do not run the whole system. For instance, one back-end is responsible for taking bookings, while another is responsible for computing prices. For speed reasons, there can be several instances of the same back-end, which then run in parallel, but each instance of a back-end is mono-threaded.

When the GUI sends a message to the back-ends, it transits through the Amadeus Open Transaction Framework (OTF) whose responsibility is to determine to which back-end it must forward it. OTF automatically takes care of the routing of the query, but also the response, simplifying greatly the handling of conversations. It also takes cares of load balancing when several back ends of the same kind are available.

As shown in the figure 3.1, there exists a tool, named TT Server (for Test Tool Server), which lets the user inject a message into the back-ends. This tool, usually used by the Amadeus developers for regression testing, is also quite handy to send and receive messages when you do not have a GUI, or when you want to simulate a part of the system, as it can act as a back-end by answering with preprogrammed messages.

3.2.2 Architecture of a back-end

The back-ends follow a strict architecture organized in layers.
3.2. ARCHITECTURE OF THE AMADEUS CRS

The global architecture of the Amadeus CRS looks like a three tier architecture, with the OTF component routing the queries from the users to the right back end, and being able to make some load balancing.

The Middleware layer takes care of the messages sent through the network (for example, coming from the GUI, or another system). It contains the auto generated message parser classes for EDIFACT and XML messages (all you have to give is the corresponding XML Schema for the messages, at least for the XML part). Once parsed, the messages are directed toward the corresponding service entry (typically, there is one service entry for each kind of message). The Service Entry coordinates the actions needed to get the answer and translate it back into a message. The AdapterIn translates the message into a Business Object Model (BOM) that the system can understand, while the AdapterOut translates a BOM into a message.

A back end can be directly controlled by another part of the system, or can make use of a C++ package (such as the Job Framework, detailed in part 4.2.2) without having to resort to messages through the use of an Internal Service Proxy.

The BOM represents the task to be accomplished, as well as the results. The use case class contains all the logic needed to perform the action.

For example: the XML message RetrieveAvail arrives in the system and is taken care of by the corresponding Service Entry class after having been parsed by the corresponding DSC class. The Service Entry then translates it into a BOMRetrieveAvailRequest object using an AdapterIn, runs the right use case with the BOM object as arguments. This use case then performs the actions required and at the same time fills a BOMRetrieveAvailAnswer object, which is returned at the end of the use case. The Service Entry then translates this BOM into a message using an AdapterOut and then sends this message back to the requester. The BOMs are
CHAPTER 3. PRESENTATION OF THE ENVIRONMENT

Figure 3.2. The architecture of a back end is organized in layers. All the logic is normally contained in the use cases.

supposed to only hold information, while the use cases are supposed to contain all the logic.

The External Service Proxy is a bit the same thing than the Service Entry, but is here to make requests to another Back End.

The DOM classes (Data Object Model) modelizes the objects of the database. They are automatically generated given the description of the various entries in the Database. They help the developer to manage the objects in the Database by providing an interface to retrieve or store entries.
Chapter 4

Specifications

The goal of the internship was quite loosely defined, and one of the first task was to determine what the testing system would exactly do. As it was common to be able to design a subsystem by a three letter word, this testing system was dubbed Performances Statistics Fetcher (PSF), because of its purpose and its operating principle.

4.1 Preliminary specifications

As stated in the first chapters, the goal was to design a generic testing tool for the inventory team, which would be able to test, among other things, but principally, the floating inventory. The inventory team did not know exactly what kind of tests they wanted to be able to run, but the idea was to make benchmarks about the performances of their system.

There were a few requirements about the PSF. First, it had to be easily extensible: one should be able to quickly add a test in the system - and generic: the system should be able to run a wide range of tests, different in nature. For example, in the case of the floating inventory, you might want to test the answer time to see if it is fast enough, or you might want to see if it accepts enough bookings to be useful, or if the system accepts that a customer might change of room during his stay, you might want to make sure he does not have to change of room every night. In short, if a developer thought of a relevant test, he or she should be able to implement it easily.

It also had to be able to store its results into a database. This supposes the possibility to manage it, at least to be able to retrieve and delete entries, but in a way which would keep the data consistent (the system should not allow the user to
accidentally erase some data which would render some other data useless). This also
supposed some kind of organization of the data, to make sure a user can retrieve
its data in a potentially big database.

Though having a GUI was not mandatory, it was welcome, and as we will see in
the part 7, the GUI part finally came to play an important part in the PSF system,
not limited to just the presentation of the results.

4.2 First decisions

Very quickly, some preliminary decisions were made about how the PSF should
work, which then became specifications.

4.2.1 Flow of booking

The general idea behind the PSF was simple: it would send a flow of bookings to
the system, analyze the answers, then store and send back the answer.

The first question that then comes to mind is: how do we generate the booking
flow? Is it something that is generated externally by the tester (or more probably,
a tool), or is it something the PSF generates internally? As a tool called Hotel
Booking Simulator (HBS) had already been developed, which, given a few param-
eters modeling the behavior of the customers, could generate an arbitrarily long
sequence of bookings, this question was answered quickly, but the question was still
interesting to consider.

