Study About the Relationship Between the Model-View-Controller Pattern and Usability

Author:
Diogo Satoru Aihara

Advisors:
Xavier Ferré Grau
Richard Torkar

European Master on Software Engineering

This thesis is submitted to the Facultad de Informática at Universidad Politécnica de Madrid and the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Contact Information:

Author:
Diogo Satoru Aihara
Email: diogo.aihara@gmail.com

University advisors:
Xavier Ferré Grau
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Facultad de Informática
Universidad Politécnica de Madrid

Richard Torkar
Department of Systems and Software Engineering
School of Computing
Blekinge Institute of Technology

Facultad de Informática
Universidad Politécnica de Madrid
Av. Montepríncipe s/n
28660 – Boadilla del Monte
Madrid, Spain

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Blekinge, Sweden

Abstract

Usability is one of the most important quality attributes in the new generation of software applications and computational devices. On the other hand, Model-View-Controller is a well known software architectural pattern and is widely used in its original form or its variations. The relationship between usability and the usage of Model-View-Controller, however, is still unknown. This thesis tries to contribute to this research question by providing the outcomes of a case study where a prototype has been developed in two different versions: one using Model-View-Controller and another using a widely known Object-Oriented guideline, the GRASP patterns. Those prototypes have been developed based on a non-functional prototype with a good level of usability. With the prototypes in hands, they were compared based on their design and based on the usability heuristics proposed by Nielsen. From this study, we discovered that the usage of MVC brings more advantages and disadvantages to the usability of the system than the ones that were found on the literature review. In general, the relationship between MVC and usability is beneficial, easing the implementation of usability features like validation of input data, evolutionary consistency, multiple views, inform the result of actions and skip steps in a process.

Keywords: Usability, Model-View-Controller, GRASP

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, due to the amount and variety of offers that are available and the presence of computational systems in everyday and everyone's life, one of the most valuable quality attributes in those systems is usability, as it is related to the productivity and satisfaction of users when using those systems [12]. Considering the technological improvements, it is quite acceptable to assume that most of the computational systems available have their User Interface (UI) made through software. Therefore, the usability of the software part has a large impact on the overall usability of a whole computational system.

On the other side, one of the most well known software design patterns is the Model-View-Controller (MVC), which separates the business logic (Model) from the presentation (View and Controller). The usage of MVC brings several advantages being the most visible an improvement in flexibility, ease of change and maintainability. However, the impact of the usage of MVC into the usability of a system is still not clear.

The relationship between MVC and usability is considered just in the aspect that the separation of business logic and presentation is one step that can be taken to achieve better user interface maintenance, through an increased modifiability [25, 5]. In iterative development, usability evaluation usually points out changes to be made to the user interface. Therefore, the ease of modifying the user interface may bring advantages to the project.

However, there are other possible positive impacts that come from the usage of MVC on usability and have not been studied thoroughly. Additionally, there could be negative impacts. For example, Seffah in [41] suggests one possible negative effect, where the intense communication between model, view and controller might result in a large processing overhead, compromising the performance of applications, and thus affecting their usability level.

In order to point out the effects on the usage of MVC in the usability of a software product, two prototypes have been developed, one using MVC and another following the GRASP patterns, which is a well known reference for Object-Oriented design. Both prototypes were based on the same non-functional prototype, which was elaborated after two iterations with usability testing. Usability evaluation based on usability heuristics has been performed in both prototypes to check the impact that comes from the usage of the MVC architecture. The study of the effects of applying the MVC pattern may bring new contributions to the issue of architectural decisions that have an effect on software usability.

1.1 Research Method in Brief

This study started with a literature survey that covers the necessary concepts of usability and software architecture. In this initial phase, the theoretical knowledge has been acquired from well established sources as books and articles that have been used on the academia and industry. This survey covers aspects like the concept of usability, usability testing, usability heuristics proposed by Nielsen [37], object-oriented design principles and Model-View-Controller.

Also, in the literature survey, the recent works that relate Model-View-Controller and usability have been surveyed. The object of this study is to analyze and understand the current knowledge about this topic. For that, a search for the recent studies has been conducted in six different bibliographic repositories and the results were filtered and the remaining ones have been analyzed in this work.

With the theoretical knowledge gathered, the development of a non-functional prototype took place. The prototype was designed, developed and tested with potential users and passed through a usability heuristics analysis.

With this non-functional prototype in hands, the development of two functional prototypes started. One of the functional prototype has been developed based on the concepts from the General Responsibility Assignment Software Patterns (GRASP) [30] and the other prototype has been developed using Model-View-Controller architectural pattern.

Those two functional prototypes had their designs compared and they have been analyzed based on the usability heuristics from Nielsen [37]. In this analysis it has been considered the actual implementation of the prototypes as well as possible features that could be implemented in both designs.

1.2 Structure

The document is structured as follows: chapter 2 gives an overview of the two concepts that constitute the background of the present work: usability and software architecture; chapter 3 presents the results of the survey of the latest studies about the relationship between usability and Model-View-Controller; chapter 4 explains how the research has been conducted; chapter 5 describes the development of the non-functional and functional prototypes; chapter 6 presents the results from the analyses regarding the developed prototypes and this work is finalized with chapter 7, where conclusions are gathered. The document is then followed by bibliographical references and appendices A to E provide the results from the usability evaluations that have been conducted on the non-functional prototype.

Chapter 2

Background

In this chapter, the two concepts that compose the theoretical background of this work are presented: software architecture and usability. With regards to software architecture, two main topics are discussed: GRASP patterns, which is a set of Object-Oriented principles that is a well known reference for Object-Oriented design and drives the development of one of the prototypes in this study; Model-View-Controller architectural pattern which drives the development of the other prototype. After that, usability is presented with a description of task-centered design, usability testing and usability heuristics.

2.1 GRASP patterns

Larman, in his book *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* proposed the *General Responsibility Assignment Software Patterns (GRASP)*, which is a set of techniques that help on developing object-oriented designs in a "methodical, rational, explainable way" [30].

Those patterns are based on the concept of responsibility, which is related to what an object does (actions) and knows (information).

There are nine GRASP patterns, which are described below:

2.1.1 Information Expert

The pattern *Information Expert* indicates the responsible class to give a specific information, answering questions like: "Who should be responsible to give this information?". In order to answer this question, the first step is finding the classes that have the information necessary to answer it. After that, we have to analyze how those classes interact and check which one of them is the best for being responsible for providing the desired information. The application of Information Expert pattern helps on maintaining the coupling of the design in a low level, which is the objective of other GRASP pattern. As the information is provided by the classes that are better suited to it, the classes become lighter in terms of amount of "work" that it is responsible of (the processing of an information is divided into some classes, not overloading a single class), and the design becomes more cohesive.

2.1.2 Creator

This pattern addresses the choice of which entity shall be responsible for creating a new object of a class. Generally, the creation of an object of Class1 shall be invoked by Class2 if [30]:

1. Class2 contains or aggregates Class1

2. Class2 has the values that are needed to create objects of Class1

If there are more than one candidate classes for being the creator, the first argument has a priority over the second in the previous list.

2.1.3 Controller

The interaction between the user interface and the rest of the system can be done in many ways. The most common is where the user interface has direct access to the other classes, accessing the services directly from their providers. Another way is having classes that are specialized in receiving the requests from the user interface and redirecting it to the responsible classes. Those specialized classes are called controllers.

The controller can generally be categorized into two types [30]:

- **Facade Controller:** responsible for handling all the requests related to a system, subsystem or device

- **Use Case / Session Controller:** handles all requests that are related to the same use case or session which is generally organized in terms of use cases

Generally, the controllers do not have a high inner intelligence, being them mostly used to coordinate the execution of the requests that are necessary to fulfill the requests that are made by the user interface.

A controller may suffer from overload of assigned tasks. In this case, it goes against the *High Cohesion* pattern.

2.1.4 Low Coupling

The concept of coupling is related to how much does one class need to know about the rest of the system in order to do its work. In other words, coupling is directly related to the amount of classes that one class is dependent to.

Larman show the following situations where we can find coupling from Class1 to Class2:

- Class1 has an attribute that of type Class2
- Class1 methods call services (methods and attributes) of Class2
- Class1 methods return objects of Class2
- Class1 is a subclass of Class2
- Class1 implements or extends Class2

The objective of reducing the coupling is reducing the impact that a change in one class may cause in the rest of the system. Also, with low coupling the effort needed to implement a change is lower. However, there are cases where a high coupling does not represent a problem, especially when the classes are coupled with highly stable elements, such as the programming frameworks.

The effort of lowering the coupling is better spent if is directed to the unstable parts of the system and choosing which parts will receive special attention for reducing the coupling is a responsibility of the designer.

2.1.5 High Cohesion

In object-oriented design, the concept of cohesion is related to "how strongly related and focused the responsibilities of an element are" [30] and has a direct relation with how well a class is easy to understand, reuse, maintain and resistant to changes that are made in the system. Moreover, it has a strong relationship with *Low Coupling*

According to Larman, there are two situations where low cohesion is acceptable:

- When there are team members that are not used to develop objected-oriented applications (for example: database experts), high cohesion may result in a large amount of classes and they will not be able to work properly. However, if we join all the database control into a single class, they may be able to work in a better way.
- The other scenario is where we are dealing with distributed objects. Having a high cohesion design brings a large amount of classes and, the overhead that is caused by having many classes in the system is not desirable under a

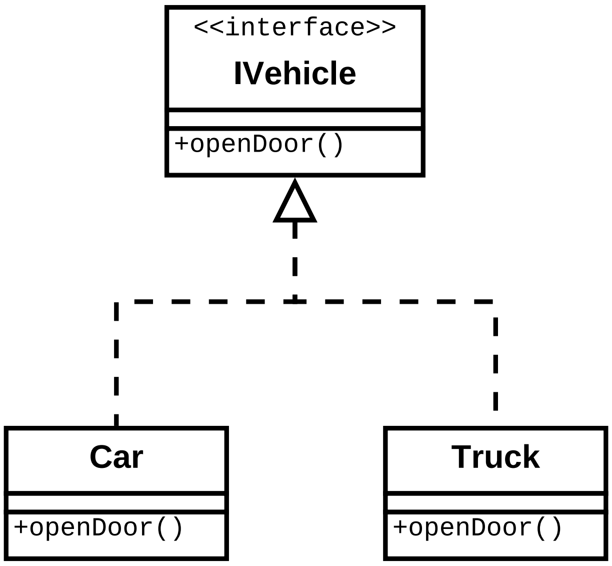


Figure 2.1: Polymorhpism with interface

performance point of view. Therefore, some classes may be joined into a single one, reducing the cohesion, but improving the performance of the system.

2.1.6 Polymorphism

Polymorphism is a pattern that exists to help on improving the design in two possible situations:

1. When there are methods that make usage of conditional statements to change its behavior. In this case, when a new variation appears, the logic may need to be changed in many places.

2. In a client-server system, when we need to make the server susceptible to changes without changing the client.

The polymorphism is done by creating polymorphic methods, which are based on the usage of interfaces or abstract classes. Figure 2.1 shows an example using an interface.

Therefore, in both cases, in order to provide a new variation of behavior, we just need to create a new class and make the necessary adaptations. In this way, we are able to make a design that is easier to change and with smaller possibility of having problems due to changes.

One of the issues that this pattern has is that if it is used improperly, it may lead to a design prepared to future changes that are unlikely to happen, becoming costly and more complex than it should be [30]. Therefore, we need to keep in mind that each usage of polymorphism needs to be carefully analyzed and planned.

2.1.7 Pure Fabrication

There are situations where the solutions proposed by the *Information Expert* pattern makes the design less cohesive or with higher coupling that it should be. In those cases, the *Pure Fabrication* pattern may be applied.

When those situations happen, the solution is creating a new class that comprehends the following methods that are not appropriate and are related among themselves. This class is purely behavioral, not having a match in the application domain. One example is the creation of a class that is responsible for data persistence of business entities into a data source.

In this way, the design still has the needed cohesion, coupling and follows the expert pattern.

When used more than it should, it leads to an excessive amount of behavioral objects that are not related with the information that is necessary to execute its actions.

2.1.8 Indirection

The indirection pattern is used to avoid / remove direct coupling between two or more entities, helping on keeping a low coupling [30].

This indirection is made by creating classes (generally one) between two entities. Generally it is placed between the system and an external entity. In this case, the intermediate class plays the role of a representation of the external entity to the system, blinding it against changes on the external entity - if the external entity changes, the changes that are needed will be focused just on the intermediate class.

2.1.9 Protected Variations

This pattern is used to protect the system against unstable and variable elements that the system may have. The patterns *Indirection* and *Polymorphism* are specializations of protected variations, as they are used to protect the system against changes that may appear in those elements.

The amount of specializations of protected variations is countless, being some of them:

- **Data-Driven Designs:** parameterization of the system at runtime by reading an external data source. One example is configuration files.

- **Interpreter-Driven Designs:** makes usage of rule interpreters, which read the rules that are generally from an external source, interpret them and execute the proper actions. One example is virtual machines.
- **"Don't Talk to Strangers" Principle:** this principle advocates that "inside a method, messages should only be sent to the following objects" [30]:
  - The object itself (this / self)
  - An attribute of the object itself
  - An element of a collection that is an attribute of the object itself
  - A parameter that the method receives
  - An object created inside the method

  This rule can be ignored when the code is dealing with stable objects, as the objective is protecting it against instabilities that may exist.

2.1.10 Model-View Separation

After applying the other GRASP patterns, the designer is supposed to achieve a high quality object-oriented design of the business entities. This design is then introduced into the whole system, which may include other activities, for example: data persistence, user interface, external system communications and logging. One of the most common choices that can be taken to organize those activities of the system is dividing it into layers.

A layer is an element that is composed by packages and classes that are grouped according to their level of abstraction. The layers can be considered as service providers, without any knowledge about their *customers*. Also, the layers consume services that are provided by the lower layers. Buschmann et al. described a layer using the Class-Responsibility-Collaborator (CRC) card that is shown on figure 2.2 [9].

One of the most used examples of layers is the Open Systems Interconnection Reference Model (OSI Reference Model) where one layer of level N provides services to the layer of level (N+1) and uses the services provided by the layer of level (N-1). Figure 2.3 describes this situation.

The Model-View separation is an architecture pattern that makes usage of 2 layers, being one called Model and another View. The model layer comprises the business entities (business logic) and the view layer contains the classes that handle the user interface. Ferré et al. in [14] acknowledges the importance of this separation, stating that "there is a widely-accepted principle in SE stating that the part of the system that manages the visual elements of the UI should be separated from

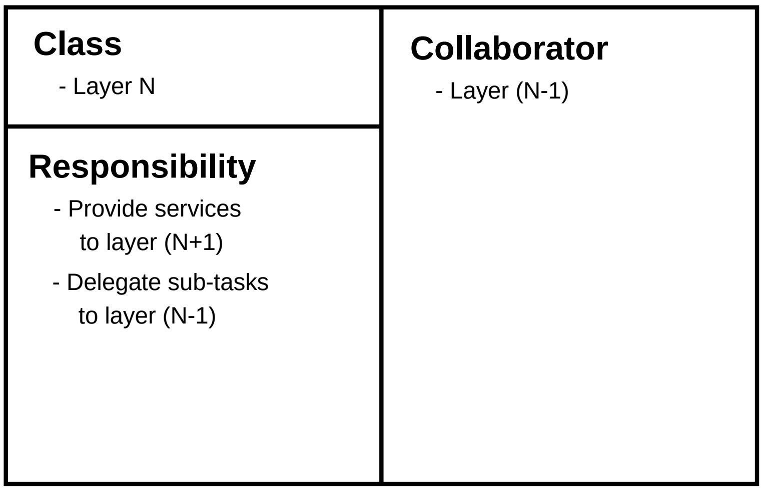

Class
- Layer N
Responsibility
- Provide services to layer (N+1)
- Delegate sub-tasks to layer (N-1)
Collaborator
- Layer (N-1)

Figure 2.2: CRC card for a layer. Adapted from [9]

the business logic (the internal part of the system)". Also, Larman shows some advantages that are brought by the separation, for example [30]:

- Reduce the coupling between user interface and business logic
- Usage of more than one view on the same model
- Change the view completely without impacting on the model

2.1.11 Summary

A summary of the patterns is found on table 2.1.

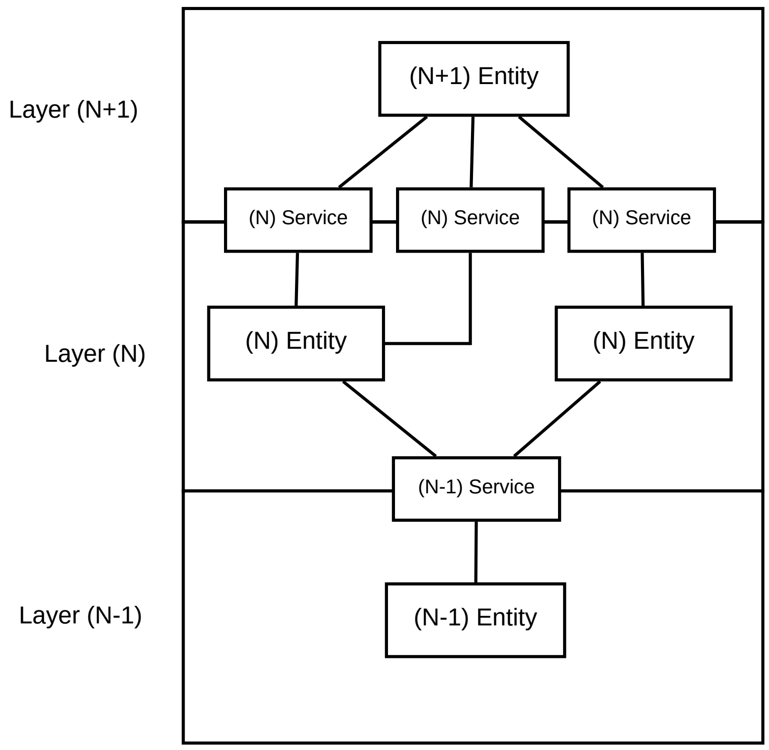


Figure 2.3: Communication among layers. Adapted from [11]

Pattern	**Description**
Information Expert	**Problem:** Finding the responsible for providing a specific information. **Solution:** Assign the responsibility of providing an information to the class that has the necessary tools for that.
Creator	**Problem:** Finding the responsible for creating a new instance of a class. **Solution:** The creation of an object of Class1 shall be invoked by Class2 if: 1. Class2 contains or aggregates Class1 2. Class2 has the values that are need to create objects of Class1
Controller	**Problem:** Centralize the interactions between the user interface and the system. **Solution:** Create classes that are responsible for handling the requests from the user interface.
Low Coupling	**Problem:** Reduce the dependency, change impact and improve the code reuse. **Solution:** Keep in mind the aim of lowering the coupling when assigning responsibilities to the classes.
High Cohesion	**Problem:** Maintain the objects understandable and manageable. **Solution:** Keep in mind the necessity of maintaining the cohesion of the design when making decisions.
Polymorphism	**Problem:** Handling alternatives based on type. **Solution:** Use polymorphic behavior on the operations of the classes where the behavior vary.
Pure Fabrication	**Problem:** How to maintain cohesion and coupling and improve the solution proposed by expert. **Solution:** Create an artificial behavioral class that contains a set of highly cohesive operations.
Indirection	**Problem:** How to avoid direct coupling between two entities. **Solution:** Create an intermediate class that will represent one class to the other element.
Protected Variations	**Problem:** Protect the system against variations and instabilities. **Solution:** Identify the points of predicted variations and instabilities. Create a mechanism to protect the system against those instabilities.
Model-View Separation	**Problem:** How to reduce the coupling between the business logic and user interface. **Solution:** Separate those two elements into different layers.

Table 2.1: GRASP patterns

2.2 Model-View-Controller

2.2.1 Definition of Model-View-Controller

Model-View-Controller (MVC) is an architecture pattern that has been created with the objective of providing ways for increasing the modularity of the software systems, dividing the responsibilities and facilitating the development, understandability and modifiability of software applications [29]. In this pattern, there are three categories of elements, which are: *model*, *view* and *controller* and are described below: And, the controller is responsible for joining the model and the view, receiving the commands from the user and coordinating changes on the model and the view when necessary.

- **Model:** The model is responsible for the business logic, which is the "behavior and data"[8] of the application.
- **View:** The view is responsible for handling the presentation of the user interface. Generally, a screen is not composed by a single view. Instead of that, the views can be nested, having a view a set of sub-views and being itself a sub-view of a parent view. Figure 2.4 shows one example of sub-views.
- **Controller:** The controller is responsible for handling the requests from the user, when he/she interacts with the user interface. For example, when a button is pressed on a screen, the related action is taken by the controller.

There are countless versions of Model-View-Controller, due to the variety of interpretations that have been made over the Model-View-Controller philosophy and adaptations to specific scenarios. Some examples are:

- In his proposal for Model-View-Controller, Burbeck proposed a pattern where model, view and controller are three entities that are mutually connected [8]. This approach is detailed later in this text.
- In Morse and Anderson interpretation, there are not connections between the view and the model, being the controller as the central element that makes the transference of information between the model and the view [35].
- Eckestein discussed about two possible interpretations of Model-View-Controller in his article called "Java SE Application Design With MVC" [13]. They are:
  - One of them has been published by Sun Microsystems where the model can receive method calls from controller and view and create events on the view, the controller can also call events on the view and the view can raise events on the controller [34].