Indeed, having an internal generator can be very interesting. First of all, for
the most obvious point, it makes the tests easier, as you no longer have to generate
the flow of bookings yourself. It is therefore practical for quick tests. Besides, by
using a pseudo-random generator, and returning the seed to the user, it ensures
that you still can run the exact same test more than once, if you give the possibility
to the user to specify the seed [7]. It is noteworthy to to say that as a user do no
longer have to give to the system the flow of bookings, the PSF is now responsible
of making sure the flow of bookings is correct, while without, it is the user (or another
tool) who ensures that. Actually, as seen in the part 2.1, the structure of a hotel
may vary: from a property to another, there may not be the same room types, for
examples. The PSF must be made aware, in a way or another, of the structure of
the property it targets.

Furthermore, though the user no longer has to give the flow of bookings to the
PSF, he or she now has to give it new parameters, to ensure the generated flow of
4.2. FIRST DECISIONS

bookings will be relevant. These parameters can be, for example, the ratio of the number of bookings for each room types, the length of the simulation, etc. The HBS is a tool dedicated to the generation of such flows, and presents clearly in a GUI all the parameters it needs, with graphs to visualize and change them, making it very practical. To be really practical, the internal generator of the PSF should present similar features, which was a bit out of scope, and integrating the HBS was too complex due to its architecture (it was only an elaborated GUI). Using the HBS as it was, or with a few modifications, was deemed the best solution.

4.2.2 Architecture

It was also decided that the PSF would stick as much as possible to the architecture of the rest of the system, especially concerning the back end, which would follow the architecture presented in the part 3.2.2. It has the advantages of letting the developer code faster, because he has models to refer to, and also can use more easily the tools already available for the same reason. For example, using OTF, the tool that routes the messages in the system, is far easier if you can refer to some existing code to see how it works.

Initially, the PSF would only consist of the back end, and a GUI would eventually be plugged to it to present the results. To simulate this flow of booking, the PSF needs to send the booking requests to the whole system, get the response and process it. This is the role of the back end. To avoid to have to run the whole system which can be very heavy, the PSF uses internal messages: rather than sending a booking request to the whole platform (a message that would be defined by the OTA specifications as explained in the section 3.1), it directly sends an internal booking request to the Inventory back end. By doing so, it bypasses the front end, the need for security back ends, authentication, etc. making the process of testing a feature a lot more lightweight, but cannot make complex queries, as he is limited to the functionalities of the Inventory back end. During the development was expressed the need to retrieve the prices of the bookings made. The PSF then also had to be able to communicate with another back end, Rates, to make pricing request.

It also quickly became clear that it was very impractical that the back end had to wait for the end of the test to answer to the user or the GUI. The tests run by the PSF potentially imply hundreds or thousands of booking requests, and maybe as many pricing requests, which can obviously take a lot of time. If the PSF had to wait the end of the test to answer, chances are that it will time out. Instead, the back end will use a daemon to make the requests, and the back end itself will only
treat the messages coming from the user (or the GUI), and push the test requests into the daemon’s queue. This has several advantages, you can then run several daemons to make several tests simultaneously, and the back end is still available to treat other requests, for, by example, getting the results stored in the database, following the progress of a test or starting another one. An internal tool called Job Framework was available for this. Information about the task it has to accomplish takes the form of a sequence of strings "marshaled" (or serialized) which is then stored in the Job framework’s database.

The final overall architecture of the PSF is summarized in the figure 4.1. The HRP and INV elements represent the two external back ends responsible for respectively Pricing and Booking Requests.

![Figure 4.1](image.png)

**Figure 4.1.** The architecture of the PSF system follows the architecture of the whole system (OTF is not represented but implicitly used). It uses a daemon which performs the tests and communicates with the inventory (INV) and rates (HRP) back ends.

To sum it up, the back end gets the test it must run, acknowledge it to the user (or the GUI), then at some point, the daemon sends the bookings, gets the answers, extracts the information it needs, and then stores the results in the database.

### 4.2.3 Business Object Models and Use cases

There are mainly 4 use cases, each pertaining to a task the PSF system can make. To each use case is associated one or several Business Object Models to represent
4.2. FIRST DECISIONS

the request and/or the result.

The Use cases are:

Insert Job : when a user gives a new test to run to the system, the PSF system does not directly performs it. Instead, it puts all the relevant information in the Job Framework’s database, which then will start a daemon to perform it. This way, the system is available for other queries, such as putting another test in the database, or database management. A BomJob is used to store all the information needed to run the scenario. It contains a list of BomBooking and a list of BomFetcher. As the only answers possible are success or fail, no BOM for the answer is needed. See figure 4.2 for the detailed design of the BomJob class (see part 5.3 for more information about the fetchers).

Retrieve Data : a BomRetrieveRequest contains the scenario names (maybe the subscenario names) of the entries the user wants to get. If empty, the PSF system retrieves everything. The GUI always retrieves everything. A BomResult is filled during the use case. It is a collection a BomScenarioResult, themselves being a collection of BomSubScenarioResult, themselves collection of BomJobResult, themselves collection of BomFetcherResult, themselves collections of BomKeyValueResult. See part 6.1.1 for more information about the data organization in the database.