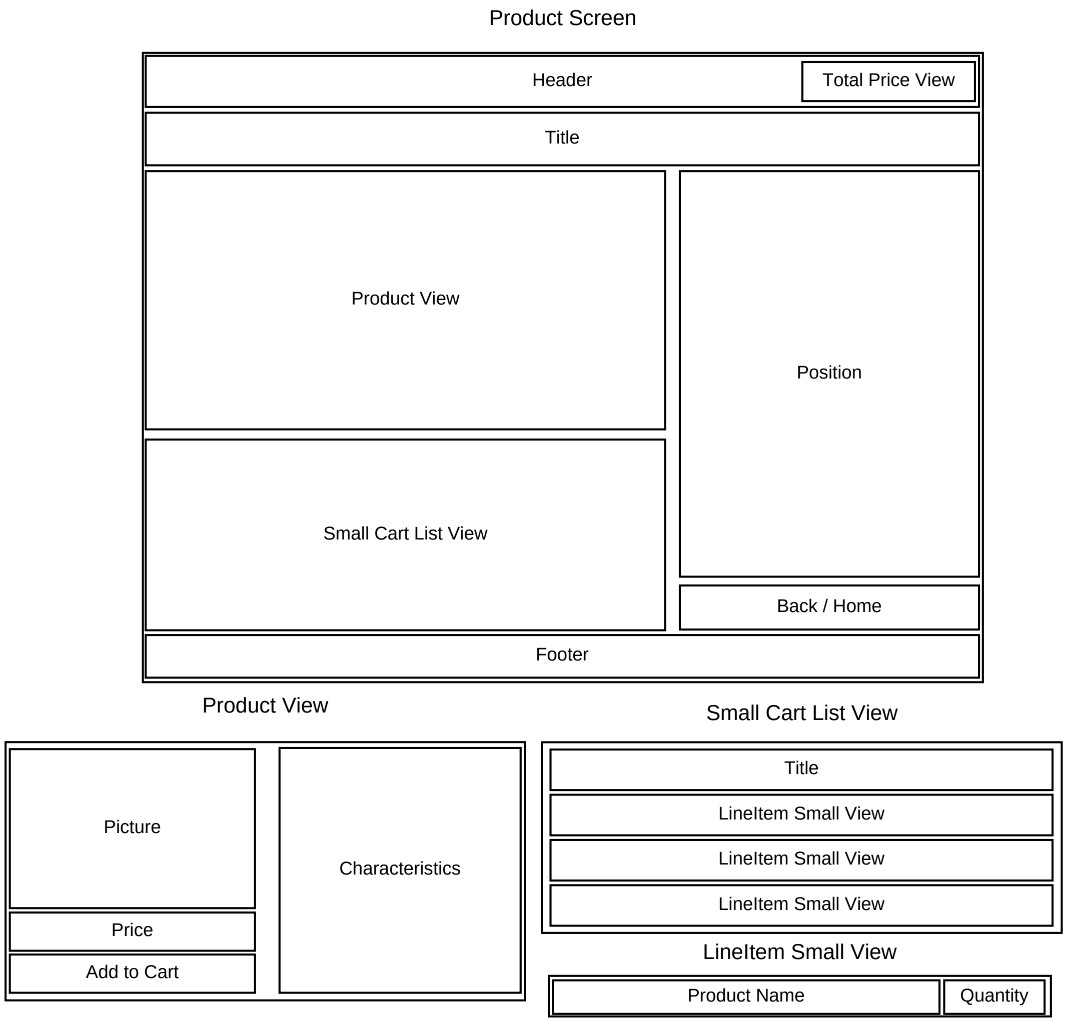


Figure 2.4: Sub-views in Model-View-Controller

- Apple Inc. proposed another interpretation where the controller is the element that joins the model and the view, without direct communication between model and view [2]. This approach is close to the one from Morse and Anderson [35]. Also, in [3], Apple Inc. stated that variations of Model-View-Controller exist and "A view object can be in charge of displaying just one part of a model object, or a whole model object, or even many different model objects.", which reinforces the point of view that there is not just one single interpretation of the Model-View-Controller pattern.

• Microsoft also provided its own proposal for Model-View-Controller in websites, called “ASP.net MVC”. In this implementation, for each webpage, there is only one single controller (being possible to use one controller for more than one webpage). Considering that one view is allowed to have sub-views, we conclude that the limitation comes from the fact that every view in a webpage need to have only one controller class. This implementation is described in [32].

In this way, one version had to be chosen for conducting this study. Microsoft provided in its digital library an explanation about MVC [33] that is interesting and based on some studies that have been used in this research as well, for example [8, 21]. Therefore, this interpretation is the one that has been used during the implementation of the prototypes.

Relationship Between Model, View and Controller

The relationship between the model, view and controller can be in two distinct forms, called *passive* and *active* and are closely related to the behavior of the model.

Passive Relationship

In the passive relationship, the model does not activate any action on the controller or the view. In this case, the role of the model is passive, being basically executing commands from the controller and providing information to the view. Generally in this situation, the controller is responsible for updating the view according to the changes that are executed into the model [8]. Figure 2.5 shows the relationships in the passive mode.

Active Relationship

In the active relationship, the model informs the view about its updates. In this way, when a model is updated, it informs the views that are related to this model about this update. Then, the views can take the necessary actions to refresh themselves. In the case where the model is isolated from the view in a lower layer, the model is not allowed to have a direct reference to the view. For this case, the usage of the Observer pattern makes this communication possible: the model keeps reference to an interface, which is implemented by the view. Figure 2.6 shows the relationships in the active relationship. In this figure, the actions that exist in the passive relationship as well are dashed in order to emphasize the relationships in the active relationship.

There are some situations where the model needs to inform not just the views, but also the controllers about its updates. In this case, views and controllers will implement the Observer pattern and receive the updates from the models. One of

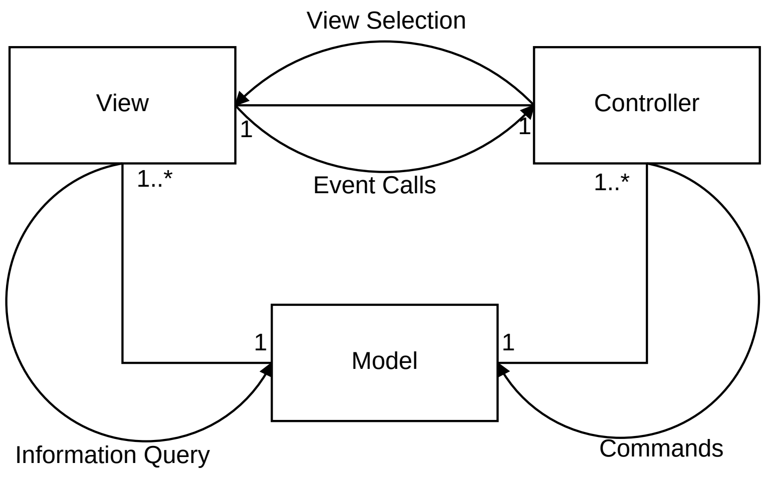


Figure 2.5: Relationships in passive MVC relationship

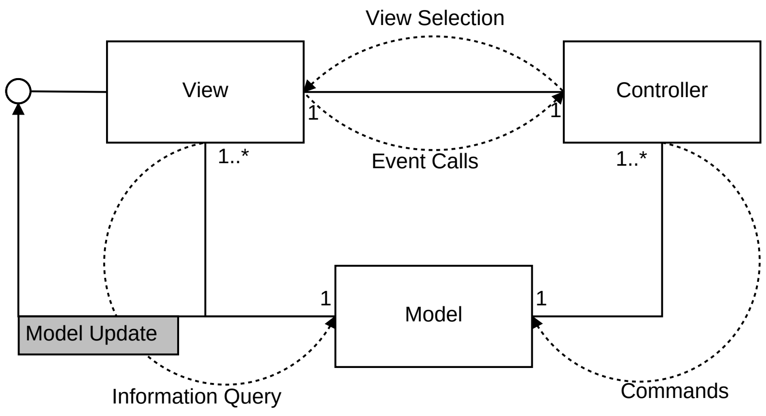


Figure 2.6: Relationships in active MVC relationship

the cases is where the controller needs to control if some user interface controls are enabled or disabled, based on the model [33]. This situation is shown in figure 2.7.

2.2.2 Differences Between MVC Controller and GRASP Controller

As the term *Controller* exists in Model-View-Controller and GRASP patterns, it is important to clarify the difference that exists between those two controllers. In both cases, the term controller is applicable as they centralize the manipulation of actions.

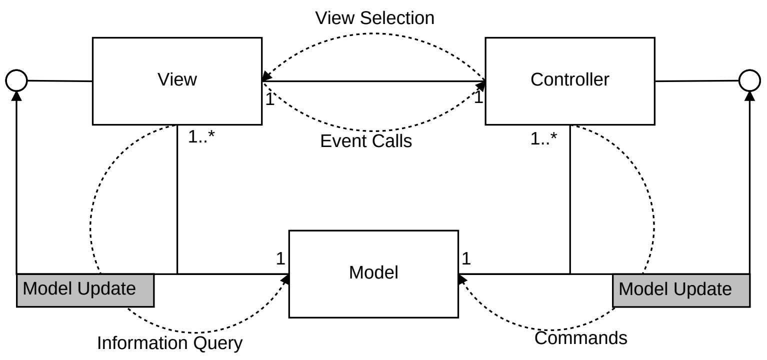


Figure 2.7: Relationships in active MVC relationship with Observer in controller and view

In the case of Model-View-Controller, controllers are responsible for handling the commands from the user. In this way, controllers are the elements that concentrate the manipulation of user commands inside the user interface. Moreover, each controller acts independently and is directly connected to its corresponding model element.

However, in the case of the GRASP controller, the controller is responsible for concentrating, inside its own package, the calls that this package will receive, making it a single entry point for communications. This controller has been designed with the intention of providing an element of indirection between the user interface and the business logic entities, reducing the coupling among those two packages.

Considering that the MVC controllers have direct contact with their respective models, the existence of those two controllers is mutually exclusive as the GRASP controllers existence is based on the fact that it concentrates all communications that would exist between the package that it belongs (in this case, business logic). Therefore, with the usage of MVC controller, the GRASP controller is not used and vice-versa.

2.3 Usability

Usability is a characteristic of every product that is related to "how people work with the product" [12]. Basically usability is focused on how efficiently the users can use the product in order to become more productive, with fewer user mistakes and improved satisfaction. According to ISO 9241-11, usability is defined as: "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [1].

The impact of usability into the computational systems cannot be ignored, as their usability is directly related to the success of those systems. Two studies are presented below to reinforce this statement:

- **E-commerce websites:** Roy et al. made a study for analyzing the impact of usability into e-commerce websites and the outcome is that usability has a direct impact on the e-commerce website trustworthiness towards the users [40].
- **Medical devices:** Zhang et al presented the impact of usability into human medical failures. According to their study, the quality of the electronic devices in terms of usability is considerably low and therefore, may be a fact that impact on deaths in medical centers [47].

From the point of view of a development company, keeping a focus on usability brings the following benefits according to [12]:

- **Increase sales of a product:** the product that had a focus on usability tends to sell more as customers care about usability and, therefore, the chances of selling it increases with a better usability level. One point to keep in mind is that a product that is useless in terms of functionality will not become "good" just because of usability.
- **Selling other products:** when a company becomes famous for having good products in terms of usability, it is likely that they will sell more units of new products because the customers will assume that other products from the same company have good usability as well.
- **Company's reputation:** good products improve the reputation of a company. As usability is closely related to the satisfaction of the customers, the company's reputation tends to increase because of good products in terms of usability.
- **Reduce support costs:** a product with good usability tends to have less amount of contacts with the consumer support department and, therefore, the costs related to support are reduced.

- **Reduce training costs:** if the product is easier to operate, the training that is necessary to start using it is lower and by consequence, the costs involved on training are lower.

- **Reduced costs of internal products:** in case of internal products, the costs are lowered by: lower costs for training and lower costs for support. In both cases, the cost is very important as if support is needed, there are two internal resources that are being "wasted": the one that offers support/training and the one that is receiving them.

2.3.1 Task-Centered Design

One of the possible approaches for developing user interfaces is the Task-Centered design, which is an approach suggested by Lewis and Reiman where the design of the application bears in mind the user interface and is based on an analysis of the actions that the potential users take in order to fulfill the tasks that will be covered by the system that is being developed [31]. Using this approach, we believe that we are able to achieve a good usability level.

When compared with the Waterfall life cycle, the Task-Centered approach differs in two main aspects: one of them is that the Waterfall approach starts with a system specification and the user interface has to adapt into this specification (the Task-Centered approach takes user interface into consideration since the initial phases of the design). This difference can be generated from the fact that generally the team that specifies the system does not have a background in user interface design. This fact is valid in a great percentage of all software design approaches. However, in the Waterfall model, it becomes more evident, as it is not iterative.

The second difference is that the Task-Centered approach has iterations on the design of a system. The usage of the iterations brings the possibility of refining the user interface as much as necessary.

The design process starts in the task-centered design with the decision of the application domain and a general idea of which business process the system will be inserted into. After that, the Task-Centered approach is divided into 11 phases that are described below.

Figure Out Who's Going to Use the System to Do What

This initial phase aims on discovering the potential users of the system and what do the users do when they are taking part of the process where the system will be present.

The analysis of the user is done in order to evaluate who they are, their background and, by this, prepare the system to adapt to the potential users in an easier and quicker way. For example: if the potential users of a system are composed by computer IT professionals we are able to assume that they have a very good computer usage experience and, therefore, include appropriate interaction mechanisms for advanced users, for example: keyboard shortcuts. Another example would be the case where the potential users are persons that do not have any sort of experience on using computers. In this case, the user interface has to be simple and mimic real world items as much as possible.

The task analysis is necessary to understand how the potential users actually interact inside the business process. With this analysis, the designers are able to understand what do the users really do, have a better look at the process and its variables and prepare the system to have a higher acceptance from the users. It can be done by preparing the system to automatically correct common mistakes, impacting the business process as less as possible.

User and task analysis demands that the design team gets a close contact with the potential users [31]. Without this close contact, the user and task analysis may result in a wrong view of the business process and a system that does not match the user needs. Therefore, it is a very important aspect of the design process that needs to be taken care of.

Choose Representative Tasks for Task-Centered Design

In this step, the design team identifies a set of tasks that makes a good representation of the future usage of the system. Also, they need to gather the material that is involved into those tasks. Considering that one of those tasks is “Transform a spreadsheet of the list of supplies that have been bought in a month into a section of a report”, one example of spreadsheet and one example of the resulting report are needed in order to prepare the system correctly.

This step also confers the designers a deeper understanding of the users and the work that the system is supposed to perform.

Plagiarize

Whenever we are working on a design of a system, one wise thing to do is checking what has already been done and is available to use. This way of thinking comes from the point of view that whenever something is being done, they way it is done is either based on previous successful cases or is an attempt to avoid the mistakes that have been done in the past. Moreover, if the system will be used in a specific environment, for example Microsoft Windows-based systems, there are a set of standards that have been used in other systems that run under this platform and the potential users may be used to the way those other systems. At the same time that Lewis and Reiman call this step as “Plagiarize”, they also state that “We don’t mean plagiarize in the legal sense, of course." [31].

Rough Out the Design

During the elaboration of a design, one important thing is putting a preliminary version of it into paper. This action will lead to discussions inside the design team about things that might be left behind or would be better if changed.

Generally this step takes place when the design team is also discussing about which features the system shall have. It is needed to keep in mind that the suggested features have to be inserted into at least one task among the selected representative tasks, otherwise the feature will be useless to the system. Lewis and Reiman also say that a checklist might be useful for checking if all the representative tasks are being covered by the actual design. If they are not fully covered, the design is still incomplete [31].

The last thing that has to be kept in mind regarding this step is that the elaboration of even a single prototype is not advisable because the design is suffering a lot of changes and the prototype would become useless easily.

Think about it

One important action that the design team can take is trying to figure out in an early point of the design the effort that the user would need in order to execute the chosen representative tasks. Also, it is important to try to predict the points of the design where the user might make a high amount of mistakes. If the design team is able to do it correctly, they will reduce the amount of rework that would be necessary to "fix" the design in a later phase of the project.

There are two techniques that have been described in [31]. They are:

1. **Cognitive Walkthrough:** in this evaluation, the reviewer imagines what a particular user would do to execute a task using the user interface that has been designed and uses the following algorithm to evaluate the design:

```
Select one of the task scenarios.
For each user's action in the task:
  Check if this action is reasonable.
  Check if the user knows what and how to do.
  Check if the user knows how to interpret the system output
    from this action.
```

   If any problem has been found, a note about this problem is taken and the reviewer moves to the next step. It is important that, even if the design that has been evaluated looks very bad, it is good that it is reviewed until the end as more issues might be detected.

2. **Action Analysis:** in this case, the focus is on breaking down the actions that are necessary to execute a task. Those actions can be external (for example: "Type your username") or internal (for example: "Transform the 12-hour time into 24-hour time"). With this list of actions, we are able to detect problems on the design. For example: if the task is too complex, if it can be done in an acceptable time, how much does an user need to learn in order to execute this action and even if it is possible for he/she to learn given the actual documentation.

Create a Mock-Up or Prototype

After the design has been created, put on paper and discussed, the next step is building a prototype. The prototypes are useful to show them to potential users, and gather feedback about the design from them. There are several tools available to make prototypes using a computer, but the prototypes "complexity" can vary from elaborated programmed prototypes to prototypes that are made using pieces of paper. In the latter case, the prototype can be used with the help of a second person, which will be responsible for simulating the extra tasks like changing the actual screen and returning the result of an action that the user has done.

Test it with users

When the prototype is finished, it needs to pass through a test session with users that are similar to the future potential users. This test session is necessary to improve the user interface, by checking the gaps between what the designers think and what the potential users think and need. This gap may be of various types: opinions and desires from the users, misunderstandings between what the designers wanted to express and what the users understood and differences of perception between the users about a specific aspect of the system. Usability testing is described deeper in Section 2.3.3.

Iterate

When developing a software, one key aspect to achieve a good quality is refining the software until it reaches the desired quality. Therefore, after we finish the testing with users, the next action that needs to be executed is to analyze the results of those tests and decide the next actions that have to be executed. Those actions comprise changes that have to be done in the design, which will reflect on a new version of the prototype, that has to go through another set of tests. This cycle will repeat until a good quality has been achieved.

The decision about when those iterations shall finish can be arbitrary or based on metrics. In case of arbitrary decision, it is generally based on cost / benefit ratio or feedback from the users that test the system. In case of metrics, the results will be based on the results from the test.

Build the Design

After the iterations for refining the design finish, the next step is implementing the software itself. Lewis and Reiman suggest us to take special care with using variables for defining parameters of the user interface that can be easily changed, predict possible changes on the user interface on the future of the project, even after the iterations of refinement of the design. One reason is that "they [those changes] are especially important for the user interface, which often represents more than half the code of a commercial product." [31].

Tracking

After the software has been released, it is very important for the design team to be aware of the real users opinions about the system because with that, the design team can be prepared to make a better design in the next projects [31]. Even if this information can be obtained by indirect ways (for example reports from customer support service), direct contact between the design team and the users is the best approach as it allows the designers to discuss with the users and acquire a deeper understanding of the users opinions.

Lewis and Reiman suggested some techniques for making this contact between the design team and the users possible: one of them is rotating the design team into the customer support service. Group meetings can also be helpful as it may involve representative persons from the users and customers of the system [31].

Upgrades

Generally, every software that is developed needs to go through upgrades for fixing problems that have been found and addressing opportunities that are convenient. Those upgrades are generally based on feedback from users and customers after releasing the software. This step finishes the design process suggested by Lewis and Reiman [31].

2.3.2 Tutorial for Applying Task-Centered Design

Greenberg presents a tutorial for applying the Task-Centered design into a computer system [22]. This tutorial is focused on the first steps of the Task-Centered design, from the beginning to the creation of a prototype. The sequence of actions that Greenberg suggests is a little different from Lewis and Reiman, however both are easily applicable.

The steps of the process that Greenberg suggested are:

Discovering the Tasks that Users Do

This step is related to the first step of Lewis' and Reiman's suggestion, called *Figure Out Who's Going to Use the System to Do What* and aims to figure out what do the potential users do in their real life with regards to the process where the future system will be part of. In order to do that, there are three approaches that are suggested by Greenberg:

1. **Observe / Interview Real End-Users:** observation of potential users in doing their activities (that are related to the future system) and interviewing them are the best ways of acquiring knowledge about the tasks that are developed. Another technique that can be used is interviewing those users. The interviews can be used together with the observation or independently. Those two techniques demand a close contact with the potential users. Therefore their usage may not be allowed in some circumstances.

2. **Interview with End-User Representatives:** if observation and/or interview is not possible, one alternative is making interviews with representatives of the end-users. Those representatives need to be persons that have a real (and not theoretical) knowledge about the potential users' needs and ways of executing the tasks that will be covered by the future system. The closer those representatives are to the end-users, the higher is the probability of getting a good view about the end-users.

3. **Usage of Own Beliefs About the End-Users:** if the other two possibilities are not available, a possible approach is using the designers' own beliefs about the end-users, their needs and their way of interacting in the process where the system will be inserted.

One important thing is deciding when the designers should stop working on this activity. Greenberg says that generally this activity stops when the amount of new information about the users and their tasks reduce significantly.

Developing Good Task Descriptions

In this stage, the designers write the results from the previous stage as task descriptions and is related to the step *Rough Out the Design* from [31]. Greenberg suggests five important points to keep in mind when writing task descriptions [22]:

1. **Focus on "what" and not "how" the users wants to do the tasks:** when describing the tasks, if the focus is on "how" the actions are executed by the users, we may be throwing away good possibilities of interaction between the users and the system. On the other hand, if we focus on "what" is being done, we have the chance to provide various designs for the same tasks and, by this, be able to choose the best option.

2. **It is specific:** with specific descriptions, we avoid potential misunderstandings that could lead to wrong designs and waste of effort in the best case. In the worst case, we could give a wrong functionality to the users in the system.

3. **It is complete:** if the description covers the task fully, from beginning to end, it allows the designers to consider correctly the inputs and outputs of a task as well as how the user interface elements may interact with each other during a specific task.

4. **It describes the user:** a good description of the potential users of a specific task allow the designers to prepare the user interface based on this user. One factor that is closely related to this fact is the amount of knowledge and capabilities the potential user has about computers, typing and the task he/she is executing. A poor description of the user in this case may lead to a system that is unattractive to the users for being too simple or too complex.

5. **The set of tasks is complete:** when considered as a whole, the set of tasks shall cover all expected users and tasks. If they do not cover, the resultant system may not attend to all requirements from the users.

Validating the Tasks

When the description of tasks is finished, one good activity to perform is checking if this list is complete and fulfills the needs of the users. The most indicated persons to answer it are the ones that have been observed and interviewed. In case they are not available, this validation may be done by other persons that have a close contact with the potential end users [22]. The more uncertain are the source of information to describe the tasks, the more useful this validation may become.

Deciding which User Types and Tasks to Include

The next step is filtering the tasks and users that should be considered on the system and can be related to the step *Choose Representative Tasks for Task-Centered Design* from Lewis and Reiman. Greenberg suggests the following practical role: "most systems are considered successful if they have about 90% coverage" [22]. It means that, we should accept that we are not able to cover all users and all tasks and 90% of them is a successful coverage.

Following this way of thinking, for filtering the users, firstly we divide them into categories. The criteria used to divide them may vary and some of them are: computer knowledge, tasks they perform and knowledge about the business process. The users are then divided into three categories [22]:

1. **Must include:** most important users of the system. Not considering them would be a serious mistake.

2. **Should include:** important users that should take care into consideration if possible. They may have the difficulty of usage increased in order to attend the *Must include* audience or be left to be attended into a second release of the product.

3. **Exclude:** less important users that will be left behind when designing the system. Possible implications are: being unable to use the system, use with difficulties or not being able to adapt easily to the system.

In the same way, the tasks can be divided into:

1. **Must include:** essential tasks that are the core of the system. Generally their usage is frequent.

2. **Should include:** tasks that should be included in the project if there are resources and time available. If it is not possible to include them into the first version of the system, include them on the second.

3. **Could include:** should be taken into consideration if their cost for putting them into the system is low. Their cost can be low if they have a lot of things in common with other tasks for example.

4. **Exclude:** unimportant or rare tasks that will not be used or missed by the majority of the users.

Design Through Scenarios

From the descriptions that have passed the filtering on the previous step, we can mount stories that have to be addressed by design alternatives that will be created. When elaborating those design alternatives, the team needs to keep in mind that the proposed design has to be able to attend the users' expectations and be applicable in the real world.

Evaluate Via Task-Centered Walkthroughs

After creating the design, it goes through a evaluation based on the usage of a walkthrough. The steps that are involved on this walkthrough are described in Section 2.3.1. Basically, the reviewer needs to imagine the actions that are need to execute an arbitrary task and check if the user will have an acceptable way of executing them.

This technique can be done by the designers themselves, but if it is executed by other team members, the results tend to be better [7].

2.3.3 Usability Testing

In order to check if an user interface design matches the potential users' needs and expectations, usability tests are used. Usability tests need the participation of potential users of the future system, who will test your prototype and provide detailed feedback for improvements.

User Selection

The first step that needs to be taken for usability testing is choosing the most appropriate users. Those users have to fit into the profiles of the users that the system is aiming. Sometimes it is not feasible to have a perfect match between your aimed users and the ones that are available for executing the usability tests. In this case, the suggestion is to find similar profiles. For example: if you are not able to find persons that are between 55 and 60 years old, perhaps you can manage the test with someone that is 52 years old. If you did not find the correct match of user profiles, you need to consider it when analyzing the selected user's feedback [31].

Task Selection

After you selected the users, the next step is selecting the tasks that will be executed by the users. The set of tasks that are selected to be executed shall reflect what has been used to plan the system. In case of task-centered design, the best tasks to be chosen are among the ones that have been selected to drive the design of the system. The description of the tasks shall not be focused on user-interface actions (for example: “buy the product that is shown on the screen” instead of “click on ’Buy’ button”), unambiguous and compatible with the users’ understanding.

Also, the designers need to prepare an introduction to the user, to make him/her aware of the scenario where he/she will be involved when performing the tasks. This will help him/her on understanding what does the system do and what he/she is doing inside the system.

The order of execution of tasks can be made starting from the simplest ones, which the users will probably execute without problems and, by this, get a first contact with the system and, when they finish those simple tasks, they will be able to go to the complex ones.

Provide the prototype

One important thing is to prepare the necessary environment to execute the tests and plan how to make it available to the users. If the prototype is in electronic form, the users need to have access to a computer in order to execute the test. This prototype may be made available online or may be distributed physically. In case of paper prototypes, the designers have to prepare themselves to be able to conduct the test in a good manner. One alternative to this case is having two persons to coordinate the test instead of one: the second person would be responsible to provide the system’s responses to the user’s inputs.

Select What Data Will Be Collected

The design team needs to decide which data they are interested in obtaining from the tests. Those data may be:

- **Numerical:** quantitative values, for example: time taken to execute the tasks, number of mistakes that have been detected, grades that the users have given to characteristics of the prototype.
- **Text-based:** open questions, for example: “What is the best and the worst functionality of the system?”, “Would you like to have this system available for your use?”, “Would it justify an investment of (amount of money) for its usage?”.
- **Numerically-transformed:** there are questions that even if the answer is a text, the answers may be transformed into a numerical value for quantitative analysis. In order to convert the text-based answer into a numerical variable, the design team needs to provide an arbitrary scale for that, with a direct mapping of text and number. In order to simplify the transformation into a

Question	"What is your opinion about the system?"	
Possible Answers	"Very Bad", "Bad", "Medium", "Good", "Very Good"	
Mapping	**Answer**	**Value**
	Very Bad	1
	Bad	2
	Medium	3
	Good	4
	Very Good	5

Table 2.2: Mapping of open question answers into numerical variable

numerical variable, the question may have its answers restricted into a set of possible answers. One example is shown in table 2.2.

Based on the data that they want to acquire, the design team prepares a form that will be used on the test, together with the description of the system and the tasks. Those data will be collected during and after the user's usage of the prototype.

Summarize the Data

Together with deciding which data the design team wants to acquire, they need to decide also, what to do with the mass of data that will be collected. Generally the treatment that is applied is based on statistics, therefore the team needs to decide which statistical information they will extract from those data and how to use it for taking their decision for improvements.

Pilot Test

Before doing the tests with real users, the execution of a pilot test is advisable as it will be useful to detect problems on the test execution, preparation and analysis of results. The pilot tests can be done with a less restrict population as it is more focused on finding errors in the test process.

Test Execution

For the text execution, generally one or two persons from the design team are required to be with the user during the test. One of them is responsible for giving instructions and taking notes and the second one would be responsible for simulating the system in case of paper prototypes.

When the tests start, the design team needs to be sure to inform the user that the purpose of the test is detecting issues on the system. Therefore:

- If there is any error, the error is caused by the system that is not well designed. Therefore, the user is never guilty for a system error. The fact that it is a prototype may be used to reinforce this fact.
- If recording is not used, the tests are anonymous. In this way, whatever the user does or says will be part of a mass of anonymous data, protecting his/her privacy.
- If recording is used, then the design team needs to ensure the users that their privacy will be protected. Ensuring that those recordings will be handled by a small set of persons and will be destroyed as soon as possible can be helpful for that as well.

There are techniques that may be used for acquiring more information from the user during the test. They are:

- **Internal recording:** in this case, the test environment will be acquiring information about what the users were doing during the test. Some examples are: log of actions on the prototype and screen captures.
- **External recording:** in some cases, the test can be recorded externally, using microphone and/or videocameras to analyze later the reactions of the user when he/she is using the system.
- **Thinking aloud:** in this case, the design team asks the users to tell them what they are thinking when they are using the system. This technique does not involve recording and the observer from the design team is responsible for taking notes of important things that have been said by the user. It is advisable also for the observer to make some questions to the users in order to extract more information about their thinking. One important point is that the observer is not supposed to answer the questions from the user that are related directly to the actions that he/she is executing as in the real world, there will not be a person to solve doubts of the user.

2.3.4 Usability Heuristics

Usability heuristics is a set of principles for designing user interfaces. A first set was presented by Nielsen and Molich in 1990 [39], and it has been refined and updated until the actual version, which has been published in 1994 [38].

In this process of evaluation, there is a team of evaluators who are responsible to analyze a user interface according to a set of heuristics, which are usability principles that, when followed correctly, provide a better user interface.

In 1994, Nielsen stated that the usage of usability heuristics was widespread and it was possible that each user had its own set of heuristics [36]. It is possible to assume that, given the large usage of usability heuristics nowadays, it is possible that there are many different sets of heuristics, as the one that has been suggested by Somervell et al in [43] and the one suggested by Zhang et al in [47].

The heuristics set that has been considered when developing this research is the one that Nielsen has suggested in [38]. They are described below, identified by a code that goes from UH1 to UH10.

UH1: Visibility of system status

The users need to be informed about the status of the system constantly. This information can be:

- **System-wise:** information about the situation of the user inside the system. Example: in a payment process, the system shall inform the users in which step of the process they are.
- **Specific:** feedback about user actions. If a user clicks in one button, the system shall give some feedback about this input from the user. This feedback can be the desired result from the user or even a message stating that the system received the request (button click) from the user and is executing the corresponding action.

UH2: Match between system and real world

The system shall follow the language that is being used in the real world. For example: in a medical system, the system shall use medical terms. Also, the system shall be similar to the way the users think and operate. For example: it is not advisable for a system to have a form to be filled where the fields are organized from the bottom to the top part of the screen.

UH3: User control and freedom

The system shall provide alternative ways of navigation to the user. Even if the system navigation is simple and straightforward (Step1 ->Step2 ->Step3 ->Finish), the system shall provide ways of going back (for example from Step2 to Step1) and even exiting from the navigation process (canceling or restarting execution). If these criteria are not fulfilled, the users might feel trapped inside the system and consider the system less attractive.

Another important aspect is that the system shall be consistent and predictable. Being predictable does not mean that it cannot give random results, if it is supposed to do so. Being predictable means that it has to work as it is supposed to do so. No one wants to use a system that does not work as it should.

Moreover, the system shall give the users the feeling that the users are the ones who are giving orders. Otherwise, the users will feel frustrated and powerless and, as consequence, losing their interest on using the system.

UH4: Consistency and standards

The usage of standards in the system is advisable as it becomes an easier system to be used. Otherwise, the users would need to learn how to input the same data in different ways in different parts of the system or even input in a way and interpret the output in a different way. Example: if the system handles dates, its format shall be the same in the whole system: input and output - if “September 15, 2009” is written as “15/09/2009” in a screen of the system, it is advisable to have it in the same way in the whole system and whenever the user puts a date, this format should be considered.

Folmer et. al. discussed about three types of consistency that a software shall handle [19]. They are:

- **Visual consistency:** consistency on language and signs that the system provides to the user is also desired as it helps the users on reducing the amount of information that they need to handle and help them on recognizing the outputs from the system in a quicker and more precise way.

- **Functional consistency:** the way of performing tasks in a software has to be consistent.

- **Evolutionary consistency:** consistency among the products of the same software family needs to be considered.

UH5: Error prevention

Whenever possible, the system should help the users on avoiding mistakes. It does not mean that the system shall reduce the users’ freedom on using it, but if the system uses “day/month/year” as default date format for example, it shall find a way of receiving the date in a unambiguous way. One solution would make usage of three fields on the screen: “day”, “month” and “year” or even a widget with a calendar. The usage of one single field for date input would lead to many mistakes as the date can be inserted as “month/day/year”, “year-month-day” and so on.

UH6: Recognition rather than recall

For every moment on the system, always give to the user the information that is necessary in order to execute the required actions in that moment. This will reduce the amount of information that the user would need to know / remember when using the system. At the same time, the system shall avoid giving useless information as it may confuse the users.

The usage of standards and consistent language also helps on reducing the amount of information that the users need to handle.

UH7: Flexibility and efficiency of use

When a user becomes familiar with a system, he/she will feel the need to avoid the interactions that are focused on the new users of the system. In order to satisfy these users, a system may provide accelerators for experienced users. Examples of accelerators are:

- **Shortcuts:** in case of computer systems, the experienced users may be provided with shortcuts to execute their actions. Those shortcuts can be in various forms, for example: command completion in programming environments and keyboard shortcuts.
- **Skip steps in a process:** some steps in a system may be skipped by an advanced user. Examples are: introductory screen that gives basic instructions and screens for data input whose default values are correct for the specific usage or even configurable (in this case, the user would configure the normal case and change just in exceptions).
- **Input buffering:** in some cases, the users can be quicker to input data than the system is to interpret those data. In this case, if the system is able to buffer the input, the users may input the data with their own speed and let the system to interpret in its speed.

UH8: Aesthetic and minimalist design

The amount of information that is displayed in the user interface impacts on the users' ability of using the system effectively. It comes from the fact that, if there is a lot of information on the screen, the users need more time to understand what is being shown and extract the necessary information out of the mass that has been displayed. When it comes to functionalities, the principle is the same. Therefore, in order to improve the user experience, the user interfaces need to be focused on showing only the necessary information and functionalities to the user, reducing the effort that is needed to use the system.

UH9: Help users recognize, diagnose and recover from errors

Error messages are a part of the system that is generally left behind because their importance is not taken into consideration. However, they can be very useful as, when used correctly, they may help the user on having a better idea about the problem that is happening and even solving system errors by themselves.

Therefore, the error messages should seek users' collaboration. In order to do that, the messages should be polite, understandable and precise. Otherwise, the users will ignore the error message or even get sad or offended with the system. In both ways, it makes the system less attractive to the users.

UH10: Help and documentation

Documentation is necessary for every system that is deployed. Considered sometimes as a waste of time, the documentation may be helpful for the persons that are using a system, even if it is considered simple by the developers. It comes from the fact that the users generally need training to use, redeploy and configure a system. Also, we need to consider that a message can be understood in different ways by different persons. Therefore, the documentation has to be provided.

A good documentation shall be focused on presenting only the necessary information: if the documentation contains useless information, the users will not use it and, if the documentation does not contain the necessary information, the users will not be able to use it either. Also, the documentation needs to be intelligible. If the users do not understand what is written, the documentation becomes useless. And, if the documentation is simpler to understand, it tends to be used more frequently.

Chapter 3

State of the Art: Usability and Software Architecture

The relationship between usability and software architecture is an issue that is being increasingly addressed in the last years. This situation was not common in the past, as they were considered as totally unrelated topics. The following sections describe the general opinion regarding this relationship in the past and the actual researches that are being conducted in this area.

3.1 Methodology

In order to conduct this literature review, it has been broken into three main stages: planning, searching and filtering studies and analyses of studies.

3.1.1 Planning

The first stage, planning, can be divided into four steps: question definition, search definition, results filtering and data acquisition.

In the first step, question definition, the objective is defining *what will be studied* in the literature review and the following variables were defined:

- **Objective:** The objective of this literature review is to discover the researches that have been conducted on analyzing the impact from the usage of Model-View-Controller into the usability of a software.
- **Background:** Usability is one of the quality attributes in a software system. As a quality attribute, software usability may be affected by the design that has been used to develop it. The understanding of this impact is necessary to improve the usability of the future software systems that may be developed.

One of the most well known software architectures patterns is the Model-View-Controller, which, together with its variations, is being widely used in the software industry. Therefore, the knowledge regarding the impact between Model-View-Controller and usability may be very useful for the future software systems.

- **Question:** Which initiatives have been conducted to analyze the impact of the usage of the Model-View-Controller pattern into the usability of a software?
- **Keywords:** The keywords were defined together with synonyms. They are:
  - Model-View-Controller, MVC
  - Impact, Influence
  - Usability

The second step, search definition, is focused on defining how the search for studies will be conducted. The following decisions were made:

- **Search methods:** Search using web search engines. In order to do that, the search string needs to be defined for each search engine.
- **Studies sources:** The search engines that were used and their search strings are:
  - IEEExplore: (”Model-View-Controller” <or> MVC) <and> (”Impact” <or> ”Influence”) <and> Usability
  - ACM Digital Library: (”Model-View-Controller” or MVC) and (”Impact” or ”Influence”) and Usability
  - ScienceDirect: (”Model-View-Controller” OR MVC) AND (”Impact” OR ”Influence”) AND Usability
  - SpringerLink: (”Model-View-Controller” OR MVC) AND (”Impact” OR ”Influence”) AND Usability
  - Inspec + Compendex (Engineering Village): ((Model-View-Controller OR MVC) AND (Impact OR Influence) AND Usability)
  - Google Scholar: (”Model-View-Controller” OR MVC) AND (”Impact” OR ”Influence”) AND Usability

With the search procedure defined, the next step is defining how the studies will be filtered. The decisions are:

- **Selection criteria:** The studies need to relate usability and either model-view-controller pattern or software architecture.
- **Selection procedure:** The selection is divided into two steps:
    1. Filter the studies based on their abstract.
    2. Filter the studies from the first filter after full text read.

After filtering the studies, two actions will be done: firstly, a description of the study takes place. After that, an analysis about the study and the relationship between the study and the impact between Model-View-Controller and usability.

3.1.2 Searching and Filtering Studies

The search for studies brought a total of 441 studies, which were divided through the search engines as shown in table 3.1.

Source	**Amount of Results**	**Observations**
IEEExplore	153 results	Full text search
ACM Digital Library	11 results	
ScienceDirect	97 results	
SpringerLink	129 results	
Inspec + Compendex (Engineering Village)	1 result	
Google Scholar	50 results	Total amount of results: 2300. However, only the 50 more relevant studies for Engineering, Computer Science, and Mathematics were considered.

Table 3.1: Amount of study search results

After gathering the search results, the studies have been filtered according to their abstract. After applying this filter, 25 studies remained. They are listed in table 3.2.

The studies that passed through the filtering by abstract were then read fully and eleven studies were considered useful for this literature review. They are listed in table 3.3.

Author	**Title**
E Folmer, J Van Gurp, J Bosch	A Framework for Capturing the Relationship between Usability and Software Architecture
E Folmer, J Bosch	Architecting for Usability: a Survey
E Folmer, J Bosch	Architecturally Sensitive Usability Patterns
E Folmer, M Welie, J Bosch	Bridging patterns: An approach to bridge gaps between SE and HCI
BE John, L Bass, MI Sanchez-Segura	Bringing Usability Concerns to the Design of Software Architecture
N Juristo, AM Moreno, MI Sánchez	Clarifying the Relationship between Software Architecture and Usability
E Folmer, J Bosch	Experiences with Software Architecture Analysis of Usability
NB Harrison, P Avgeriou	Leveraging Architecture Patterns to Satisfy Quality Attributes
A Seffah, T Mohamed, H Habieb-Mammar, A Abran	Reconciling usability and interactive system architecture using patterns
L Bass, BE John	Supporting usability through software architecture
L Carvajal	Usability-Enabling Guidelines: A Design Pattern and Software Plug-in Solution
NC Junior, AJ Abackerli, LA Consularo	User Interface Design for VCMMs
J Kaartinen, J Palviainen, K Koskimies	A Pattern-Driven Process Model for Quality-Centered Software Architecture Design –A Case Study on Usability-Centered Design
BE John, L Bass, E Golden, P Stoll	A Responsibility-Based Pattern Language for Usability-Supporting Architectural Patterns
L Bass, BE John	Architectural Patterns for Usability
V Paelke, K Nebe	Integrating usability engineering and software engineering in mixed reality system development
L Bass, BE John	Linking usability to software architecture patterns through general scenarios
N Juristo, A Moreno, MI Sanchez-Segura	Moving usability forward to the beginning of the software development process
J Stocklein, C Geiger, V Paelke, P Pogscheba	MVCE - a design pattern to guide the development of next generation user interfaces
MJ Mahemoff, LJ Johnston	Principles for a usability-oriented pattern language
E Nakagawa, E Sousa, K Murata, G Andery, L Morelli, J Maldonado	Software Architecture Relevance in Open Source Software Evolution
KM Hansen, M Thomsen	The domain model concealer and application moderator patterns
E Golden, BE John, L Bass	The value of a usability-supporting architectural pattern in software architecture design a controlled experiment
B Biel, V Gruhn	Towards Design Patterns for Mobile Usability Support
NB Harrison, P Avgeriou, U Zdun	Using Patterns to Capture Architectural Decisions

Table 3.2: Studies filtered by abstract

Author	**Title**	**Reference**
E Folmer, J Van Gurp, J Bosch	A Framework for Capturing the Relationship between Usability and Software Architecture	[18]
E Folmer, J Bosch	Architecting for Usability: a Survey	[16]
E Folmer, J Bosch	Architecturally Sensitive Usability Patterns	[15]
E Folmer, M Welie, J Bosch	Bridging patterns: An approach to bridge gaps between SE and HCI	[20]
BE John, L Bass, MI Sanchez-Segura	Bringing Usability Concerns to the Design of Software Architecture	[25]
N Juristo, AM Moreno, MI Sánchez	Clarifying the Relationship between Software Architecture and Usability	[28]
E Folmer, J Bosch	Experiences with Software Architecture Analysis of Usability	[17]
NB Harrison, P Averigou	Leveraging Architecture Patterns to Satisfy Quality Attributes	[23]
A Seffah, T Mohamed, H Habieb-Mammar, A Abran	Reconciling usability and interactive system architecture using patterns	[41]
L Bass, BE John	Supporting usability through software architecture	[4]
L Carvajal	Usability-Enabling Guidelines: A Design Pattern and Software Plug-in Solution	[10]
NC Junior, AJ Abackerli, LA Consularo	User Interface Design for VCMMs	[26]

Table 3.3: Studies filtered by full text

3.1.3 Analyses of Studies

The studies that have remained after the second filter, full text read, were then described and analyzed based on the studies themselves and the relationship that they had with the main question of the literature review: the impact between Model-View-Controller and usability. The result is shown in section 3.2.

3.2 Literature Review

3.2.1 Introduction

In this section a discussion about the works that have been selected using the aforementioned methodology takes place and it starts with a summary of the beliefs of the Software Engineering and Human-Computer Interaction fields regarding the relationship between Model-View-Controller and usability in the past decades of 1980 and 1990. After that, we present the results that were achieved in a literature review from 2004 about the knowledge that has been gathered about this matter.

After the description of the two introductory studies, the description about eleven works takes place and each of them is divided into two sections. The first is a brief description of the study itself; the second is an analysis about the study and the contribution to the knowledge regarding the relationship between Model-View-Controller and usability.

3.2.2 Previous Decades: 1980 and 1990

In 1980s, usability was considered as an issue related exclusively to the user interface. In this way, the approach that used to be taken was separating the user interface from the rest of the system [24]. Considering the opinion that existed that time, this separation would be enough for helping on solving the usability issues that might appear, as the changes related to usability would be isolated in the user interface code.

However, as two entities are not able to communicate among themselves if they are totally separated [24], the architecture that makes the separation of user interface and the rest of the system had to be completed with an extra element. The most famous architectures that follow this approach is the Model-View-Controller architecture, which is divided in:

- **Model:** Business logic of the system.
- **View:** User interface.
- **Controller:** The *extra element* which is responsible for joining the user interface and the rest of the system (business logic).

A more detailed explanation about Model-View-Controller architecture can be found in section 2.2.

Also, in order to improve the development of systems, tools for designing user interfaces started to appear and being used. Those tools worked in the same way as the tools that are being used nowadays, creating user interface elements (for example buttons and text boxes) in an easier way.

In the beginning of 1990s, it was assumed that the techniques that have been developed in the previous decade were still the solution for dealing with usability in software development [24], which means: development with separation of user interface / rest of the system and changing the software after testing based on usability. However, in the middle of the decade, Newell and Card suggested that a better approach for dealing with usability in software development would be considering usability during the design of the system, proactively.