Delete Data : The BomDeleteRequest object is very similar to the BomRetrieveRequest, except you can target jobs and fetcher results. The database can be corrupted by this request if the wrong entries are deleted (for example if you delete the wrong fetcher result), as the back end does not provide any protection. No answer is needed for the same reason than for the insert job use case.

Play Job : This is the main use case. It comes in two flavors: one which retrieves rates and one which does not. This use case is not directly use by the PSF, but by its daemon. This use case performs a single booking at a time, and each fetcher is responsible for storing the data pertaining to the test it must perform. Once the last booking has been sent, the daemon asks the BomFetcher to store the information in the database.
CHAPTER 4. SPECIFICATIONS

Figure 4.2. The BomJob and the BomFetcher classes. The BomJob contains all the data necessary to perform a test (bookings, target property and the criteria (fetchers)). Each BomFetcher contains all the information it needs to compute the results for this test.
Chapter 5

Testing Algorithm

The main challenge was of course determining the logic behind the tests. The general idea was set, but still left a lot of possibilities and questions. In this section, we are going to decide of the fate of the flow of bookings, and introduce the system of fetchers.

5.1 How does the test take place

5.1.1 Flow of bookings

We know that we give to the PSF a flow of booking, which is supposed to simulate the customers behavior. To limit the size of this booking, we will cut down the information for each booking to a minimum. We need a check-in and check-out date (arrival and departure), the rate plan, the room type, the property (which means the chain code, brand code, property code) and the number of guests and their category (children, adult, senior etc.). To simplify the simulation, we drop the number of guests and their category, and assume that a booking is always made for only one adult. Although simulating the number of guests might be relevant, we deemed that the results given would be independent enough of the number of guests not to implement it. Indeed, the number of guest influences the price of a booking, but not whether or not it will be accepted, as this only depends of the availability.

We will also restrict the number of properties a flow can target to one. Indeed, the main goal was to test the floating inventory, which operates only in one property, so even if we want to test a complex containing several properties, we still can use several flows, each one targeting a different property. Being able to target several properties could be relevant, if, for example, the system we wanted to test had a
CHAPTER 5. TESTING ALGORITHM

recommendation algorithm which offered to the customer another room in a similar property should his first choice be unavailable.

Finally, the flow of booking has the property in its heading, but the booking themselves only contain the check-in and check-out dates, the rate plan, and the room type.

5.1.2 Choice of algorithm to test

We have one system, but possibly several algorithms to test. For example, we might want to compare several algorithms of floating inventory, or we might want to see what are the advantages of the floating inventory over a system without any kind of such optimization. The question is: how do we tell the PSF which algorithm it must choose?

The ideal would be, of course, that we give a parameter to the PSF so he can choose. But when looking closely, this is not really practical. It supposes a lot of work on the system itself so the system can choose too (notably in the messages which would then have to be able to pass this new parameter and all the modifications it supposes in the rest of the system), when the final system won’t have to, and developers generally don’t want to put too much code in a system just for testing purposes, as it can lead to unnecessary errors in the final system. Besides, it does not always make a lot of sense. For example, let us say that your flow of bookings targets a property with no meta room types defined: it would not make a lot of sense to be able to tell the PSF that he should use a floating algorithm, and the opposite would not make sense at all.

So the solution we retained was that a property would be associated to an algorithm, which is easier to implement in the system, and by choosing the property, we indirectly make the choice of the algorithm. This has an immediate advantage: if you just want to test if the floating inventory gives better results than a system without, you don’t have anything to change anything in the system, you just have to make a flow of bookings which does not make use of the meta room types.

5.2 Sequence of a test

The format of the flow of booking being now defined, we can now focus more on how the PSF will use it. To be generic, the sequence of the test should follow something along the lines of the figure 5.1.
5.2. SEQUENCE OF A TEST

```plaintext
initTest()

forall Booking from Bookings
    BmTestAnswer Answer // will contain info gotten from answers
    getPricing(Booking, Answer)
    tryToBook(Booking, Answer)
    fetchInfoFrom(Answer)
end

compileInfo()

saveInfo()
```

Figure 5.1. Sequence of a test in pseudo code. This sequence is as simple as it can get: we send all the bookings, and each time extracts the information needed from the answer. At the end, all the results are compiled and saved.

This sequence, while simple, is quite powerful. Thanks to this, we can easily test a lot of parameters. For example, we can compute the income generated by the bookings - simply by summing the rates returned each time a booking has been accepted, or the number of bookings made in a similar way, or the average answer time - by taking the time when we get the data about the booking and when we get the answers and then by simply compute the difference. This assures us that the PSF will be generic enough, as all the relevant tests the developers of the inventory team thought about could be easily achieved with this.