The point of view that usability needs to be considered since the design of the system was the one that has been predominant in the end of the 1990s, but it did not consider the software architecture particularly [24].

3.2.3 Architecting for Usability: A Survey

In 2004, Folmer et al. provided a literature review where they cover the software engineering practices for covering usability in the developed systems. For that, they discuss about the concept of usability, usability evaluation and design approaches in software architecture for developing applications with good level of usability [16].

Based on this discussion they conclude that the current approaches in software engineering for usability are not giving satisfactory results. Generally usability is only considered in the beginning of the development during the requirements elicitation and in the end of the software development, in the test phase.

The reason for conducting usability evaluation only in the final stages of its development is that it needs a working system and a set of representative users. Therefore, the changes in the software that are related to usability can only be done in the end of software development. Considering that impact of usability changes can cover not only the user interface, but the software architecture as well, the changes related to the software architecture may be left behind due to its high cost for implementation.

One way of reducing the cost of usability changes implementation is considering usability through the whole development process. For that, the knowledge regarding the impact between usability and the inner sections of the software, i.e. software architecture, needs to be improved as well as techniques for driving the software design considering usability. This fact has driven some recent researches in this area.

3.2.4 Framework for Relating Usability and Software Architecture

In the work entitled “A Framework for Capturing the Relationship between Usability and Software Architecture” [18], Folmer et al. proposed a framework for mapping the relationship between the usability of a system and its software architecture. This framework received later the name Software-Architecture-Usability framework and has been discussed in the work entitled “Architecturally Sensitive Usability Patterns” [15] as well.

Using the results from a literature survey [16], they proposed the division of *usability* into a set of *usability attributes*. They are:

- **Learnability:** the speed and ease that a user have to learn and remember how to use a specific system.
- **Efficiency of use:** the amount of tasks that a user can perform using a system in a certain period of time.
- **Reliability in use:** the amount of mistakes that a user commits in a certain period of time when using the system as well as the needed effort for recovering from those errors.
- **Satisfaction:** the opinions of the users regarding the system.

Also, they proposed a set of *usability properties*, which are lower-level usability characteristics that, together, form the usability attributes. Figure 3.1 shows the relationship between the usability properties and usability attributes. The usability properties are:

- **Natural Mapping:** the system shall provide elements that makes easier for the user to understand how to operate the system.
- **Minimize Cognitive Load:** the system shall provide to the user mechanisms for reducing the amount of information that he/she must keep in mind.
- **Adaptability:** the system shall adapt itself to the needs of the user.
- **Guidance:** the system shall provide relevant and easy to understand information for the user, with the aim of guiding he/she during the execution of the system.
- **Error Management:** the system shall provide mechanisms for avoiding errors and, in case they happen, the system shall provide ways for the user to overcome this error.

- **Explicit User Control:** the user shall have the feeling that he/she is in control of the system.
- **Provide Feedback:** inform the user about the status of the system. The status shall be informed in every convenient moment.
- **Consistency:** the system shall provide similar ways for executing similar actions and use similar elements for representing similar things.
- **Accessibility:** the system shall provide as much ways of using it as necessary.

The framework finishes then with a set of *architecturally-sensitive usability patterns*, which are design solutions for implementing the usability characteristics that are desired in a specific system. The authors provided a set of fifteen patterns, which are:

- **Workflow Model:** split the business process into a flow of tasks; provide to the user only the tools and information that he/she needs to execute a specific step of the flow of tasks.
- **Scripting:** provide mechanisms for executing one or more actions in one or more objects.
- **User Modes:** different ways of using the system, according to the user preference.
- **Wizard:** conduct the user on executing a task by splitting the task into smaller steps and conducting the user through the execution of those steps.
- **User Profile:** for each user, record the user preferences regarding the system and adapt the system to the user preferences.
- **Context Sensitive Help:** provide to the user the documentation that is necessary for the specific section of the system that he/she is using.
- **History Logging:** record the actions that the user executed in the system as well as the actions that have been executed by the system, if necessary and possible.
- **Data Validation:** validate the data that has been inserted by the user into the system.
- **Cancel:** cancel an ongoing operation.
- **Undo:** rollback an action.

- **Actions for Multiple Objects:** provide a mechanism for executing the same action in different objects at the same time.
- **System Feedback:** inform the user about changes in the state of the system.
- **Multiple Views:** exhibit the same information in different ways.
- **Emulation:** emulate the behavior or appearance (or both) of a different system.
- **Multichanelling:** usage of the system with different devices.

Figure 3.1 shows the relationship between the usability patterns and the usability properties.

Analysis

This study has provided two significant results. One of them is the breakdown of usability into smaller items which are called *usability attributes* and *usability properties*. This breakdown is based on a literature review which considered the most important usability definitions that were known. The second contribution is the set of *architecturally-sensitive usability patterns* that relate usability and software architecture.

Those patterns, however have a higher level approach, i.e. they do not cover the implementation of the patterns, being limited to which characteristic has to be provided and a list of examples. In this way, we believe that this set of patterns have a higher value for the initial phase of the development of a software - the requirements elicitation. The patterns may be used by software architects as well, but their usage may be not as easy as it would be if they provided a deeper analysis into the implementation.

In their article, the authors mentioned that multiple views is one advantage that arises from the usage of Model-View-Controller pattern for usability of a system as the usage of Model-View-Controller brings the possibility of using different user interfaces elements (views) to display the same data (model) in different ways.

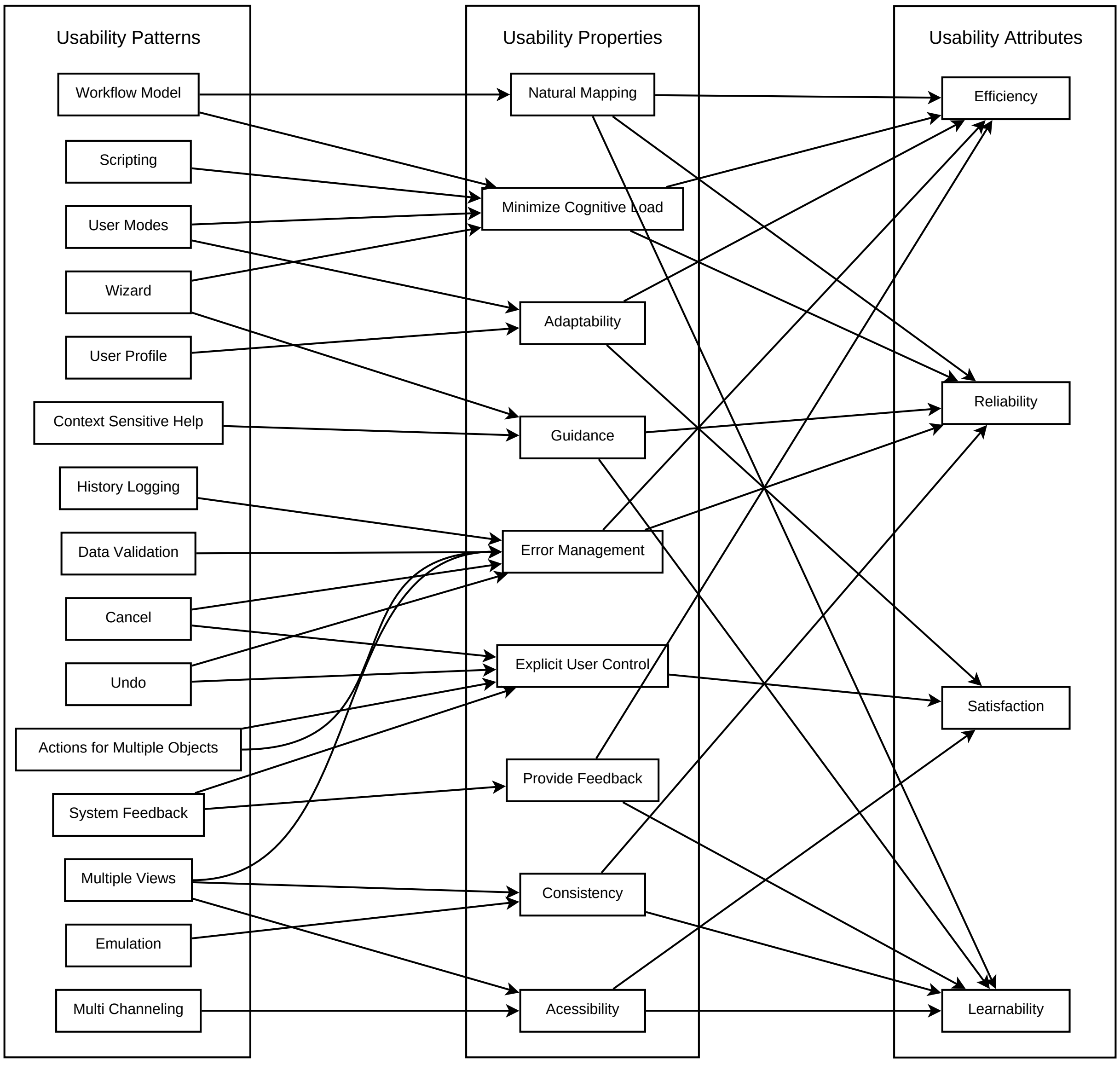


Figure 3.1: Software-Architecture-Usability Framework. Extracted from [18].

3.2.5 Bridging Patterns: An Approach to bridge gaps between SE and HCI

This work, provided by Folmer, Welie and Bosch, continues the work that has been developed in [18] (section 3.2.4) and [45]. The main step that the authors took is providing an implementation approach for the *Interaction Design* (ID) patterns, which comprises the *architecturally-sensitive usability patterns* (from [18]) as well [20].

Bridging patterns are patterns that extend the Interaction Design patterns by adding a general approach for implementing the pattern. In their work, the authors provided four bridging patterns, which are briefly described below:

- **Multi-level undo:** when a user wants to rollback his/her last action, *Undo* is used. The *multi-level undo* is an extension of the *Undo* that provides a record and the possibility of undoing the N previous actions.
- **Multi-channel access:** this pattern provides access to a system using different devices through different user interfaces.
- **Wizard:** conduct the user on executing a task by splitting the task into smaller steps and conducting the user through the execution of those steps.
- **Single Sign-On:** used to provide a single authentication method for authenticating users in different independent systems.

Analysis

This work continues the one described in section 3.2.4 by providing an implementation approach, which is a discussion that was missing in the previous work. Moreover, the authors provided a deep discussion about the possible mechanisms for implementing those patterns in a good depth and making the approach as generic as possible.

The authors state that this work may be useless for experienced designers, but the value for beginners is surely high. Unfortunately, they only discussed about four patterns.

Regarding the relationship between Model-View-Controller and usability, the authors presented the same advantage as the one that is seen is section 3.2.4, which is the fact that the usage of MVC architectural pattern provides the necessary mechanisms for implementing multiple views.

3.2.6 SALUTA: Scenario based Architecture Level UsabiliTy Analysis

In the work entitled “Experiences with Software Architecture Analysis of Usability’, Folmer and Bosch presented the experiences from the usage of a framework for architecture analysis of usability, called *Scenario based Architecture Level UsabiliTy Analysis* (SALUTA) [17].

SALUTA is based on another framework, called *Software-Architecture-Usability* (SAU), which is divided into two contributions: the first is a division of *usability* into *usability attributes*, being each one of them related with a set of *usability properties*; the second is a set of *architecturally-sensitive usability patterns* that are related with the *usability properties* and provide solutions for achieving a better level of usability. This framework is better discussed in section 3.2.4.

The SALUTA framework is composed by four steps, which are:

1. Usage profile creation

In this step, the analysts create a set of usage scenarios. Each usage scenario describes a specific way of using the system and is composed by the following elements:

- **User:** the category of user that is involved in this scenario.
- **Task:** the action that is being performed.
- **Context:** the situation where the action is being executed.
- **Prioritization of usability attributes:** for each one of the four usability attributes considered in this framework, a prioritization value is given. The prioritization value is based on the requirements of the system and vary from 1 to 4.

One example of a usage scenario is shown in table 3.4.

Users	**Tasks**	**Context**	**Satisfaction**	**Learnability**	**Efficiency**	**Reliability**
Account manager	Insert new customer in database	Training	User should feel that he is in control	How easy this task is to understand	The time it takes to do this task	No errors should occur during this task

Table 3.4: Example of usage scenario for SALUTA. Extracted from [17]

2. Analyze the software architecture

In this step, the objective is detecting architectural elements that may support the usability scenarios. This process is divided into two steps:

1. **Detect usability patterns:** in this step, the analysts search for architecturally-sensitive usability patterns inside the software architecture. Those patterns are the ones provided by the framework described in section 3.2.4.

2. **Detect usability properties:** in this step, the analysts check the software architecture and the decisions that have been taken when elaborating it (through reconstruction of the design process) searching for possible usability properties that have been taken into consideration. The usability properties are the ones that have been listed in the SAU framework. Check section 3.2.4 for further information.

3. Scenario evaluation

Using the results from steps 1 and 2, the analysts match the results and, using the SAU framework, they provide grades for level of the usability attributes in each scenario. These results may be expressed in different ways. The authors suggested true/false statements, a five-scale (–, -, +-, +, ++) or a percentage related to how much did the software architecture cover the scenario needs to a specific usability attribute.

4. Interpretation of results

In this stage, the analysts aggregate the results and draw the conclusions regarding the analyzed system. The results may be used for improvements in the software architecture or for concluding that the system achieved the desired level of usability.

Case studies

The SALUTA framework has been applied into three case studies, analyzing the software architecture of three web applications related to their usability.

After applying the SALUTA framework in those three case studies, the authors raised fourteen observations about the framework and its usage as well as general Human Computer Interaction - Software Engineering observations.

Regarding the framework, one of the most important improvements that may be pursued is the reduction of amount of effort that is necessary for applying this framework. Another observation is that as the analysis is subjective, its precision may be improved by using other quantification techniques. And, for applying the framework on different application domains, some specialization of the framework may be useful for more accurate results.

Analysis

This work proposes a different approach for the relationship between software architecture and usability, as it address the evaluation of the usability of the system and its architecture. In this way, it can provide to the software engineers the next steps on the software design for achieving the desired usability level, which can be done using other techniques that focus on the creation and changes of software design and are complementary to this study.

The authors did a good study on analyzing the weak points of their framework and considering that they emphasized that this is still a prototype of a framework, we can expect a more elaborated version of this framework in the future.

Regarding the relationship between Model-View-Controller and usability, the authors presented the same advantage as the one that is presented in “A Framework for Capturing the Relationship between Usability and Software Architecture” [18] and “Bridging Patterns: An Approach to bridge gaps between SE and HCI” [20]. This advantage is the fact that the usage of MVC architectural pattern provides the necessary mechanisms for implementing multiple views.

3.2.7 Supporting Usability though Software Architecture

In this article, Bass and John presented a relationship mapping between three different aspects in the relationship between usability and software architecture [4]. They are:

- **Usability benefits:** usability characteristics that are familiar to usability engineers. This study considered nine usability benefits.
- **Architectural mechanisms:** software architectural mechanisms that are familiar to software engineers. This study considered thirteen architectural mechanisms.
- **Usability facets:** usability aspects that demand software architectural support. They are focused on a single desktop user and the architectural support does not include user interface / business logic separation. This study considered twenty-six usability facets.

The relationship between those three elements is represented in the form of a table and can be seen in [6].

This work has been continued in the form of architectural patterns. This work is discussed in section 3.2.8.

Analysis

In this article, Bass and John considered Model-View-Controller as an example of user interface / rest of the system separation architecture and they state that “separating the user interface is by itself insufficient” for improving the usability of a system. As the authors presented Model-View-Controller as one example of an architecture that separates user interface from the rest of the system and they did not state anything else regarding Model-View-Controller, we assume that their opinion about the relationship between MVC and usability is restricted to the user interface / rest of the system separation and, therefore, insufficient for improving the usability of a system.

3.2.8 Bringing Usability Concerns to the Design of Software Architecture

John and his colleagues proposed a different approach for detecting ways of achieving a good level of usability through changes in the software architecture [25]. In their work, they defined the concept of *usability-supporting architectural pattern*, which is composed by the demands (which are called forces in this study), their context and the solutions for those demands.

The forces can be divided into four categories:

- **Forces from the environment and task:** these forces are related to the demands from the environment that surrounds the software and the user.
- **Forces from human desires and capabilities:** forces that result from the users needs and his/her characteristics.
- **Forces from the state of the software:** forces that come from the execution of the system, inner states, or possible situations that might impact the usability of the software.
- **Forces that come from prior design decisions:** forces that raise from architectural decisions that came before than the proposed solution.

Table 3.5 illustrates a template for the *usability-supporting architectural patterns*. One approach for creating those patterns is as follows:

1. **Identification of scenarios:** identify usability concerns that are not solved by user interface / business logic separation
2. **Identification of forces that impact on those scenarios:** identify which forces impact on this scenario and their origins (types).
3. **Identification of general responsibilities:** from the forces, identify the general responsibilities (requirements) for each set of forces (*environment and task*, *human desires and capabilities*, *state of the software*).
4. **Specific solution:** based on the previous software architecture and the general responsibilities, formulate the specific solution for this scenario.

In their work, the authors provided an example that involves the *Canceling Commands* pattern, which is related to canceling commands that are running and the user wants to cancel them. Further information can be found on [25].

Name: the name of the pattern.			
Usability Context			
Situation: a brief description of the situation, under the user's point of view, where this pattern may be useful.			
Conditions on the situation: conditions that define when this pattern can be useful.			
Potential usability benefits: a brief description of the benefits to the user if the solution is implemented. This description is based on [5, 6]			
Problem			**General Solution**
Forces from the environment and the task	**Forces from human desires and capabilities**	**Forces from the state of the software**	**Responsibilities of the general solution**
Maximum of one force per row			One solution per row
Specific Solution			
Responsibilities of the general solution Repeated from the General Solution column (above).	**Forces that come from prior design decisions**	**Allocation of responsibilities to specific components**	**Rationale** Justifying how this assignment of responsibilities to specific modules satisfy the problem.
Component diagram of specific solution.			
Sequence diagram of specific solution.			
Deployment diagram of specific solution. (if necessary)			

Table 3.5: Template for Usability-Supporting Architectural Pattern. Extracted from [25]

Analysis

The authors proposed a promising approach for elaborating software designs with a higher level of usability. Even if they just provide one example in their work, they have in mind the possibility of gathering more patterns into one single publication and they have about twenty-four patterns that are documented in several levels. One possible issue to be discussed in their future work is the possible relationship that those patterns might have with each other, or even an analysis regarding the forces in different patterns that might be similar.

The impact that the usage of Model-View-Controller might have into the usability of a system is briefly stated in this text. John et al. stated that "it is not sufficient to merely use a separation based pattern such as MVC and expect to deliver a usable system". This point of view goes in the same direction as the one that is provided by Bass and John in “Supporting Usability though Software Architecture” [4].

3.2.9 Clarifying the Relationship between Software Architecture and Usability

In this paper, Juristo et al. provided the results of an empirical study about the relationship between software architecture and usability, focused on the concept of *architecture-sensitive usability mechanism* [28].

In order to do that, they divided the concept of usability into a set of *usability properties*. This concept has been created based on aspects that help developers on building systems with usability in mind. The proposed usability properties are:

- **Keeping the users informed:** the users shall be informed about the status of the system at all times.
- **Error management:** the system shall provide to the user ways of preventing and fixing errors.
- **Consistency:** the system shall provide consistency in its user interface and its functionalities.
- **Guidance:** the system shall provide easy and relevant guidance in the application and in the user documentation, helping the user on using and understanding the system.
- **Minimize cognitive load:** the system shall minimize the amount of information that the user needs to bear in mind when using the system.
- **Explicit user control:** the system shall provide to the users the feeling that they have the control of the system.
- **Natural mapping:** the system shall provide a clear relationship between the intentions of the user and the mechanisms for executing it.
- **Ease of navigation:** the system shall be easy to navigate.
- **Accessibility:** the systems shall be accessible as many ways as necessary.

The authors then related those usability properties with a set of *architecture-sensitive usability mechanisms*, mapping the impact of the mechanisms into the usability properties. This mapping is illustrated in the table 3.6. Considering that this work is relatively close to the one provided by Folmer et al. [18] due to the fact that it came from the results of the same project, we believe that the description of the usability mechanisms that are available in section 3.2.4 is applicable here as well.

Usability Properties	**Architecture-Sensitive Usability Mechanisms**											
	Different languages	Feedback	Undo	Form/Field validation	Wizard	User Profile	Cancel	History Logging	Command Aggregation	Action for multiple objects	Workflow Model	Provision of Views
Keeping the user informed		X										
Error prevention		X		X	X		X		X		X	
Error correction			X				X	X				
Consistency												
Guidance		X			X							
Minimize Cognitive load									X		X	
Explicit user control			X				X			X		X
Natural mapping												
Ease of navigation										X		
Accessibility	X											
Adaptability						X		X				X

Table 3.6: Relationship between usability properties and architecture-sensitive usability mechanisms. Extracted from [28]

Analysis

This work provided a mapping between usability and software architecture. Even if the mappings (this one and the one discussed in section 3.2.4) originated from the same team in the same project, the results are a little different among them. One possible reason is that they originated from the same starting point, but followed different approaches for reaching a common agreement later. Therefore, we expect that there is a common point of view among those authors in some point after those two studies.

This study also provided an observation about the relationship between Model-View-Controller and usability, where the separation of user interface from the rest of the system is not a solution for improving usability, as "functionalities buried in the application logic can sometimes affect the usability of the whole system" [28].

3.2.10 Usability-Enabling Guidelines: A Design Pattern and Software Plug-in Solution

In this article, Carvajal presented the partial results that have been achieved in her doctoral thesis, which is a research for developing a type of artifact called *Usability-enabling guidelines*. One guideline is an artifact composed by the following elements: *Elicitation Guideline*, *Usability-Enabling Design Guideline*, *Code Toolbox* and *Usage Guide*.

The *Elicitation Guideline* is the result of a research conducted by Juristo et al. where they addressed the requirements elicitation, negotiation and specification related to usability [27]. The result from the usage of this guideline is a set of requirements where each one is described with the following information:

- **Identification:** the name of the usability mechanism related to this requirement and the family (type) of this mechanism.
- **Problem:** states that this artifact exists for providing information regarding the identified mechanism.
- **Context:** where and when this usability requirement is necessary.
- **Solution:** rationale for this requirement and the outcomes from the discussion of the requirement with the stakeholders.

The *Usability-Enabling Design Guideline* receives the outcomes from the *Elicitation Guideline* and is divided into two steps.