We can take note of the fact that we always test one property (therefore one algorithm) at a time in a test. We test one algorithm at a time, and at the end we can compare the results between the two algorithms. This was not originally the case. In the case the two algorithms or more can accept the same flow of bookings, for example when a developer wants to compare two different algorithms of floating inventory, testing several algorithms at the same time might be interesting. It makes it possible to compare the two algorithms on a per-booking basis, rather than just on the entire flow of bookings. See figure 5.2.

But it comes with a few drawbacks. For each booking, the system has to send it to several properties. If we want to be able to test several algorithms at a time, we basically have two solutions: we can keep track of the results for each booking for a later comparison, which can be tedious and memory hungry, and this pretty much is the solution where you test only one algorithm at a time, with a few more constraints. Or we must send the booking to the different properties
Figure 5.2. The difference between testing one algorithm (or property) or several at a time is made clearer by this figure. While on the left, the system has to wait for the end of both tests to compare the results (which will be the compiled result), on the right, the system can compare the results after each booking, potentially extracting more information. On the second case, though, the two tests have to use the flow of bookings if this comparison is to hold any meaning.

Around the same time, but this makes parallelization more difficult: the system makes it difficult for a given component to handle several communications at the same time, and sending a booking to one property at a time is no parallelization at all. Besides, treating and organizing the data is more difficult when you have tests such as ALGO1 vs ALGO2, then ALGO2 vs ALGO3, then ALGO1 vs NO FLOATING, than when each test concerns only one algorithm, in which case you can easily compile all the results concerning one algorithm.

Finally, it was deemed that the advantages of being able to test several algorithms at a time did not overcome its drawbacks. By dividing the tests in a one property at-a-time way, it is easy to parallelize them just by running several daemons. Each of these test (that is, a flow of booking sent to one property) is called a job. This choice is further discussed in the part 7.2.1.
5.3 Fetchers

We want the PSF to be easily extensible, so we have to organize how the tests are performed internally. After some reflexion, the fetchers system were designed, inspired by the object approach.

5.3.1 Presentation of the fetchers

A fetcher represents a test. For example, a fetcher might represent the Answer Time test which will be responsible for computing the average answer time during a simulation. The system will initially offer several kind of fetchers as a proof of concept: the answer time, a fetcher which will compute the income that would have been generated by the flow of booking named Income, a fetcher that computes the number of bookings rejected and the total number of days booked named Difference, and a fetcher that computes the occupancy rate of the hotel at the end of the test named Occupancy. A last fetcher named Statistics, whose role is a bit special as we will see later in the part 7, computes some basic statistics about the test, such as the number of bookings that has been rejected because of an error in the system.

When giving the information about the job to run to the PSF system, along with the flow of bookings and the property targeted, a list of these fetchers is specified. It tells the PSF system what the user wants to test. The fetcher statistics is automatically added, as it is needed later.

These fetchers were added for different reasons: the fetcher Difference was added as a test for the PSF system because of its simplicity. All the others but Statistics were added because they are quite different tasks in nature, and therefore show that the system is generic.

Base of a fetcher

All those fetchers derive of the base fetcher class. We are going to see in detail the function of the different methods the base class contains.

Among them, three virtual functions which are re-implemented (or not) in the derived classes and their responsibility is to fetch all the information the fetcher needs. They are called during the use cases as shown in the figure 5.3, and it is their implementation that defines the behavior of the fetcher:

\[
\text{feedBooking(BomTestBookingRQ)} \quad : \quad \text{the BomTestBookingRQ contains all the relevant information about the booking which is going to be sent to the system.}
\]
CHAPTER 5. TESTING ALGORITHM

Thanks to this, the fetcher is aware of the bookings sent.

**feedAnswer(BomTestAnswer)**: the BomTestAnswer contains all the relevant information about the response of the system to the booking: has it been accepted, if yes, at which price, or has it been rejected because of an error in the system.

**compileResult()**: this function notifies to a fetcher that a test is finished, so that it can compile its results, gather the last bits of info, etc.

We will see how we can use these different methods in the part 5.3.2.

The getName() method let the system identify the fetcher, which is important when it must present the information gathered during the tests.

The isRateFetcher() method returns by default FALSE unless it is re-implemented to return otherwise. This way, at the beginning of the test, the PSF knows if he has to retrieve the rates. This way, if no fetcher need the rates (for example because the only thing you want to compute is the answer time of the inventory part of the system), the PSF can avoid to send unnecessary messages and therefore performs the test faster. Furthermore, the developer do not have to configure the rates for his test property and do not have to launch the rates back end, making the testing procedure much easier for him.

The save*() methods are, as their name suggest, here to save the results into the database. The save(JobId, BOManager) method is called by the PSF system for each fetcher (the JobId is the primary key in the database of the test currently run, and the BOManager is the object which lets the user save entries in the database). This method calls the saveIt() virtual method, which is re-implemented in the different fetcher classes and which basically contains calls to the saveInt(string, int), saveFloat(string, float), saveString(string, string) methods, which handily saves pairs of key/value in the database. This way, when a developer creates a new fetcher, he just has to write the saveIt() method, and does not have to manage the database.