The first step prepares an artifact that is called *System Responsibility Cluster Map* and is responsible for analyzing the discussions with the stakeholders for dividing the requirements into "clusters of common functionality from which the system responsibilities will evolve" [10]. Each cluster is a set of questions and answers (from the requirements discussion with the stakeholders) that belong to the same functionality.

The second step receives the clusters of common functionality and prepare the *Usability-Enabling Design Patterns*. In order to do that, the analysts breakdown the clusters of common functionalities into sets of *Component Responsibilities*. Each component responsibility is related to one action that the system shall execute in order to achieve the desired usability mechanism. The component responsibilities then evolve into a set of sequence diagrams. The developed patterns are made based on Model-View-Controller pattern, i.e. they will assume that the system that uses those patterns are based on MVC architectural pattern..

With the second step finished, the *Code Toolbox* takes place. In this step, the sections of the sequence diagrams that are not related to the business domain and do not need further specification are then transformed into source code. This step has not been implemented by the author until the elaboration of the article.

The final artifact will be a usage guide for guiding the future users of this framework and has not been developed by the author yet.

Analysis

One of the main characteristics that the author uses to distinguish this work from the already existent researches is that this work will be able to provide traceability between the software requirements and proposed patterns, for verification and validation and for requirements engineering.

The assumption of the usage of Model-View-Controller for implementing the patterns can be considered as a restriction to the patterns. However, if this assumption is not made, the author would not be able to go deeper into the solutions that will be proposed and the work would be less complete.

Even for systems that do not base their architecture on Model-View-Controller will be able to take profit from the artifacts that will be generated from this work. However, it will be without its main characteristics which are the architectural solution and the traceability from requirements to the solution. Therefore, the advantages of the usage of this guideline will be discussable in this case and be focused mainly on the work that have been provided by Juristo et al. in [27] with a smaller contribution from this work itself.

Carvajal observed also that the usage of (user interface / rest of the system) separation-based patterns, like Model-View-Controller, are insufficient to address all aspects of usability. However, this separation is necessary but not sufficient for addressing the usability aspects.

3.2.11 Leveraging Architecture Patterns to Satisfy Quality Attributes

In this study [23], Harrison et al. proposed a way of integrating the information of architecture patterns with regards to their impact on quality attributes.

For that, they analyzed seven quality attributes and eight architecture patterns based on their description. In this analysis, the relationship can be positive (*strength*), negative (*liability*) or *neutral*. In case of strength and liability, when the impact is considerably large, they receive the names *key strength* and *key liability*.

Table 3.7 shows the relationship that those quality attributes with usability in particular and table 3.8 shows the impact of the usage of Model-View-Controller in the quality attributes. The authors provided a complete list of relationships in [23].

Pattern	Impact	Pattern	Impact
Layers	Neutral	**Blackboard**	Neutral
Reflection	Neutral	**Microkernel**	Neutral
Model-View Controller	Key Strength: Synchronized Views	**Presentation Abstraction Control**	Strength: Semantic separation
Pipes and Filters	Liability: Generally not interactive	**Broker**	Strength: Location transparency

Table 3.7: Impact of architecture patterns on usability. Extracted from [23]

Quality Attribute	Impact	Quality Attribute	Impact
Usability	Key Strength: Synchronized Views	**Maintainability**	Liability: Coupling of controllers and views to the model
Portability	Liability: Coupling of components	**Implementability**	Liability: Complex structure
Reliability	Neutral	**Security**	Neutral
Efficiency	Liability: Inefficiency of data access in view		

Table 3.8: Impact of Model-View-Controller on quality attributes. Extracted from [23]

Analysis

This relationship mapping can be very useful for software architects when they will be choosing the architectural pattern that will be applied in their systems. This study has some limitations, which are discussed by the authors in the text and probably are the motivation for their future work on this issue. They are: amount of quality attributes and amount of patterns.

In their table of relationships between quality attributes and software architecture, the outcome from the relationship between Model-View-Controller and usability is the usage of MVC brings (or eases the implementation of) synchronized views. This feature is an extra step from the usage of multiple views and the usage of Model-View-Controller makes the views synchronized given the fact that all views objects that are related to the same data are referring to the same model object inside the system.

3.2.12 Reconciling Usability and Interactive System Architecture Using Patterns

Based on the assumption that usability is related to the system as a whole (and not just the user interface), Seffah et al. proposed a framework for identifying and solving usability issues [41]. This framework is divided into four steps:

1. Identification of scenarios

The framework starts on identifying typical situations where software usability and inner sections of the system are related. Those situations are documented by the usage of scenarios. A scenario is a narrative story that contains a cause-effect relationship, being the cause an inner element of the system (called *invisible software entity*) and the effect a usability issue.

2. Model the scenarios as a cause / effect relationship between quality criteria of invisible software entities and usability factors

In this step, the objective is finding the cause / effect relationship of the quality criteria of the invisible software entities and the usability factors. In their work, the authors used the ISO 9126 measurement framework to create the list of possible quality criteria for the invisible software entities. Examples of quality criteria include: suitability, accuracy, maturity, fault tolerance and understandability among others. The usability factors are the ones that are well known for example: efficiency, user satisfaction, visual consistency.

For better understanding the reasoning behind this cause / effect relationship, figure 3.2 divides the quality attributes of a system as a tree. Based on this division and the concept of "nearly decomposable systems" [42], Seffah et al. suggest that any node on the usability section of the tree may suffer impact from nodes from lower levels on the other side of the tree and the most important relationships would be focused on 1 level of difference. Figure 3.3 illustrates those relationships.

In order to provide examples for this model, the authors provided a table with relationships between usability factors and software quality criteria. Those examples are shown in table 3.9.

3. Usage of patterns as solutions for the scenarios

For describing the solutions for the issues that are described in the scenarios, the proposed framework makes usage of two types of patterns:

- **Software Design Patterns:** software design patterns that are being widely used. Examples include design patterns proposed by Gamma et al. in [21].
- **Interaction Design Patterns:** design patterns that are focused on interaction of the user with the software and can be defined as "are proven user experience patterns and solutions to common usability problems" [41]. The author of this study used the following two sources for interaction design patterns: "Amsterdam Collection of Patterns" [46] and "Common Ground: A

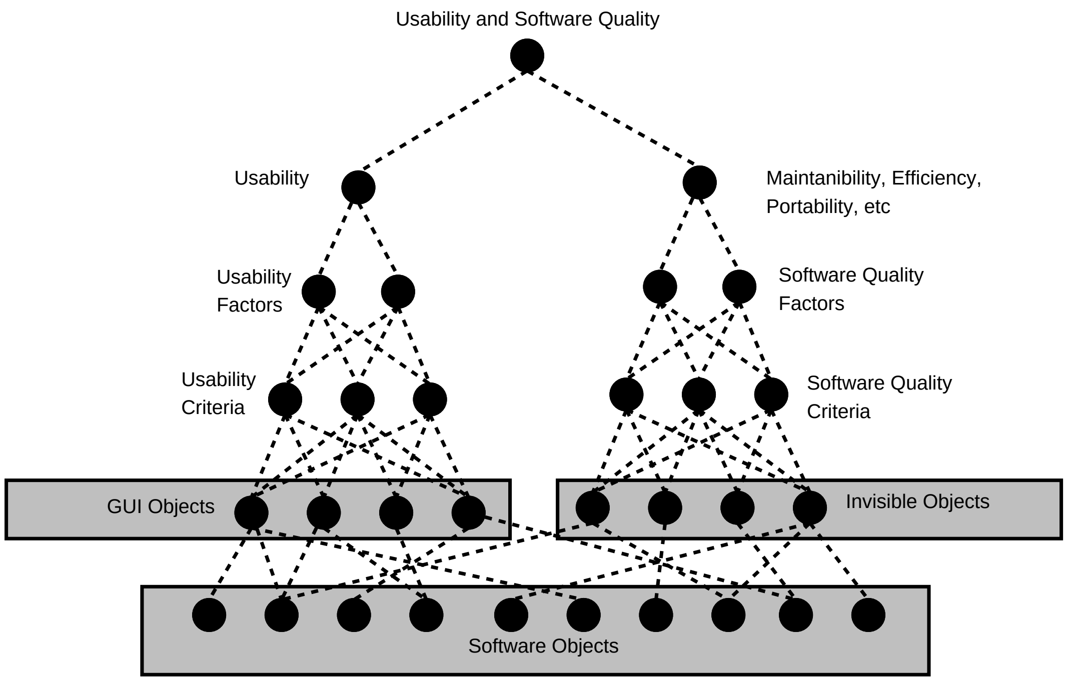


Figure 3.2: Top-down breakdown of software quality attributes. Extracted from [41]

Quality attribute	**Usability Factor Affected**
Performance	User satisfaction
Integrity	Visual consistency
Functionality	Understandability
Suitability	Operability
Recoverability	Attractiveness
Interoperability	Adaptability

Table 3.9: Examples of relationship between usability and quality factors. Extracted from [41]

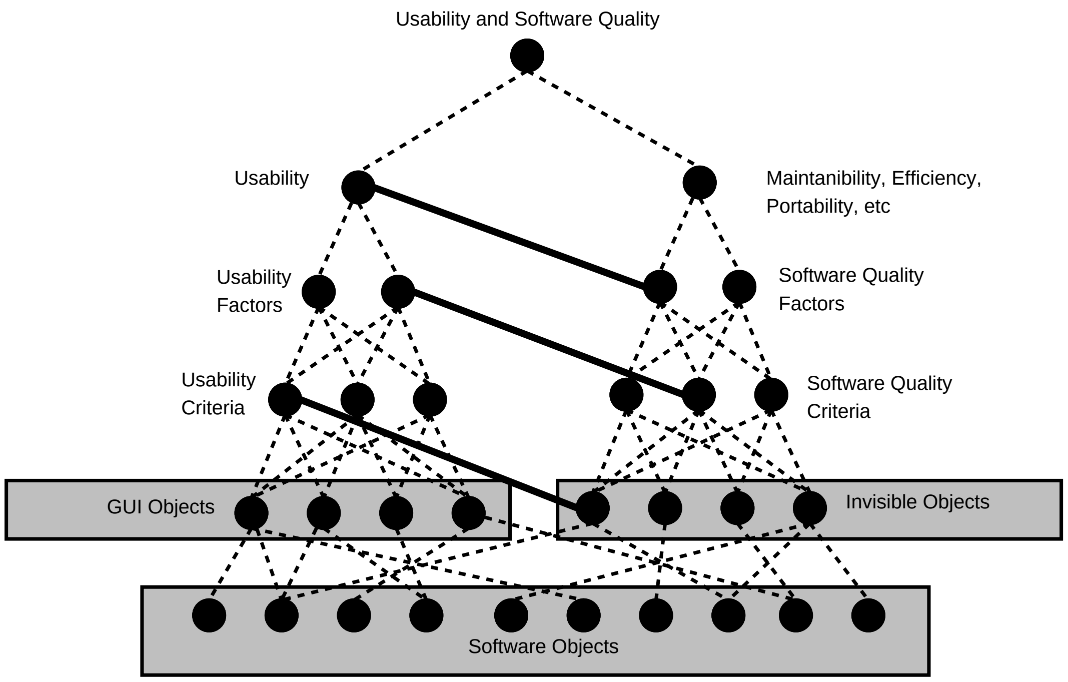


Figure 3.3: Usability and software quality relationship model. Extracted from [41]

Pattern Language for Human-Computer Interface Design" [44]. One example of interaction design pattern is the usage of progress indicator in long processes such as a large file copy.

4. Illustrate how these patterns can be applied in existing architectural models

In this step, the focus is in providing an example of how the developers can apply the suggested patterns into typical existing architectural models, for example Model-View-Controller. One example is the approach that John et al. presented for implementing the *Cancelling Commnands* pattern in [25].

Analysis

We believe that the work that has been developed by Seffah et al. is very promising. Even if the relationships mapping is still "in the beginning" (in authors' words), it provides a good point of view regarding the relationships between usability and quality criteria of invisible software elements.

One thing that seems to be missing is that in step four, the authors did not provide any further comments about how to apply the patterns in existing architectural models. This issue is discussed in other works, for example [25].

In this study, the authors provided a possible drawback that the usage of Model-View-Controller may bring to the usability of a system: in a system that "provides fast, frequent and intensive semantic feedback" [41], the communication between model and view for this feedback may bring an usability issue as it may overload the system and impact in the general usage of the system.

As the work is still in "the beginning" (in authors' words) and we believe that this research has a large potential, we expect to see interesting results regarding it in the near future.

3.2.13 User Interface Design for VCMMs: an Approach to Increase Fidelity and Usability

In their study, Junior et al. presented a case study where they redesigned a software which is a *Virtual Coordinate Measuring Machine* (VCMM). The VCMM is a software that "is an important software tool used in the context of quality systems to generate measurement uncertainty estimates" [26].

This type of tool needs to handle hundreds of variables that are set by the user, therefore usability plays a major role in this system as it is highly related to how the users learn and manipulate the system. The main concerns of the authors for the usability of the new version of the system were:

1. Match between the system and the real world
2. Simple Design
3. Learnability
4. User profiles
5. Recovery from errors

In the new version of the software, the Model-View-Controller pattern has been used as the base architecture of the system. And, in order to provide better usability, the authors took the following actions:

- To divide the data acquisition process into smaller steps, which match the real-world operation of this device and are easier to understand.
- To use the *Decorator* software pattern, which helped on providing visual consistency in the software.
- To create of four types of usage (user) profiles based on the experience of the users regarding the system and the application domain.

After comparing the redesigned version with the previous one, the authors detected an impacting improvement on the usability of the system and they attributed this improvement mostly to the usage of Model-View-Controller, more specifically "on the layers defined by the *View* and the *Controller*".

Analysis

In this study, the authors presented the benefits from using Model-View-Controller in an application for VCMMs systems. The list of benefits were based on the comparison of the usability level between two versions of the same application. The previous version and the new version, which has been redesigned having usability concerns in mind and using Model-View-Controller architectural pattern. The outcome is that there was a large improvement of the usability of the system, which in the authors opinion is due to the usage of MVC pattern. The main reasons are:

- The division of data acquisition variables into partitions through the usage of views
- The usage of views inside views facilitated the creation of usage profiles, which would impact into the user interface according to the selected user profile
- The storage of the simulation variables in the model helps on reducing the complexity for accommodate changes and increase the amount of simulation variables
- The user interface may be changed without affecting the rest of the system, due to the separation of the views and the rest of the system

However, this study did not present a direct comparison between a MVC-based application and a non-MVC-based application, as they had different requirements because the MVC-based application results from an upgrade from the non-MVC-based application, with usability concerns in mind.

Also, most of the improvements could be possibly implemented without the usage of MVC as the are related with the redesign of the software usage process and user interface. Even so, it is possible that the usage of Model-View-Controller has made this implementation easier.

3.2.14 Summary

In this literature review, a representative fraction of the studies that relate usability and Model-View-Controller was gathered. After analyzing those studies, we were able to see that most of the studies have their focus into software architecture in general and, as a consequence, they do not give much specific details about the Model-View-Controller. In general, the observations that are made regarding the relationship between Model-View-Controller and usability are based on one of the two options:

- **Separation of user interface and the rest of the system:** in this aspect some authors consider that this separation is considered one advantage of the usage of Model-View-Controller into the usability and some authors do not include this aspect as an advantage of the usage of MVC, considering it as a more general concept.
- **Multiple views:** this characteristic is related to the possibility of having different views that represent the same object from the business logic. The usage of Model-View-Controller brings an advantage for implementing this mechanism. The reason is that in MVC, a model object may have different views objects related to it.

Table 3.10 shows a summary of the results of this literature review.

One of the studies ("User Interface Design for VCMMs: an Approach to Increase Fidelity and Usability" [26]) goes closer to the analysis of the relationship between MVC and usability. However, in this study, the approach has been redesigning a system with usability concepts where the resultant system used the MVC architectural pattern. We believe that an analysis of the relationship between Model-View-Controller and usability may be better studied if one application that uses MVC is compared with another that does not use it and both of them are developed with the same requirements. In this way, we are able to remove the impact that the pursuit of improvement of the usability level of the application may bring to the analysis.

Therefore, in this work, we conducted a study about the impact between Model-View-Controller and usability based on the approach of implementing two applications that have the same requirements and different software architectures: one using Model-View-Controller and the other without using it. Next section describes the study process in detail.

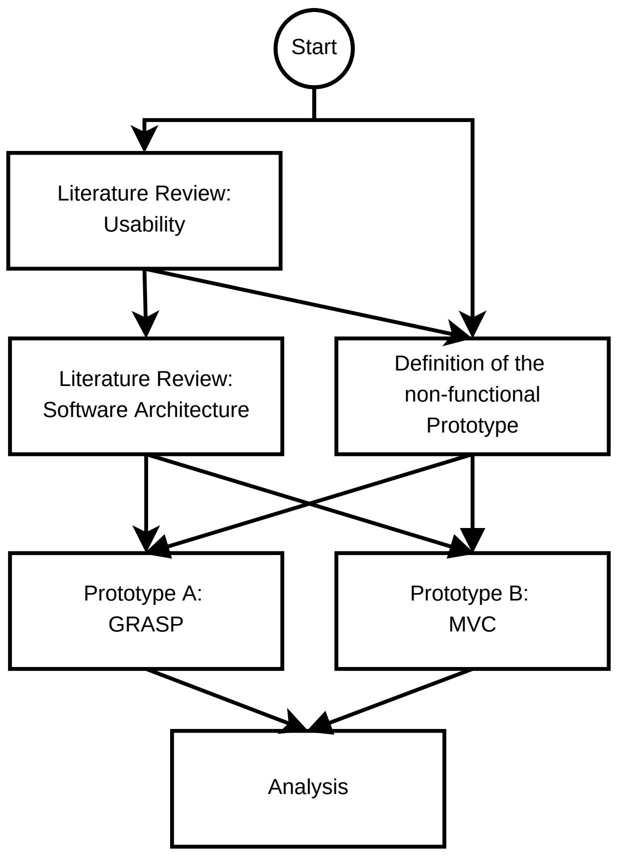
Authors	**Study**	Impact			**Observation**
		-	**0**	**+**	
E Folmer, J Bosch	Architecting for Usability: a Survey		X		
E Folmer, J Bosch	Architecturally Sensitive Usability Patterns			X	Multiple views
E Folmer, J Bosch	Experiences with Software Architecture Analysis of Usability			X	Multiple views
E Folmer, J Van Gurp, J Bosch	A Framework for Capturing the Relationship betwen Usability and Software Architecture			X	Multiple views
E Folmer, M Welie, J Bosch	Bridging patterns: An approach to bridge gaps between SE and HCI			X	Multiple views
L Bass, BE John	Supporting usability through software architecture		X		
BE John, L Bass, MI Schancez-Segura	Bringing Usability Concerns to the Design of Software Architecture		X		
N Jursito, AM Moreno, MI Sanchez	Clarifying the Relationship between Software Architecture and Usability		X		
L Carvajal	Usability-Enabling Guidelines: A Design Pattern and Software Plug-in Solution			X	UI separation (not enough, but needed)
NB Harrison, P Averigou	Leveraging Architecture Patterns to Satisfy Quality Attributes			X	Synchronized views
A Seffah, T Mohamed, H Habieb-Mammar, A Abran	Reconciling usability and interactive system architecture using patterns	X			Model-view comm. in complex systems
NC Junior, AJ Abackerli, LA Consularo	User Interface Design for VCMMs			X	Perception that MVC has greatly contributed to the usability improvement.

Table 3.10: Summary of literature review

Chapter 4

Methodology

This research has been divided into five steps, being them: literature review, definition of the prototype, implementation of the prototypes A and B and analysis. Figure 4.1 shows the steps that have been followed to conduct the research.



Start
Literature Review: Usability
Literature Review: Software Architecture
Definition of the non-functional Prototype
Prototype A: GRASP
Prototype B: MVC
Analysis

Figure 4.1: Steps in the research

The literature review of usability has been applied on designing the non-functional prototype and on executing one part of the analysis of the functional prototypes. This part of the literature review has focused on the following topics: principles of usability, task-centered design, usability testing and usability heuristics evaluation.

The literature review of software architecture served as basis for developing the two functional prototypes and has been focused on the following topics: GRASP - General Responsibility Assignment Software Patterns and MVC - Model View Controller pattern.

The non-functional prototype has been designed since the beginning of the study and has been built using the knowledge gathered from the literature review of usability. The result is a non-functional prototype that has been passed through two usability heuristics evaluations.

With the knowledge of software architecture gathered and with the non-functional prototype built, the development of the two functional prototypes (prototypes A and B) took place: the prototype A has been developed based on the concepts of the GRASP principles, which represent a well known basic reference for Object-Oriented design. On the other hand, the prototype B has been developed based on the Model-View-Controller pattern.

After the development of the functional prototypes, an analysis took place. This analysis is divided into two parts: the first part is based on usability heuristics, where the two prototypes are compared with each other based on the usability heuristics proposed by Nielsen [38]; the second part is a direct comparison between the two prototypes and their design, focusing on the differences that may exist and their impact on the system.

Chapter 5

Prototype Development

In this chapter, the description about the developed prototypes takes place. This description starts with the non-functional prototype, which serves as an input for the two functional prototypes. After that, one of the functional prototypes, which is based on the GRASP patterns is described. After that, the prototype based on the Model-View-Controller architectural pattern is described.

5.1 Definition of the Prototype

The next step has been taken during the course called "Software Usability" on *Universidad Politécnica de Madrid*, where the students had to plan and implement a non-functional prototype of an application. In this step, the work has been developed with the cooperation of Martin Vigo Fariña and Nguyen Thi Phuong Mai.

Firstly, we decided that the application would be a shopping cart assistant, which is a device that would be placed in every shopping cart inside a supermarket. After that, we identified the potential users of the system and investigated how they conduct the activities where the application would be present through observation and interview as suggested in [22]. We defined three types of potential users, which are:

- **Singles:** usually without experience on cooking, they look mostly for easy ways of preparing the food, for example pasta, pizza and frozen food. As they generally do not have salary, price is an important aspect. And, they mostly buy food for themselves. Therefore the amount of purchase is small.

- **Families:** composed by three or more persons, they generally buy food once per week or even monthly. Generally spend a large amount of time on selecting the products and are interested on special offers as well.

- **Pensioners:** with low income and possibly reduced mobility, they do not go often to supermarket. Can be considered as a mixture of *singles* and *families* profiles. Also, they have special interest on special offers.