5.3.2 Examples of implementation

The figures 5.4 and 5.5 show how, in pseudo code, can be implemented the different fetchers, which means, how the feedBooking(), feedAnswer() and compileResults() methods are re-implemented to perform the test.

In the figure 5.4, the updateAnswerTime(elapsedTime) method just hide a small calculus, and in the figure 5.5, the fetchOccupancy(date) hides a few database
transactions to get, for a given date, the total capacity of the hostel, and the number of rooms that have been booked that day. The rest of the code just keeps track of the period in time relevant for that test (to avoid fetching data at dates where no booking has been made)
CHAPTER 5. TESTING ALGORITHM

Figure 5.4. Pseudo code for the AnswerTime fetcher that computes the average answer time of the system.

```java
constructor:
    this->numberOfBooking = 0
    this->answerTime = 0.0

feedBooking(aBooking):
    this->beginTime = currentTime()

feedAnswer(anAnswer):
    if(! anAnswer.hasErrors) // if no error encountered
        elapsedTime = currentTime() - this->beginTime
        (this->numberOfBooking)++
        updateAnswerTime(elapsedTime)

compileResults():
```

Figure 5.5. Pseudo code for the Occupancy fetcher that computes the day-by-day occupancy rate for the hotel at the end of the test, giving an indication of how full it is.

```java
constructor:
    this->firstValidBooking = true

feedBooking(aBooking):
    this->currentBeginDate = aBooking.beginDate
    this->currentEndDate = aBooking.endDate

feedAnswer(anAnswer):
    if(! anAnswer.hasErrors) // if no error encountered
        if (this->firstValidBooking)
            this->firstValidBooking = false
            this->beginDate = this->currentBeginDate
            this->endDate = this->currentEndDate
        else
            this->beginDate = min(this->beginDate, this->currentBeginDate)
            this->endDate = max(this->endDate, this->currentEndDate)

    // this way, we keep track of the period over which bookings // have been done

compileResults():
    for(date = this->beginDate, date <= this->endDate, date++)
        fetchOccupancy(date)
```
Chapter 6

Messages and Database

We must organize the way the data is organized in the database, so we can retrieve later all the info we need about a particular test. Indeed, for now, the results of the different jobs are stored separately, without any mean to know to what they correspond to, or any mean to regroup them by theme.

6.1 Organization of the tests

Each job is associated to an algorithm, as seen in 5.1.2. So, it would seem natural to be able to regroup all the jobs into one category. This way, we can keep track of the meaning of a job. But if the jobs were only sorted on the basis of the algorithm they are testing, this would still be a bit too rough, as a user could not regroup the jobs by theme. For example, a developer might want to test the respective merits of the ALGO1 and the ALGO2, and to to have a separated proving that using a property configured with the floating inventory will generate more income than one without. So we have two tests, each with two categories, regrouping several jobs each (as you might not want to stick to the results of just one flow of bookings).

So to keep it simple, we add two levels of subdivision. For historical reasons, we have named these layers Scenario and Subscenario. In the first of the two previous example, the Scenario would be called, say "ALGO1 vs ALGO2", and the Subscenarios would be "ALGO1" and "ALGO2". The jobs then belong to one of the subscenario, depending of which algorithm they are associated to.
6.1.1 Database

Now that we have organized conveniently the different jobs, let’s take a look at how they are stored in the database, and more particularly, how the raw results given by the fetchers are stored. Indeed, each job uses a certain number of fetchers (we know that at least the fetcher Statistics will be used), and we have considered mainly three possibility to store them.

**Free form**: each of the fetcher is responsible of how its own database table(s). For each kind of fetcher, one or several tables are added in order to store the results. This solution has the advantage of being without any constraints, therefore you always can organize your database in accordance with the type of data you want to store, which then will be fitting. But this solution has a cost: it requires a lot of fetcher-specific coding, and a lot of database management, in particular when retrieving the data for a specific fetcher.

**Formatted text**: there is a unique table containing strings of characters, each entry representing the results of one fetcher during a job. This implies that the system is able to parse the string of characters and that the string of characters has to be correctly generated by the system. Other than that, the database management is very easy and is the same for all the fetchers, but parsing the string of characters can quickly become annoying.

**Key Values**: the fetchers store their results by using key-values, and these pairs are stored in one or several tables (for example, you might want to have a table for integral numbers, one for floating numbers and a third for strings). The database management is easy as you only have a few tables, you can avoid to have to do too much parsing on the values, but you will certainly have to parse the keys, now.

The solution used at first was the second one, as it was easy to implement, and was sufficient to store the results of the first fetchers implemented. It had the advantage of being easy to read as it (as you could choose the way the data was presented, and therefore was practical for a tool without GUI). But it finally was not deemed very elegant and a bit cumbersome notably for the Occupancy fetcher (the last to be implemented). One of the motivations to change of solution was that the size limit of a VARCHAR2 in an Oracle database is 4000 characters, which can be a problem when a fetcher such as Occupancy has a lot of data to return. It
6.2. MESSAGES

eventually was replaced with the Key/Value solution with a variant: there only is
one table containing strings, and the helper functions of the base fetcher class lets
the developer save a pair with a value of type int, float or string in an easy way. As
the fetchers could have quite heterogeneous data types including but not limited to
the int, float and string types, having three tables for those types when it might not
be sufficient did not seem to be worth the effort of having to manage those three
tables. Finally, the database structure can be summarized by the figure 6.1.