The goal of the shopping assistant is to improve the experience of the user inside the supermarket, by providing the following functionalities:

- **Product Catalog:** Catalog of products of the supermarket. Search of products by category (listing of categories and sub-categories) and by name.
- **List of Items on Sale:** Special offers from the supermarket, products on discount.
- **Gift Suggestions:** List of products that are good to be given as a gift.
- **Map:** Map of the supermarket, with the position of the user inside it.
- **List of Products Inside the Shopping Cart:** List of what the customer has already put into the shopping cart, with the total price of the purchase until the specific moment.

A navigation map that illustrates the functionalities inside the system is shown on figure 5.1:

After the initial analysis being done, the first non-functional prototype has been made, using Microsoft Power Point. Figure 5.2 shows a screen of this prototype.

Also, we prepared a test script for conducting the usability tests. This test script is composed by seven tasks and covers a complete usage of the prototype. The test script can be found on appendix A. With this prototype and the test script, we ran a pilot test, which has been useful to refine our test script for the first usability test. Also, the pilot test has been useful for detecting a first set of usability issues in the developed prototype and some of them were corrected before the first usability test. From the results of the pilot test, we were able to list the issues that were found by the users and we fixed the most important problems before running the first usability test. Appendix B shows the summary of the pilot test. After the pilot test, we made the first usability test of the prototype. A summary of the first usability test is located in appendix C.

Also, a usability heuristic evaluation has been conducted by the following colleagues: Victoria Blanco Alegria, Gabriel Lozano and Himanshu Saxena. The outcomes from this evaluation can be found in appendix D.

With the changes and fixes that have been suggested in the tests of the first non-functional prototype, the second non-functional prototype has been built. This version has been evaluated using the same test script as the previous version and

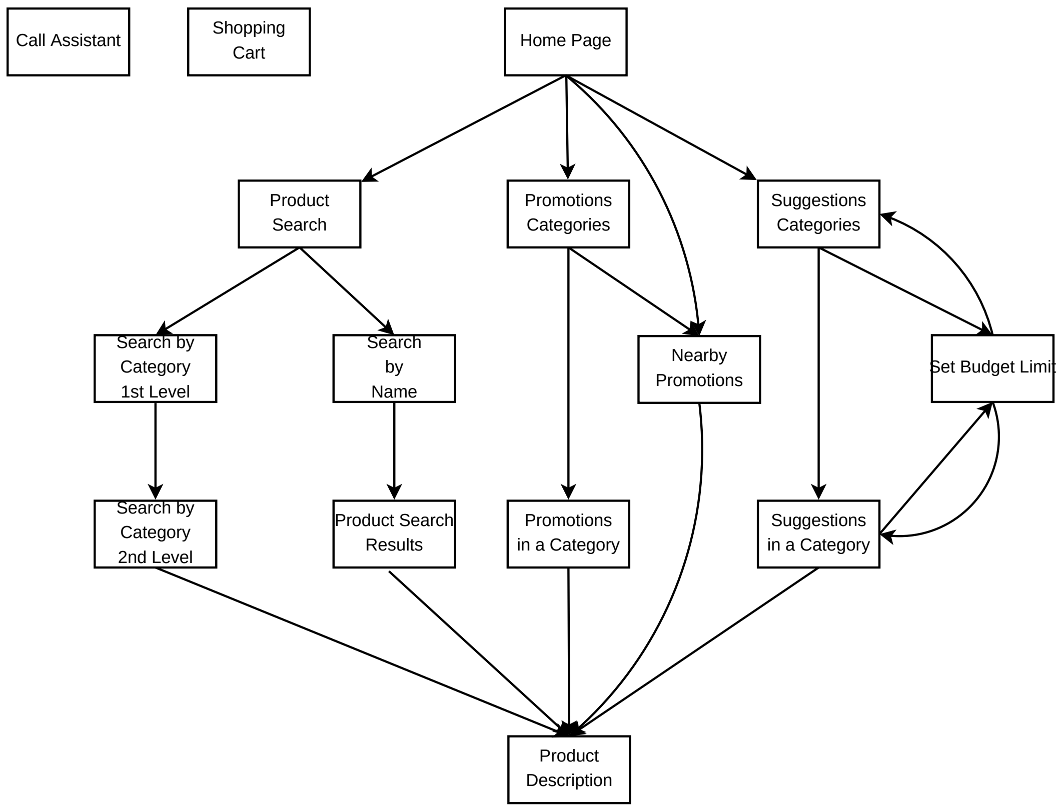


Figure 5.1: Navigation map for the non-functional prototype

the results, which are located in appendix E, have been taken into consideration for constructing the functional prototype.

Considering the results from the final usability evaluation of the second version of the non-functional prototype, we believe that it achieved a good level of usability, and with the minor fixes that have been proposed, we consider that the prototype has an improved usability level.

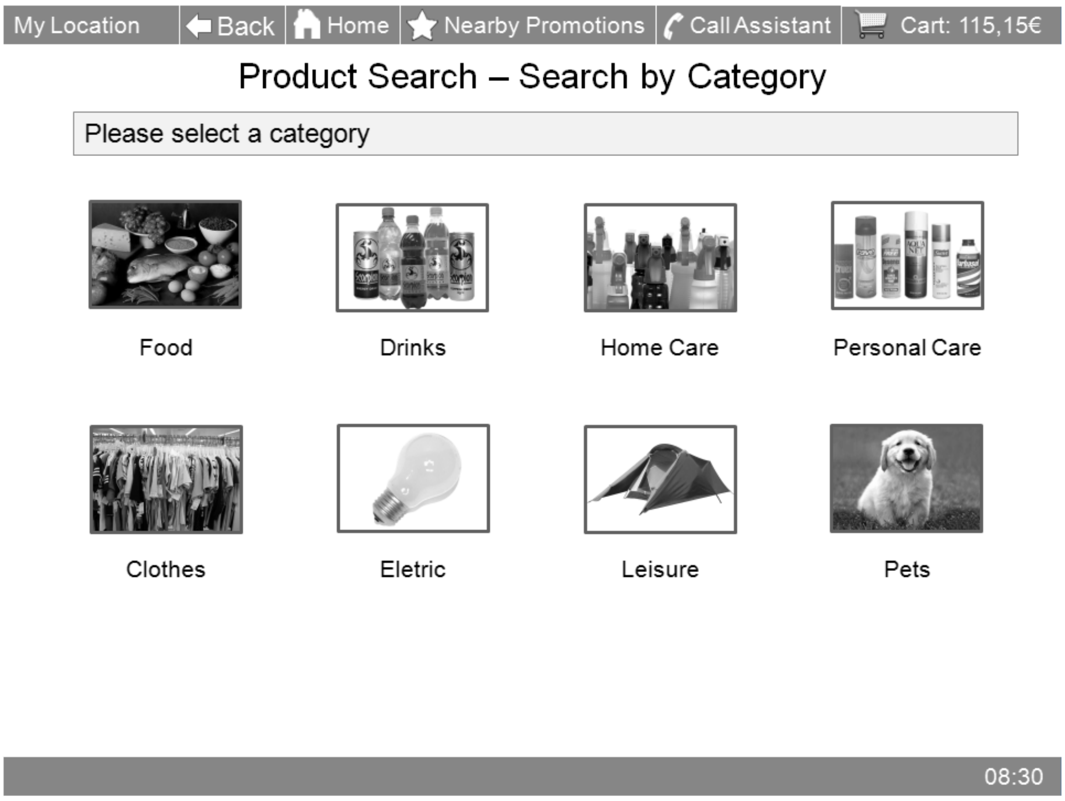


Figure 5.2: Non-functional prototype

5.2 GRASP Controller-Based Design

After the usability analysis, the next step was the elaboration of a software design based on the GRASP Controller pattern. With this design and the usability analysis, we built the first application and compared against another application that was developed in this research (described in Section 5.3). The applications have been developed in Java, using Swing components.

The software architecture followed the Model-View separation, which has been described in Section 2.1.10. This separation helps on keeping the user interface and the business logic isolated so we were able to manipulate them easily. As this research is focused on the interactions inside the user interface and the communication between user interface and business logic, we will not deal with data persistence in this report.

Considering the aim of this research, we have selected the two most representative tasks from the point of view of software architecture and relevance for the user: *Search Product by Category* and *Shopping Cart*.

5.2.1 Business Logic Layer

The business logic is basically composed by six main classes:

- **Corridor:** handles a corridor of the supermarket.
- **Shelf:** handles a shelf in a corridor of the supermarket.
- **Product:** represents a product that is being sold.
- **Category:** category of products. Contains products and sub-categories.
- **Sale:** represents a shopping cart.
- **LineItem:** handles a pair {Product, Quantity}.

Also, there are five auxiliary classes that have been created in order to apply patterns to improve the quality of the design. They are:

- **Element:** entity created to apply the Composite pattern [21] and simplifying the logic of the design.
- **ShoppingAssistant:** class related to the *Controller* pattern. It is the point of contact between the user interface and the business logic, handling all the requests that come from the user interface and go from the business logic to the user interface.

- **CategoryManager:** object manager for the categories and products. Loads all the items into the memory and is responsible of providing a particular element (category or product) that may be needed by any other object. Its usage removes the need to access the data source during the execution of the application, just needing to access it once.
- **CorridorManager:** works on the same way as the *CategoryManager*, but for corridors and shelves.
- **IUserInterface:** considering the point of view where the business logic is a package that does not depend on the user interface, the business logic shall not have any reference to an user interface object. However, the business logic may have an interface that lists the methods that need to be implemented by the main user interface class. As the main user interface class implements this interface, the business logic may call methods on the user interface by communicating through this interface.

Figure 5.3 shows the resulting class diagram.

The business logic layer has its boundary defined by two classes: *ShoppingAssistant* and *IUserInterface*. Figure 5.4 illustrates this boundary and, from that we can see that there is just one point of entrance, which is the *ShoppingAssistant* controller. The output is also centralized into the *ShoppingAssistant* class, who communicates then with the user interface through the *IUserInterface* interface.

The retrieval of the sub categories is an example of how the two elements of the boundary of business logic operate. Figure 5.5 shows the sequence of actions that are related to the listing of elements (sub-categories and products) that a category may have. On the user interface side, when the user interface shows a list of categories, it keeps record of those elements in a hashtable, which is composed by the id of the element and a reference to the element itself (figure 5.9 shows an example of this hashtable). When a user selects an element, the user interface retrieves the corresponding element object from the hashtable and informs the controller that the user has chosen the retrieved element. Then, the controller gets the element and calls the method "select", which has different implementations on a category and on a product. In this case, the element is a category, the execution of "select" method calls the method "showElements" in the *ShoppingAssistant*, which passes this request to the user interface, through *IUserInterface* interface.

One alternative behavior is when the element is a product. In this case, the method "select" of the product calls the "showProduct" method of *ShoppingAssistant*, which passes it to the user interface.

Figures 5.6 and 5.7 show diagrams that are related to the shopping cart functionality. Figure 5.6 describes the action of inserting a product in the shopping cart and figure 5.7 shows the sequence of actions that are taken to show the shopping

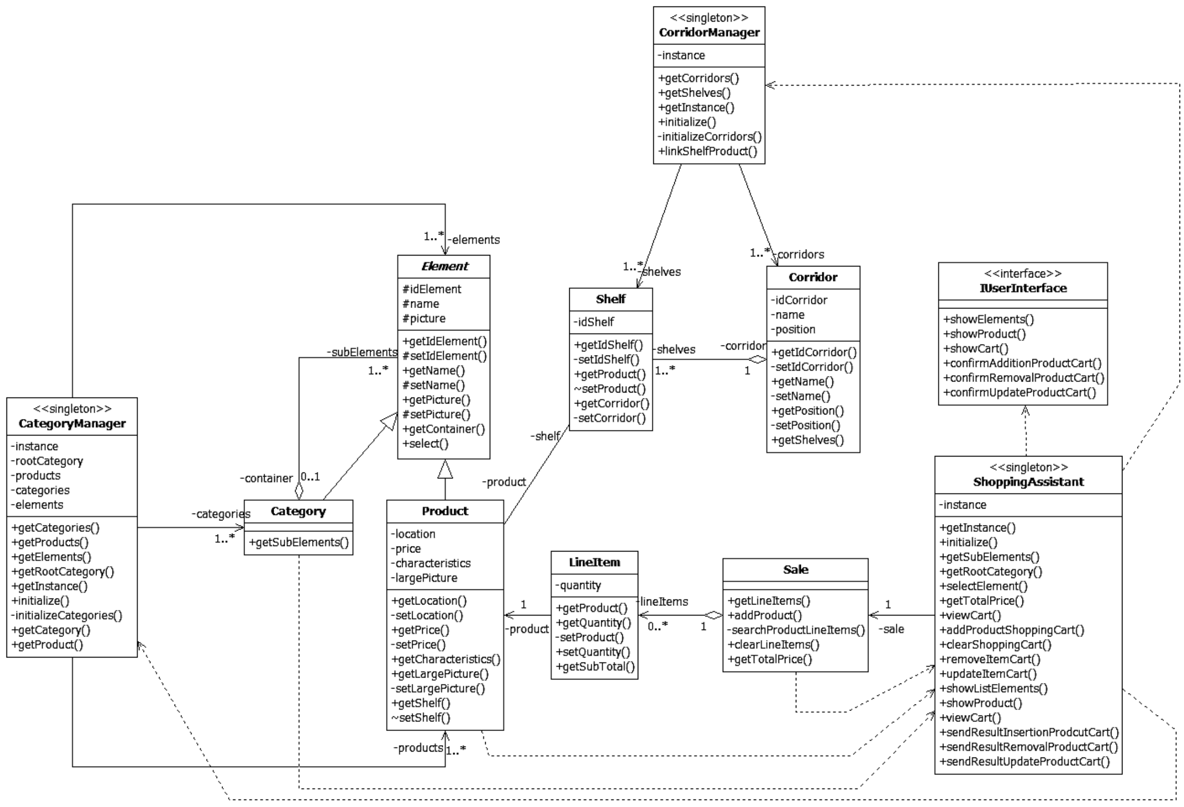


Figure 5.3: Class Diagram for business logic for GRASP Prototype

cart. The first case involves an interaction similar to the one shown in figure 5.5 with the addition of creation of elements, according to *Creator* pattern. The second case is a shorter interaction, as the *ShoppingAssistant* controller just makes a call to the user interface to display the information related to a sale.

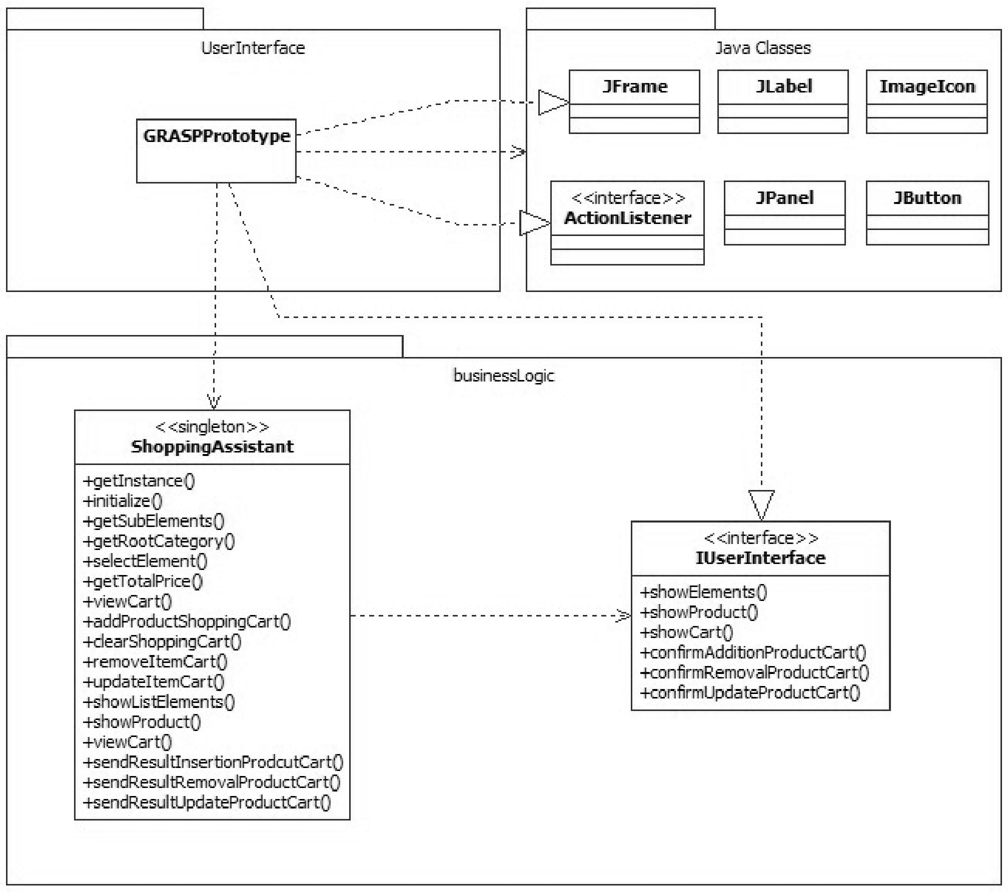
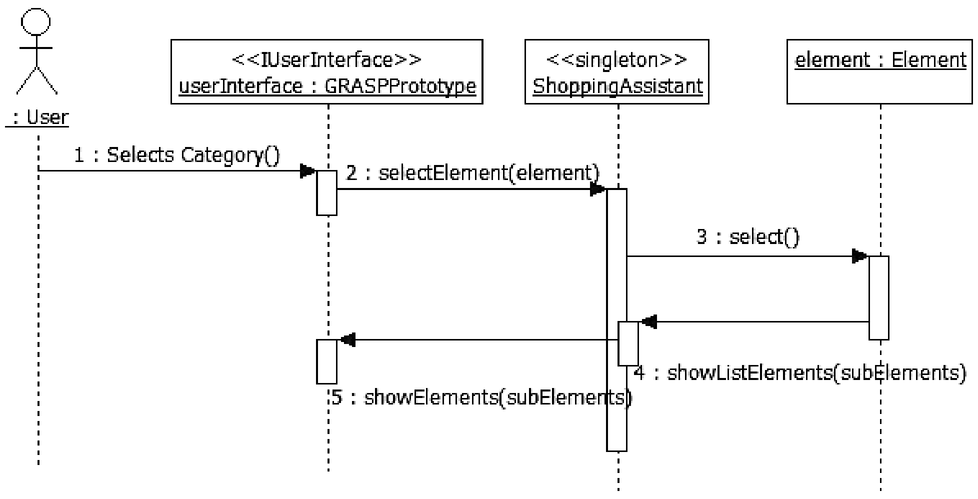
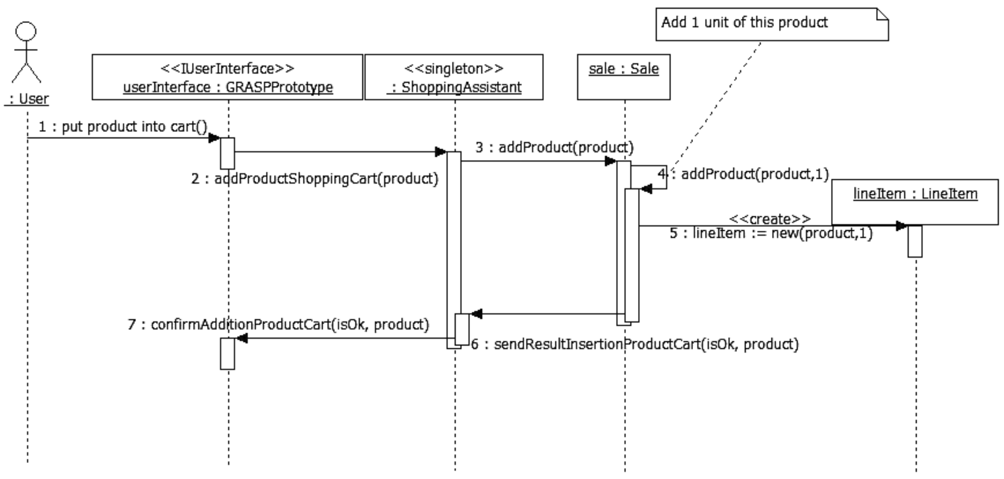


Figure 5.4: Boundary of business logic layer on the GRASP prototype



: User
<<IUserInterface>>
userInterface : GRASPPrototype
<<singleton>>
ShoppingAssistant
element : Element
1 : Selects Category()
2 : selectElement(element)
3 : select()
4 : showListElements(subElements)
5 : showElements(subElements)

Figure 5.5: Selection of sub-elements on the GRASP prototype



Add 1 unit of this product
: User
<<IUserInterface>>
userInterface : GRASPPrototype
<<singleton>>
: ShoppingAssistant
sale : Sale
lineItem : LineItem
1 : put product into cart()
2 : addProductShoppingCart(product)
3 : addProduct(product)
4 : addProduct(product,1)
<<create>>
5 : lineItem := new(product,1)
6 : sendResultInsertionProductCart(isOk, product)
7 : confirmAdditionProductCart(isOk, product)

Figure 5.6: Inserting a product in the shopping cart on the GRASP prototype

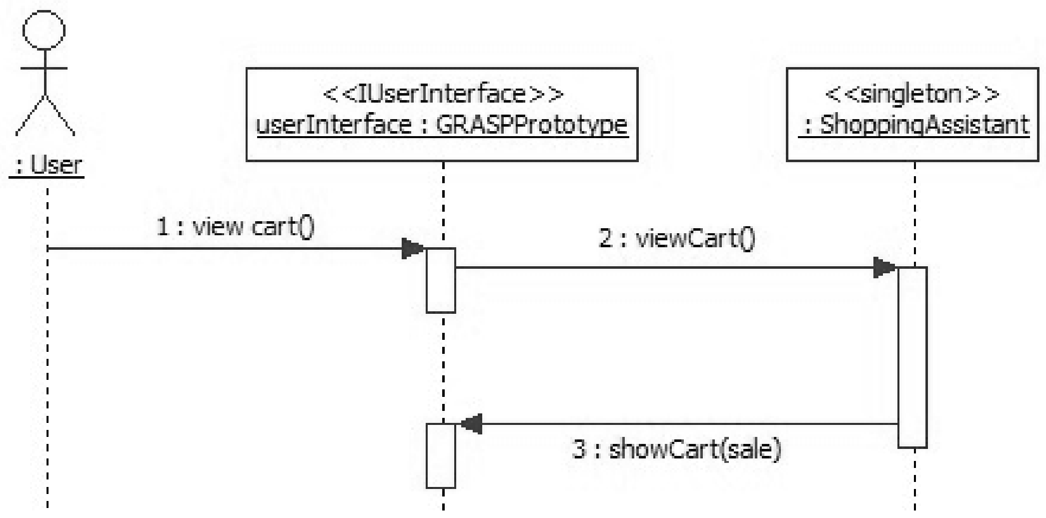


Figure 5.7: List of items in the shopping cart on the GRASP prototype

5.2.2 Presentation Layer

The user interface is composed by a main class, called *GRASPPrototype*, which implements the interface *IUserInterface* from the business logic. Also, this class extends the *JFrame* interface, which belongs to the Java Swing package (and is necessary for creating a window) and implements the *ActionListener* class, which is responsible for handling the events that are created by the inputs from the user.