Figure 6.1. Structure of the database of the PSF system. FK stands for
Foreign Key, which basically means, for instance, that each ResultPart entry
belongs to a Result entry.

6.2 Messages

As seen in part 3.1, a back end is controlled using pairs of XML messages. Those
messages have changed a lot of time during the project as the way results are stored
in the database influences a lot the format of the messages. But the goal of the
three pairs of messages remained essentially the same throughout the internship.

The only message we are going to see is also the most important one for several
reasons: first, it is the message used to describe a test to run to the system, second,
it is the only message the user has to understand, as the other ones are used by the
GUI, and therefore a user does not have to know them. A very simple example is
shown in figure 6.2. Each SubScenario node basically encases a flow of bookings.
It can be sent to one or more properties (for example, you might want to test the
respective merits of the ALGO1 and the ALGO2, and thanks to that, you can send
the flow to both with one message), and each property has to be assigned to a SubScenario (here, it is the "Floating" SubScenario, but if two properties had been specified, they could have been associated to "ALGO1" and "ALGO2", for instance). The message also specifies, for each flow of booking, the kind of tests the user wants to perform (here, AnswerTime and Income).

The answer to this message is a simple message specifying the success or the failure of the query, as the back end just puts the job in the Daemon database.

```
<HDP_InvFloatestScenarioRQ ScenarioName="FloatingTest">
    <SubScenario>
        <Properties>
            <Property BrandCode="2X" ChainCode="HRI01" PropertyCode="HRI001"
                SubScenarioName="Floating" />
        </Properties>
        <Results>
            <Result Type="AnswerTime" />
            <Result Type="Income" />
        </Results>
    </SubScenario>
    <Bookings>
        <Booking CheckInDate="2010-05-02" CheckOutDate="2010-05-06"
            RatePlanCode="TS3" RoomTypeCode="DBL" />
    </Bookings>
</HDP_InvFloatestScenarioRQ>
```

**Figure 6.2.** This is a very simple example of the message that lets a user insert a job into the PSF system. In an actual message, there would obviously be a lot of bookings.

The two other pairs of message are of course the DeleteResults query and answer and the RetrieveResults query and answer.
Chapter 7

GUI and compilation of results

The GUI was initially optional, and the back end was supposed to use messages readable by a human. Its role was extended, and its implementation became critical. The GUI then became fully part of the PSF system which would not be complete without it.

As for the back end, we tried to stick to the standard architecture for the design of the GUI. Therefore, several technologies were used to code it: most of the logic was written in Java, the interface was presented as web pages, written using JSP, Javascript and the framework Webwork. The GUI had to be carefully designed, as it had to be as generic and extensible as the back end.

7.1 Managing the database

One of the things you can do with the GUI is to manage the database. It presents in a clear hierarchical way the different scenarios, subscenarios, jobs... stored in the database. From there, you can delete a scenario, a subscenario, a job, or just the result a one fetcher. The GUI prevents the user from corrupting the database by not letting him delete the Statistics fetcher of a job without deleting its associated job and all its content: indeed, as it will be seen in the part 7.2.1, without the results of the Statistics fetcher, the associated job is basically useless for the PSF system. A javascript method automatically untick the checkbox associated to the Statistics fetcher if the associated job is unselected (and it is automatically unselected should any of its associated result be unselected). A user can hide or show the pairs of key/value of a result by simply clicking on its name, so the view is not littered by potentially huge amounts of entries.
CHAPTER 7. GUI AND COMPILATION OF RESULTS

If a job has been inserted in the job framework, the GUI shows its status (0 is not started, 1 is started, 4 is finished and 5 is failed, this code came from the framework).

While being rough, the GUI lets a user safely and easily manage the database.

Figure 7.1. The GUI presents the results stored in the database in a clear way. The "Hide Jobs" button let the user hide the content of the jobs so the view is less littered by all the entries.
7.2 Presenting the results

While being handy, the database management was neither necessary nor the main goal of the GUI. With the back end, we are now able to gather the information needed to make some tests. We now want to compile the data and eventually present it graphically.

7.2.1 Combination

When the user click, in the GUI, on the name of a scenario, the system understands that he wants to see the results of the test of this scenario. The system then has to compile the results he has, which means that, for each of the subscenarios, the system must fusion all the jobs it contains in a meaningful manner. The result of the fusion of two results is to be determined on a per-fetcher basis, and can use the associated Statistic results.

An example: a subscenario contains two jobs. Each job contains, beside the mandatory Statistic result, an AnswerTime result. The obvious idea is then to fusion the two AnswerTime results by meaning the two answer times, weighed by the number of bookings in the job, so a job with more bookings (being supposedly more representative) will have more influence in the final result. If the two jobs are associated to the same algorithm, which should be the case if the tester organized his different tests according to the rules described in the part 6.1, you now have a more accurate answer time for this algorithm.