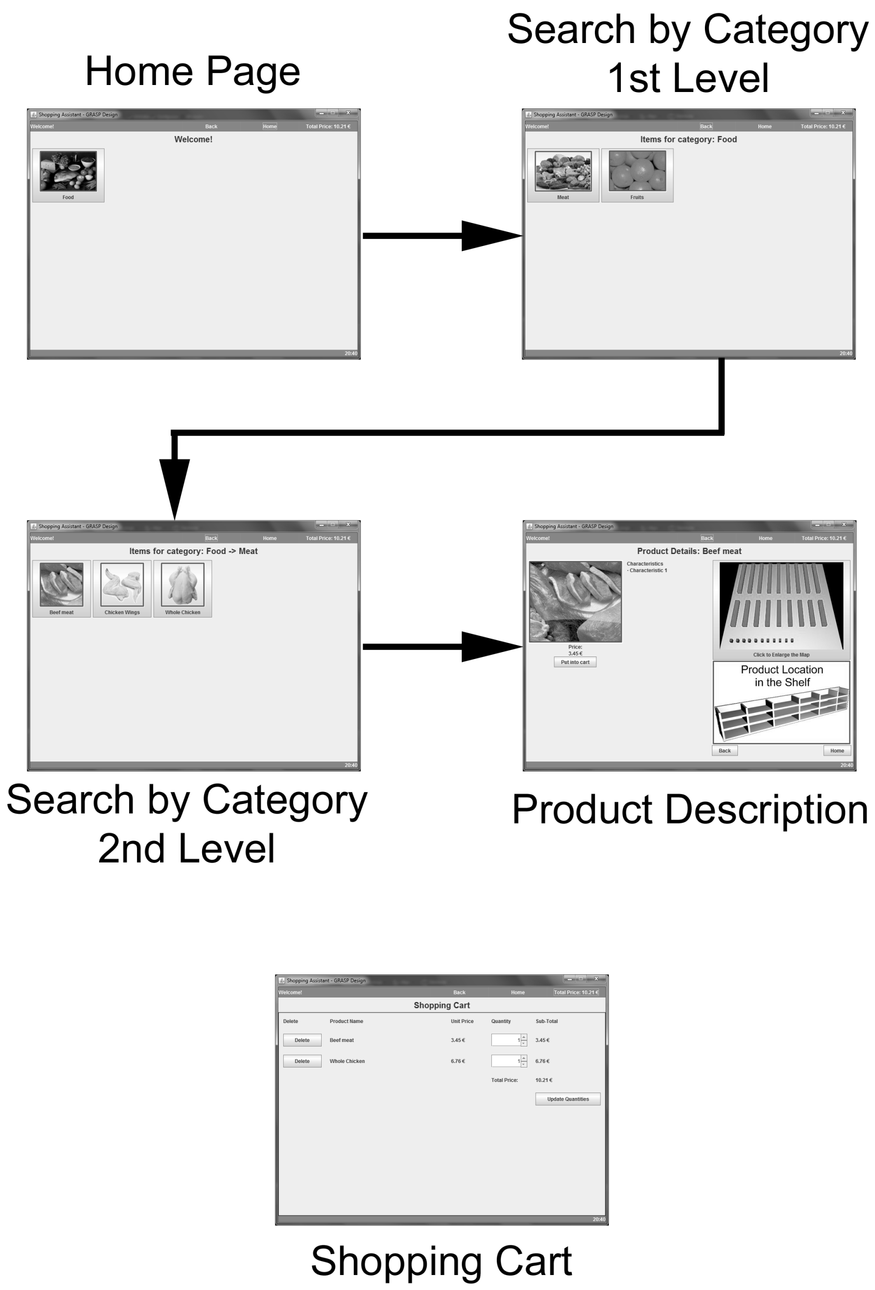
The screen is divided into three main sections, being two (top and bottom) responsible for smaller actions and one (middle) responsible for the majority of the system. Their description follows below:

- **headerPanel:** located in the top part of the screen and contains three actions: Back, Home and View Cart.
- **footerPanel:** located in the bottom section of the screen and contains a clock.
- **mainPanel:** located in the middle of the screen and responsible for the main part of the system.

The navigation map is illustrated on figure 5.8.

As a consequence of implementing the *IUserInterface* interface, the *GRASPPrototype* class has to implement the following methods:

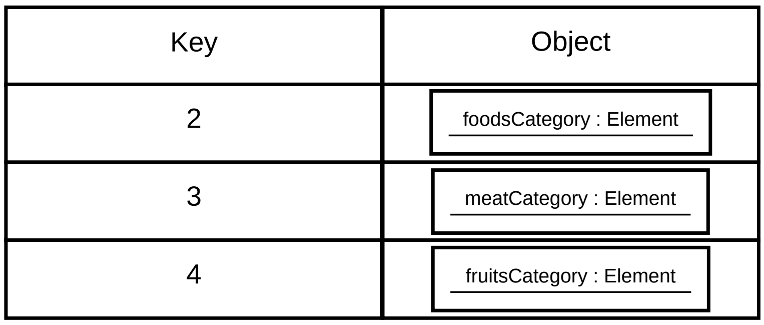
- **showElements:** shows a list of elements into the screen. Each element is described by a picture and a name, encapsulated by a button that, when clicked, tells the business logic (through *ShoppingAssistant* controller) that an element has been selected by the user. Also, it stores in a hashtable the elements that are being displayed, using their ids as the keys for indexing on the hashtable. An example of the hashtable can be found on figure 5.9. The ids are then used inside the command arguments for the buttons on the screen. When the user selects an element, or wants to go the previous screen, the ids are extracted from the command arguments and the hashtable is then accessed to provide the elements that shall be used to perform the required actions. This hashtable is used on the whole application, for at least "Back" and "Home" navigations, however, it is in "showElements" where it has the largest usage. Figure 5.10 shows the actions involved on this method.
- **showProduct:** shows the information regarding a specific product, which is composed by: product picture, price, description and position inside the supermarket.
- **showCart:** shows the list of products that are inside the shopping cart, their unit prices, quantity and total price (for each product and for the whole



Shopping Cart

Delete	Product Name	Unit Price	Quantity	Sub-Total
Delete	Beef meat	3.45 €	1	3.45 €
Delete	Whole Chicken	6.76 €	1	6.76 €
			Total Price:	10.21 €

Figure 5.8: Navigation Map for the Functional Prototypes



Key	Object
2	foodsCategory : Element
3	meatCategory : Element
4	fruitsCategory : Element

Figure 5.9: Example of Hashtable on user interface of the GRASP prototype

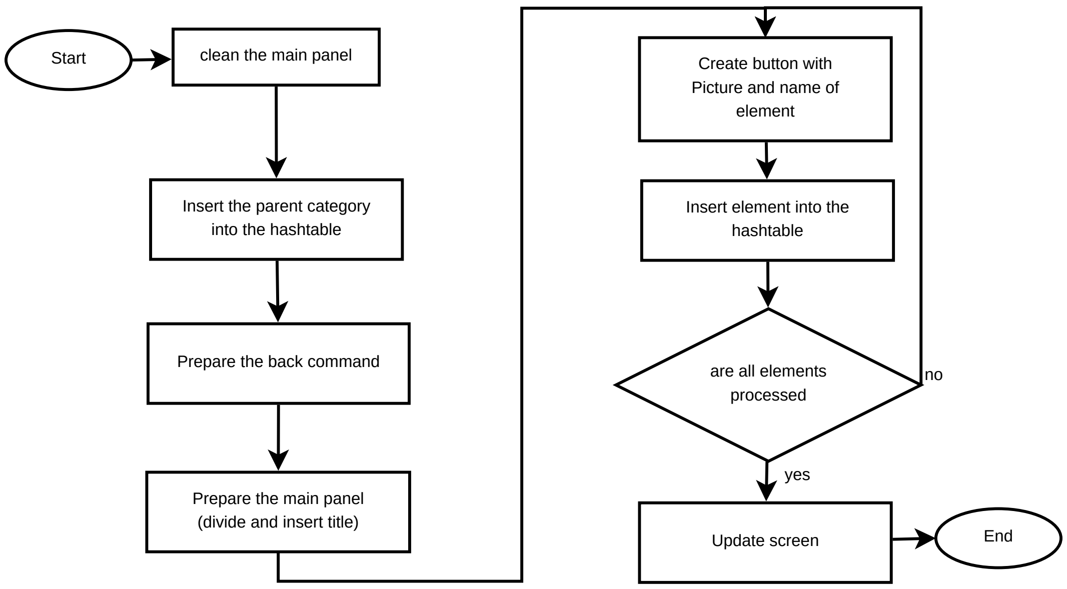

Start
clean the main panel
Insert the parent category into the hashtable
Prepare the back command
Prepare the main panel (divide and insert title)
Create button with Picture and name of element
Insert element into the hashtable
are all elements processed
no
yes
Update screen
End

Figure 5.10: Actions for listing the elements on the screen of the GRASP prototype

purchase). Also, it keeps a record of the items in the shopping cart. This record is needed to perform the actions of delete and update items.

- **confirmAdditionProductCart:** confirms that the insertion of a product inside the shopping cart has been done successfully.
- **confirmRemovalProductCart:** confirms that the removal of a product inside the shopping cart has been done successfully.
- **confirmUpdateProductCart:** confirms that the update of the shopping cart has been done successfully.

In order to illustrate the objects that exist in a screen, figure 5.11 shows a diagram with the objects on the “Show Product” screen and the relationships among them.

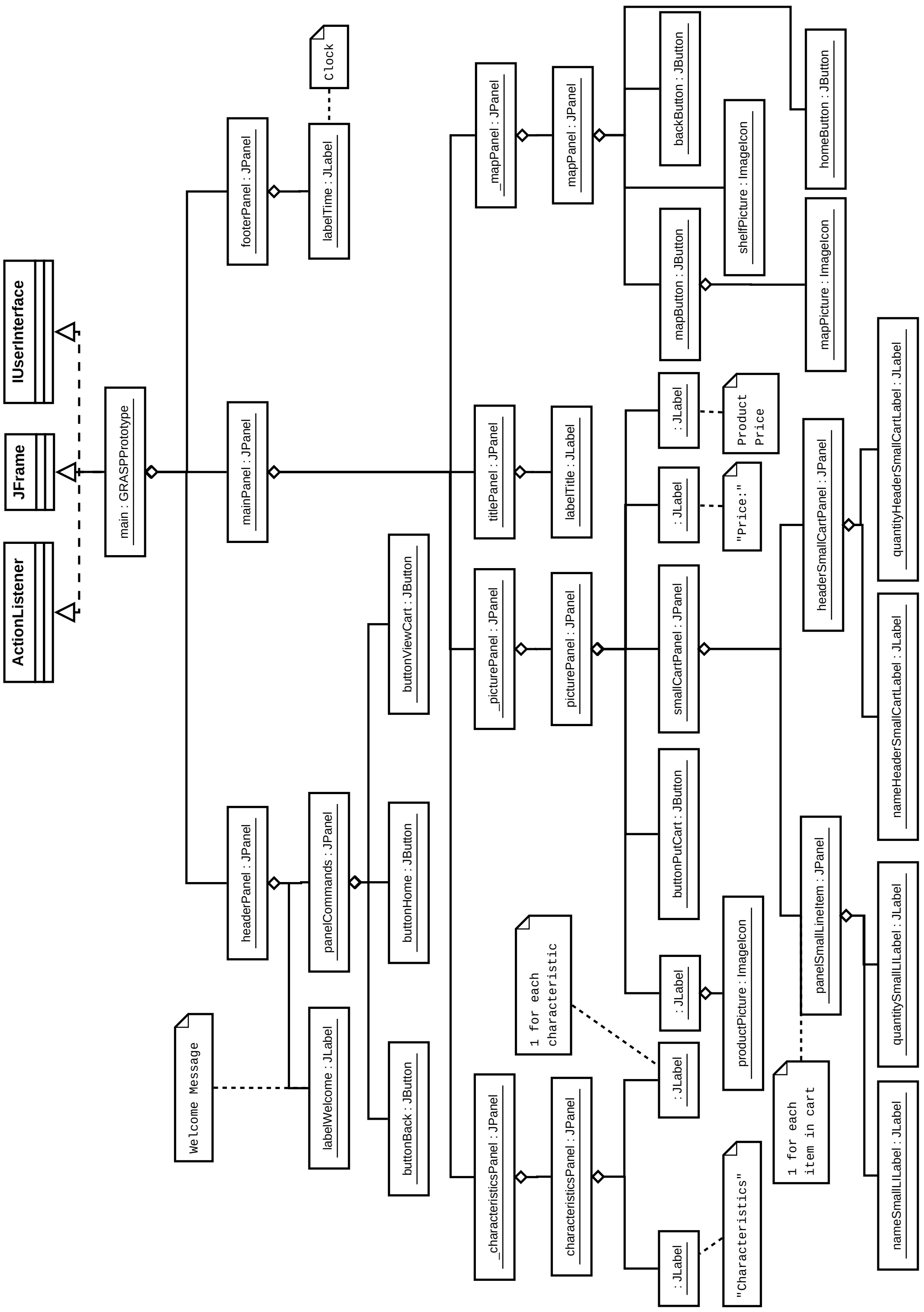


Figure 5.11: Objects in *Show Product* Screen of the GRASP prototype

5.3 Model-View-Controller-Based Design

This prototype has been built using the Model-View-Controller pattern and aims to provide the same functionalities and user interface of the GRASP-based prototype.

As described in the Section 2.2, the Model-View-Controller pattern divides the user interface into View and Controller and keeps the Model being represented by the entities inside the business logic.

5.3.1 Business Logic Layer

The business logic (model) in this prototype is composed by eleven classes, which are:

- **Corridor:** handles a corridor of the supermarket.
- **Shelf:** handles a shelf in a corridor of the supermarket.
- **Product:** represents a product that is being sold.
- **Category:** category of products. Contains products and sub-categories.
- **Sale:** represents a shopping cart.
- **LineItem:** handles a pair {Product, Quantity}.
- **Element:** entity created to apply the Composite pattern [21] and simplifying the logic of the design.
- **CategoryManager:** object manager for the categories and products. Loads all the items into the memory and is responsible of providing a particular element (category or product) that may be needed by any other object. Its usage removes the need to access the data source during the execution of the application, just needing to access it once.
- **CorridorManager:** works on the same way as the *CategoryManager*, but for corridors and shelves.
- **Observable:** class created for implementing the Observer pattern [21]. It provides the operations related to controlling the observers and sending messages to them.
- **IObserver:** interface that the observers shall implement in order to apply the Observer pattern.

Figure 5.12 shows the class diagram for the model of this prototype.

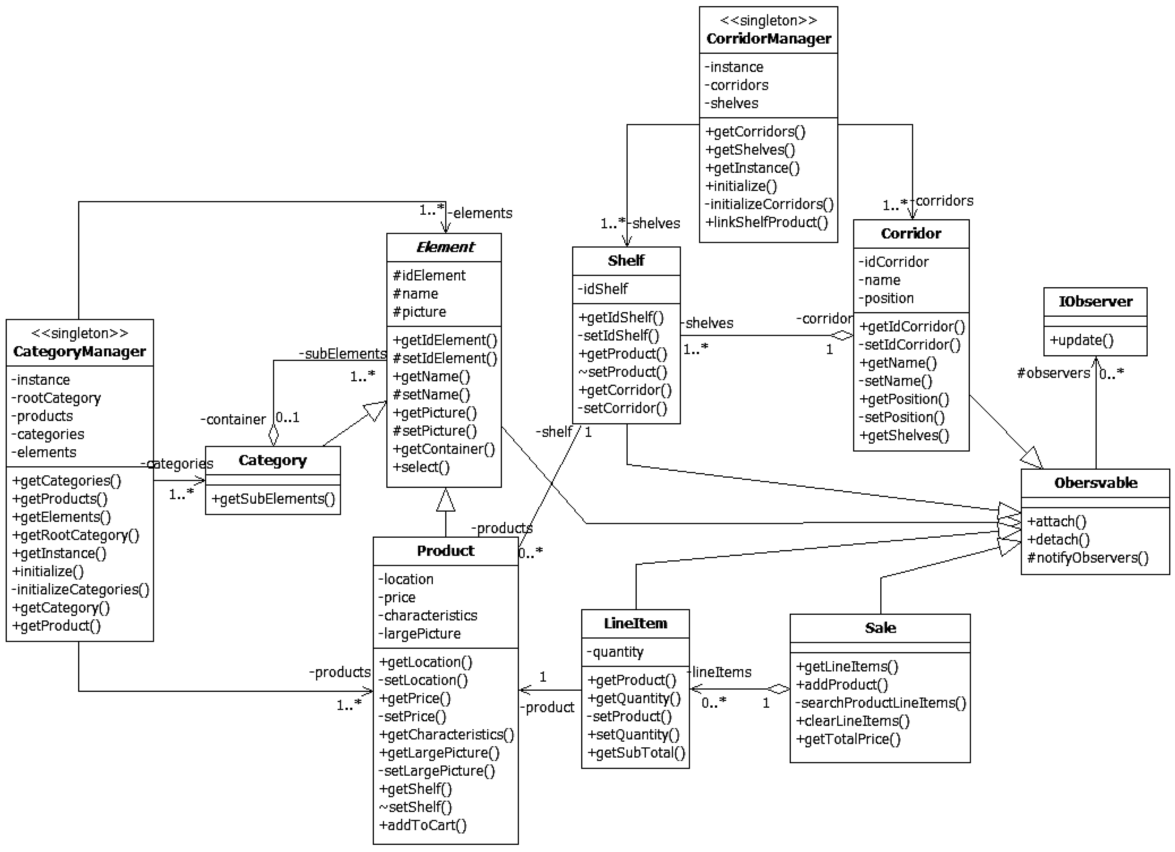


Figure 5.12: Business logic class diagram for the MVC prototype

5.3.2 Presentation Layer

The presentation layer of the prototype is composed by three main groups of classes, which are:

- **Views:** the views of the system. Display the information to the users.
- **Controllers:** the controllers of the system. Manipulate the user inputs, executing the necessary actions.
- **Auxiliary elements:** provide support for the operation of views and controllers.

Figure 5.13 shows a class diagram with the relationships that exist among the user interface elements.

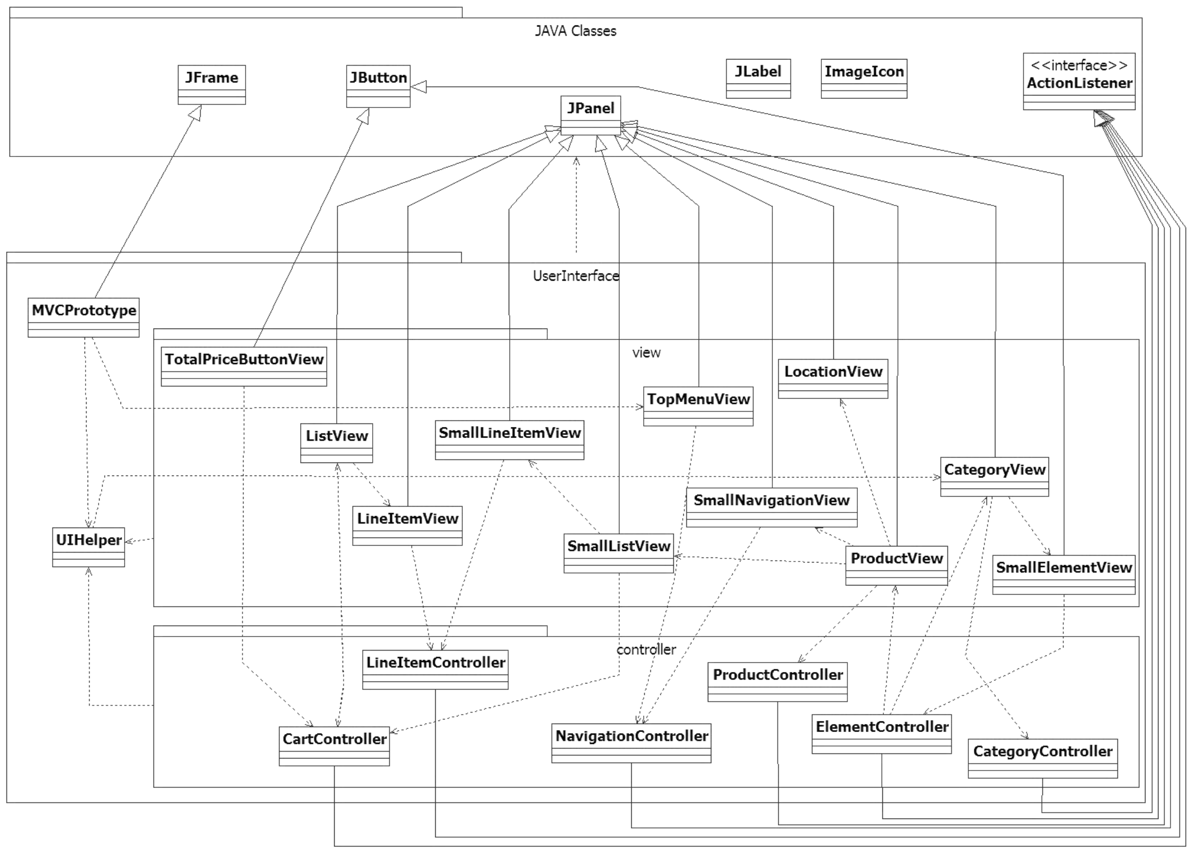


Figure 5.13: User interface class diagram for the MVC prototype

Views

One characteristic of the usage of MVC is that the user interface shall have at least one view for each object from the model that has a representation inside the user interface. Considering that we have aggregations in the model, it becomes possible having an arrangement where the user interface is composed by views, which are composed by other views. One example is shown on figure 5.21 which is composed by a view that is used to list the products in the shopping cart in a compact manner, by showing the name and quantity of each product inside the shopping cart and is composed by smaller views, being each one of them responsible for showing the name and quantity of a single product inside the shopping cart.