The system, for each kind of fetcher, has to browse through the jobs of a subscenario, and for each job, get the Statistics result (if it does not find it, it just withdraws the job), then, using the Statistics, it merges the different kind of results it finds.

This is mainly in order to make the combination part of the system more easy and more flexible that the one-algorithm-at-a-time solution was retained. Indeed, the several-algorithms-at-a-time solution presented two main problems during the combination.

First, the data structure tended to be made more complex, and therefore, less simple to combine. Indeed, you often have a table when with the simple algorithm you only have a figure. For example, the Difference fetcher was not only to compute the number of bookings each algorithm accepted, but also, keep track of the number of bookings an algorithm accepted while another did not. This could be useful to determine, for example, if two algorithms performed differently (hence the name of
CHAPTER 7. GUI AND COMPILATION OF RESULTS

the fetcher), if for example, they globally accepted the same number of bookings, but not the same bookings, which could have hinted that the algorithms were specialized, and maybe could be combined for better performances. This turned out to be of little practical use: first, this seemed quite unlikely to happen, and second, there was only one algorithm to test at the moment. So this feature finally deemed counterproductive, as it only made the combination harder (and therefore hindered the extensibility).

The second problem was that it could theoretically prevent the re-use of results. For example, for your first test, your test ALGO1 vs ALGO2. At some point, the ALGO3 is developed (maybe a combination of the first two), and you want to compare it to the first two. Then you immediately see that, for the Difference fetcher, in order to compute the differences, you have to re-run the test with all three algorithms. This further limits the usefulness of this feature.

The solution which keeps track of all the results for each bookings could be implemented to work around these problems, but would require a more work in order to conveniently store and retrieve the results and identify the flows of bookings. This said, this solution has another advantage: some fetchers could then be computed afterwards, depending on what kind of results are stored. Difference and even income (provided the rates are retrieved) could be computed afterwards, but neither the AnswerTime (though this kind of information could easily be made part of the results stored) nor the Occupancy (as it requires to check the occupancy just after the flow of bookings has been sent). This solution has not been explored by this master thesis.

7.2.2 Presentation

Once the PSF system has merged the different results coming from the different jobs, it builds and display one or several graphics, using Fusion Chart (as Amadeus had acquired a license to use it).

The system, during the combination, keeps track, for each subscenario, of which kind of fetcher are used. This way, the system can avoid displaying empty charts, and empty entries in charts. For example, if no subscenario make use of the AnswerTime fetcher, then this fetcher won’t appear in the presentation of the scenario. If some subscenarios, but not all, make use of this fetcher, then only the subscenarios using it will appear. This makes the resulting charts easier to read.

The part 8.2 shows how the system outputs the different fetchers.
7.3 Architecture

The GUI is basically made of two pages. Therefore, for the display part of the GUI, the architecture is organized around those two pages.

The framework Webwork revolves around the concept of actions [9], whose name is quite explanatory. There are three actions: display the content of the database, delete entries in the database and display a scenario (which involves combination of the different fetchers). To each of these actions corresponds one java object which uses several other objects presented in the figure 7.2.

The BomResult actually recovers the BomScenario, BomSubScenario, BomJob and BomFetcher classes, the last one containing a map of Key/Values. The BomCompiledResult recovers the BomCompiledScenario, BomCompiledSubScenario and BomCompiledFetcher classes, the last one being declined for each kind of fetcher. The schema does not show the part which handles the communication with the back-ends.

Figure 7.2. The architecture of the GUI (not showing all the details). It shows the two JSP pages, the different actions and classes and how they interact.
Chapter 8

Results

8.1 About the performances of the PSF system

About the speed of the PSF system, it was never a criterion, but the fact that several tests (if they do not interfere, which means if they all target different properties) can be run simultaneously is a plus. Thanks to the one-algorithm-at-a-time solution, the different parts of a test (jobs) are performed at the same time (as they necessarily target different properties) by several daemons running at the same time. The GUI is fast enough to be comfortable. It was also clear and functional.

The problems brought by the several-algorithm-at-a-time solution overcame the benefits: the only advantage of being able to more finely compare two algorithms is mitigated by the non re-usability of previous tests as seen in 7.2.1, and it made the combination more complex. The third solution (storing all the results for each booking) could solve this problem, but would require a lot of work, and could make the combination of results even more complex.

8.1.1 About the flexibility of the PSF system

About the genericity, the PSF system was deemed satisfactory by the inventory team. Thanks to its simple principle, it was able to perform a wide variety of tests, from computing the average AnswerTime, to the income generated by the bookings, to the Occupancy of the hotel at the end of the run. All the relevant tests the team could think of could be added. The different fetchers, which were added The few features which were abandoned, such as parameters for the test, or testing several algorithms at the same time, were never needed, confirming that doing without was the right choice. The result of the implementation of the different fetcher is
summarized in 8.1.

<table>
<thead>
<tr>
<th>Fetcher</th>
<th>Implementation Result</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistics</td>
<td>Successful</td>
<td>Made necessary by design</td>
</tr>
<tr>
<td>Difference</td>
<td>Successful</td>
<td>Revealed most of the problems with the several-algorithms-at-a-time approach</td>
</tr>
<tr>
<td>AnswerTime</td>
<td>Successful</td>
<td></td>
</tr>
<tr>
<td>Income</td>
<td>Successful</td>
<td>Revealed the need for pricing</td>
</tr>
<tr>
<td>Occupancy</td>
<td>Successful</td>
<td>Revealed a flaw in the architecture of the GUI, which makes adding a test a bit cumbersome</td>
</tr>
</tbody>
</table>

Figure 8.1. The implementation of the fetchers taught different lessons, but all of them were ultimately successful

About the extensibility, the conclusion was more mitigated. As a test, the Occupancy fetcher was added once the PSF system was more or less complete. On one hand, the architecture of the back end was very convenient when it came to adding a test: a developer just had to create and implement a new BOM deriving from the BomFetcher class, then add two lines of code in the FetcherBuilder class (which associates the name of a fetcher to its fetcher class), and that was all. The implementation of the saveit() method was made very easy thanks to the helper functions. On the other hand, adding it into the GUI was not. While it took me about one hour to add the Occupancy fetcher to the back end, it took a whole afternoon (and a bit more) to add it to the GUI. It occurred at this time that the GUI had not been designed carefully enough. The GUI involved several technologies in interaction (JSP, WebWork, Java), which, though they are designed to work together seamlessly, add to the complexity of the design. A different design for the GUI, notably a change in the hierarchical order of the compiled data for the GUI, may help. To add a fetcher in the GUI, you have to modify a lot of different classes, which is not practical.

8.2 Presenting the results

The PSF system offers the possibility to give a graphic representation of the results through the GUI, the easiest way being using charts, each kind fetcher outputting one or several chart, most of which are very simple. Even though having a chart for the AnswerTime might seem unnecessary, it always is nice to have some graphical representation rather than a set of numbers.
8.2. PRESENTING THE RESULTS

Among the possible fetchers, we present the graphic representation of three of them, to give an idea of what is possible. The test used to get these results was very simple: the same property was configured twice in the system, and essentially the same flow of booking was injected into both property, except that one of them had bookings which targeted the whole property instead of a special room type (the system could handle different types of meta room types).

The first fetcher presented in the figure 8.2 was dubbed Difference as it was initially supposed to keep track of bookings which were accepted by one property and rejected by the other, which could be interesting in order to compare the difference of behavior between two algorithms. But, as seen in the part 5.2, the system now tests one algorithm at a time, and thus made this kind of test difficult. Having only one partial floating algorithm to test anyway, this was not deemed a big loss, but the name staid.

The Statistics fetcher (figure 8.3) computes more than it really needs (it is supposed to contain the information necessary to be able to merge two results of the same kind coming from two different jobs), but as it was initially created to give an overview of the test, so it still gives some additional information. It also lets the user know if the tests were run without problems, and if the test flow of bookings run for each algorithm are comparable and therefore give unbiased result.

Finally, the Occupancy fetcher (figure 8.4) gives an idea of how much an algorithm can fill a property, and its graphical representation reflects that.

![Figure 8.2](image.png)

**Figure 8.2.** The fetcher Difference is represented by two simple charts, but it makes it quite obvious that the Floating algorithm accepts more bookings.
Figure 8.3. As the third chart of the representation of the Statistics fetcher shows unequivocally (as we see only one line because the two are perfectly superposed), the flows of bookings run for each algorithms are very similar. Also, the first chart informs us that the PSF system encountered no error during the test, and the second that the Floating algorithm is significantly slower than its non-floating counterpart.
8.2. PRESENTING THE RESULTS

**Figure 8.4.** The Occupancy fetcher really benefits a graphical representation. The user immediately sees that the floating algorithm was able to fill the property more efficiently.
Chapter 9

Conclusion

Finally, the PSF system was quite satisfactory. It was able to deliver what was expected of him, for as far as we could test it. The simple testing sequence proved to be able to perform a lot of different kind of tests, and the approach with fetcher gave the system a good level of extensibility. Splitting tests into smaller tests on one algorithm at a time (the jobs) makes the parallelization easier, and more importantly, the same goes for the compilation of the results, without really hindering the potential of the system. It also gives a great deal of control, as we can discard only parts of the total test, or test another algorithm and compare its performances to the others with no problem.

This thesis leaves another question unexplored: how to ensure we are running an accurate simulation? All we do is injecting a randomly generated booking sequence into the system. While there are some parameters we can tune to have a credible flow of bookings, this remains quite crude and quite delicate. For example, how many bookings for a given period should we generate? Furthermore, the more a hotel is booked to maximum capacity, the more expensive its rooms tend to be, so customers tend to book less in reaction. Currently, there is no way to induce such a feedback in the system. The generation of a realistic flow of bookings is an interesting problem to explore.
Bibliography