Also, in order to build the user interface, there are some parts that need to be inserted into the user interface but are not related to the model. For that, some artificial views were created. Those views are related to an artificial controller. The views are: *TopMenuView* and *SmallNavigationView* and the controller is *NavigationController*.

For better understanding, the views that have been developed in the prototype are described below.

TopMenuView

Displays the top menu of the prototype, with its actions: *Back*, *Home* and *Total Price / View Cart*. Figure 5.14 shows a representation of this view. Sub-view: *TotalPriceButtonView*.

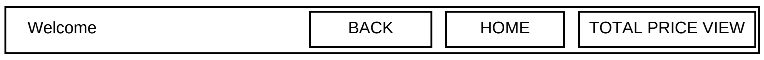


Figure 5.14: TopMenuView for the MVC prototype

TotalPriceButtonView

Displays the actual total price of the purchase and, when clicked, shows the shopping cart. It does not have any sub-views.

CategoryView

Displays the elements that belong to a specific category. Figure 5.15 shows a representation of this view. Sub-view: *SmallElementView*

Figure 5.16 shows a screen capture of this view in the execution of the prototype.

SmallElementView

Displays an element in a form of a clickable button. This button selects the specific element to be displayed in the application.

ProductView

Displays the information of a product. Figure 5.17 shows a representation of this view. Sub-views: *LocationView*, *SmallNavigationView* and *SmallListView*.

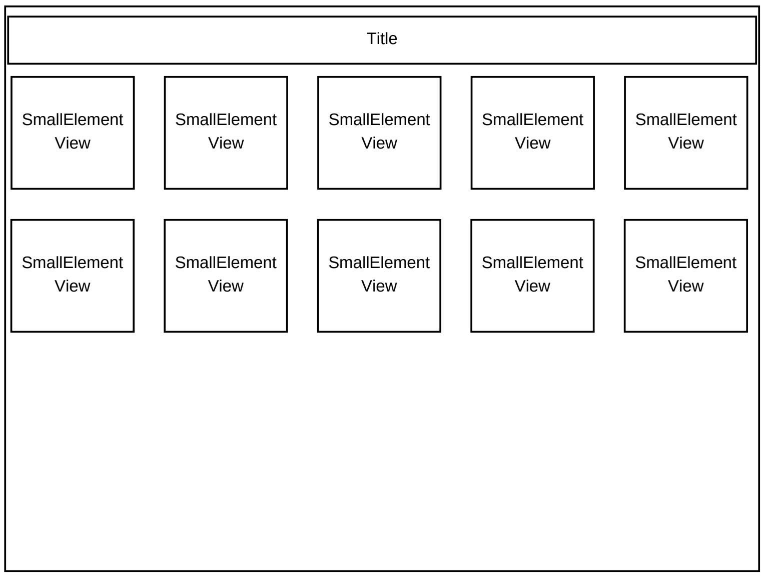


Figure 5.15: CategoryView for the MVC prototype

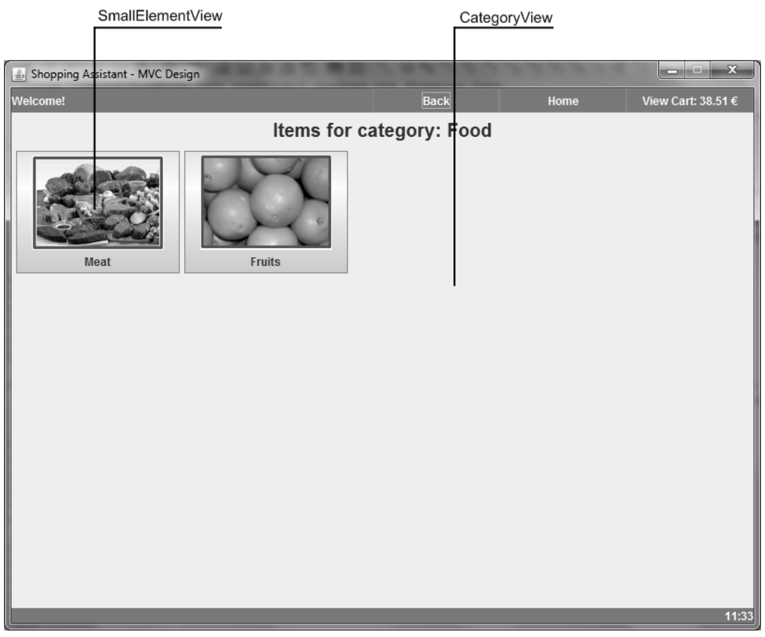


Figure 5.16: CategoryView: List of sub-categories and products on the MVC prototype

Figure 5.18 shows a screen capture of the prototype with the main elements highlighted.

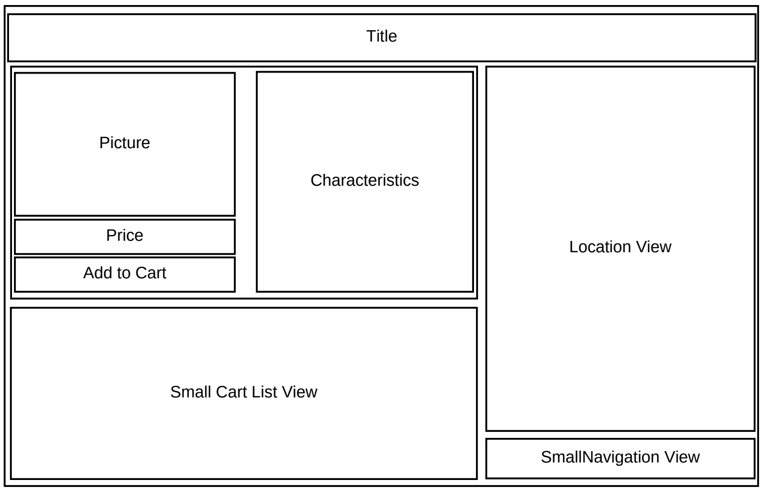
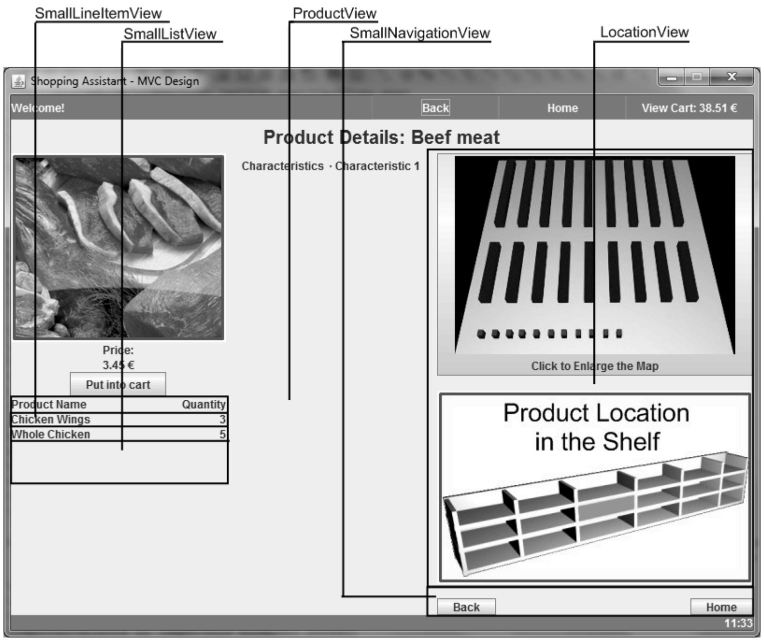


Figure 5.17: ProductView for the MVC prototype



Product Name	Quantity
Chicken Wings	3
Whole Chicken	5

Figure 5.18: ProductView: Product information, location and small shopping cart list on the MVC prototype

Figure 5.19 illustrates the objects that exist when showing the “Show Product” screen. It is composed by the *ProductView* together with *TopMenuView* and the footer.

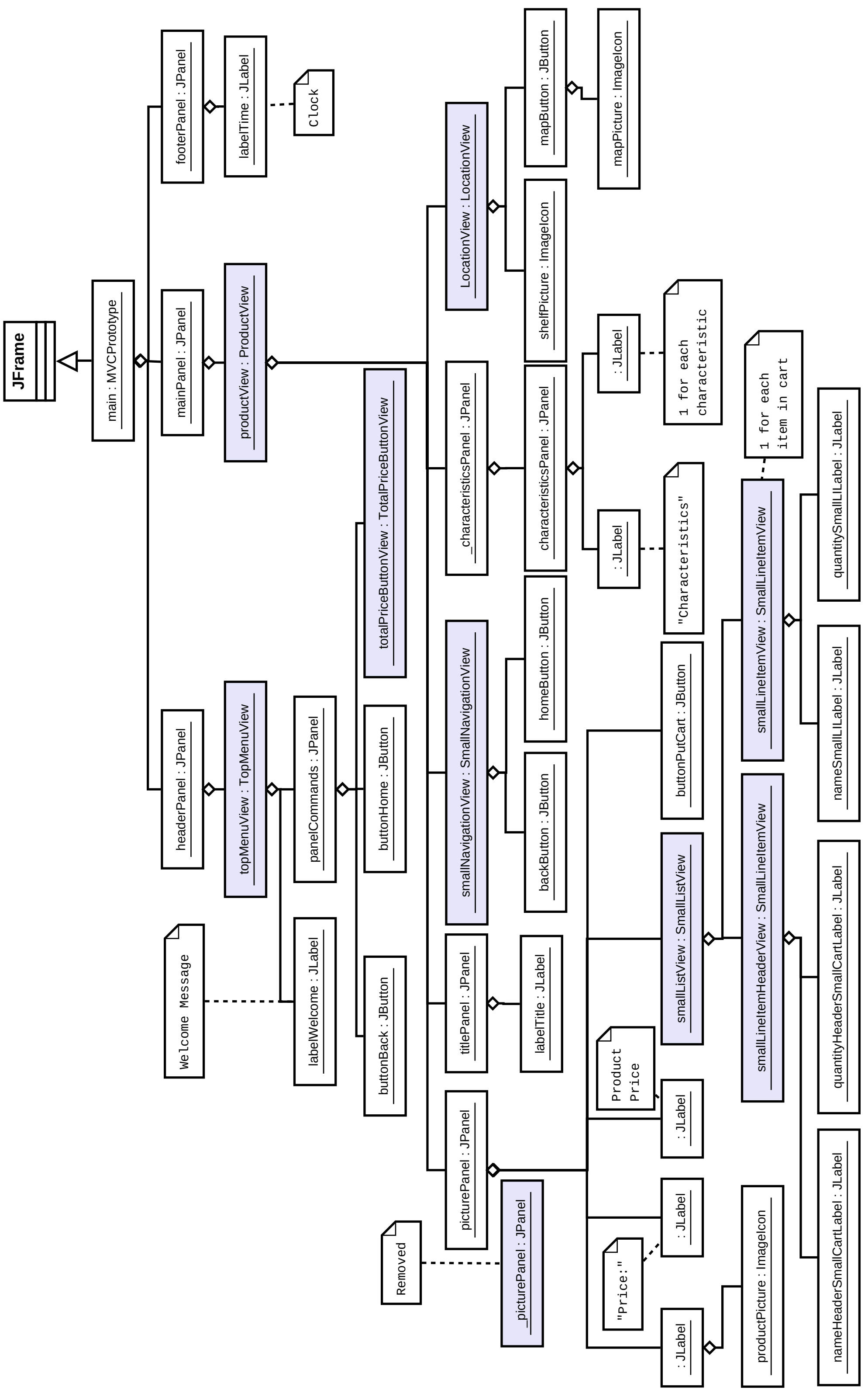


Figure 5.19: Objects in *Show Product* Screen of the MVC prototype

LocationView

Displays the location of a product.

SmallNavigationView

Provides the options of going back and going to the home of the application.

SmallListView

Displays a small list of the products inside the shopping cart. Figure 5.20 shows a representation of this view. Sub-view: *SmallLineItemView*.

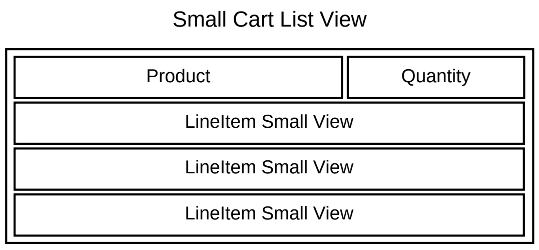
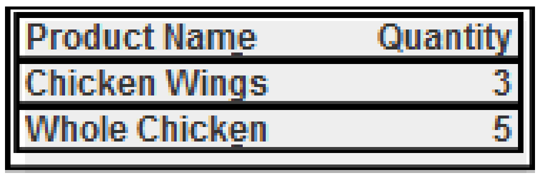


Figure 5.20: SmallListView for the MVC prototype

Figure 5.21 shows a screen capture of this view in the execution of the prototype.



Product Name	Quantity
Chicken Wings	3
Whole Chicken	5

Figure 5.21: Small shopping cart list: reduced list of products on the shopping cart for the MVC prototype

SmallLineItemView

Displays the information of a product in the shopping cart in a smaller view.

ListView

Displays a list of the products inside the shopping cart. Figure 5.22 shows a representation of this view. Sub-view: *LineItemView*.

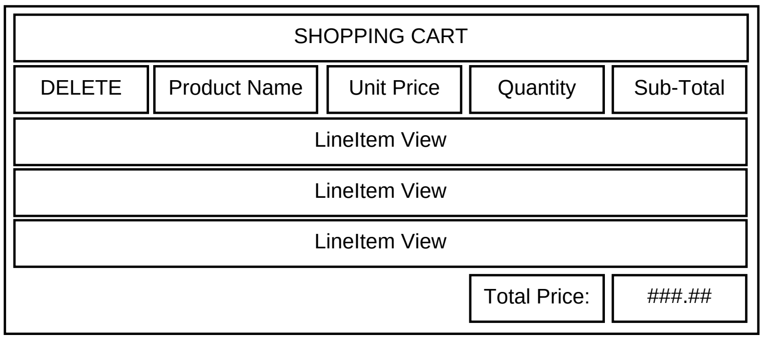


Figure 5.22: ListView for the MVC prototype

LineItemView
Displays the information of a product inside the shopping cart.

Auxiliary Elements

This prototype has three auxiliary elements, which have been developed to help on the development of the user interface. They are described below.

MVCPrototype
The *MVCPrototype* class provides the initialization of the prototype and is the main container for the views. When the application starts, there is a set of actions that takes place. They are:

1. **Main window creation:** creates the main window of the system.
2. **Main window initialization:** initializes the main window: divides it into header, main panel and footer.
3. **Object managers initialization:** initializes the two object managers: CategoryManager and CorridorManager.
4. **Shopping Cart creation:** creates the Shopping Cart.
5. **Shopping Cart initialization:** inserts the initial set of products into the Shopping Cart.
6. **UIHelper creation:** creates the UI Helper, which is an object that helps on user interface manipulation.

UIHelper
The UIHelper is an object that is referenced by all views and controllers and provides the following functionalities:

- Back action: go to the previous screen.
- Home action: go to the list of first-level categories.
- Shopping Cart: provides access to the shopping cart.
- Change Main Panel: change the view on the main panel of the screen.

About the *Back* functionality, there is an issue that has been found when developing the MVC prototype. The reason is that, considering that the element that is in charge for changing elements is the controller, it needs to know which was the previous step on the process. Given the fact that in the design there was not a session manager to store the actions of the users, the solution that has been found is implementing a stack that stores the views that are displayed in the main panel based on the user navigation. This stack is stored and controlled in the *UIHelper* object and one example of its usage is shown in figure 5.23. In this example, the actions (selections) that the user takes are shown in the arrows and the last action is going to the previous screen (Back).

Navigation Controller

The *NavigationController* is used on views that perform “Back” and “Home” actions (through UIHelper), which are actions that are related to navigation and its usage is aimed on providing a better navigation of the system.

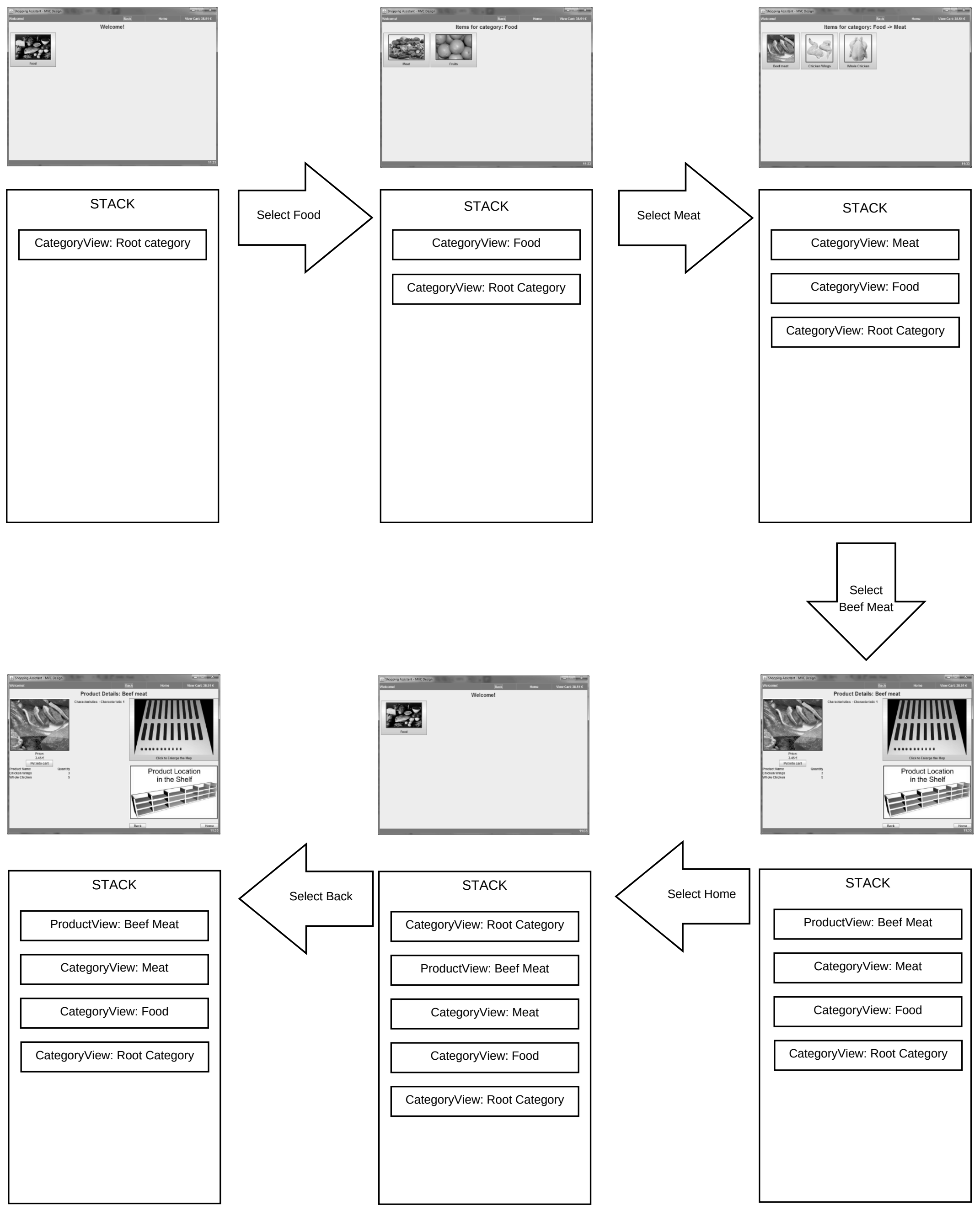


Figure 5.23: Back stack in the MVC prototype

5.3.3 Communication between User Interface and Business Logic

The communication between user interface and business logic is through the connexion that exists between the model, view and controller elements. In order to illustrate this communication, two examples are described below.

Select Category

In this case, the operation starts when the user selects a new category to be displayed from the listing of the elements that belong to a category as shown in figure 5.16. In this operation, when the user selects a sub-element, he/she clicks in a button that is a view related to this sub-element. Then, the controller related to this sub-element creates a new view for this sub-element (a *CategoryView* or a *ProductView*), which is then changed into the main window of the prototype. The sequence of actions that occur is displayed on figure 5.24.

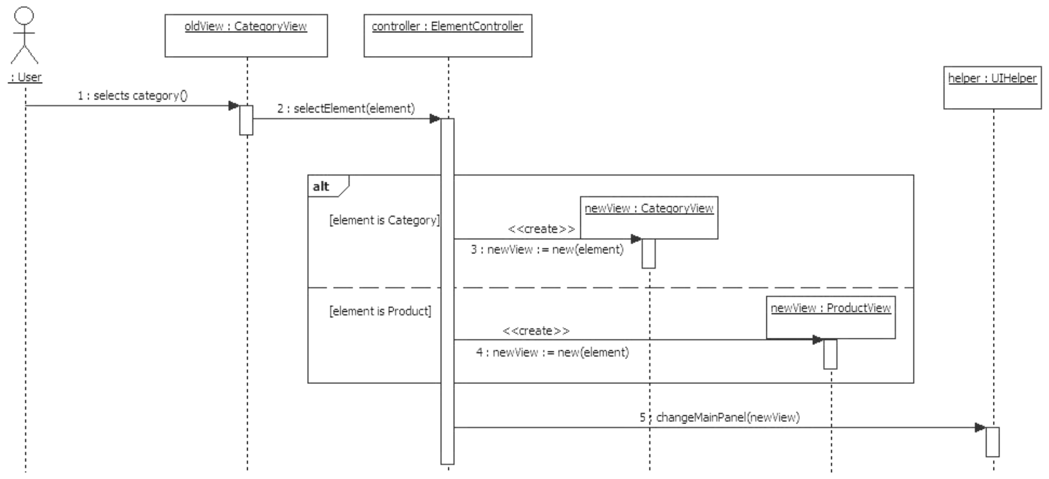


Figure 5.24: Select Category sequence diagram for MVC prototype.

Add Product to the Shopping Cart

In this example, the operation starts when the user orders the system to add a new product into the shopping cart through a button inside the *ProductView*. Then, the *ProductController* detects this click and asks its product object to add itself into the shopping cart. The product object then asks the shopping cart to add it into the cart. The shopping cart then creates a new LineItem related to this product and, after that, notifies its observers that it (the shopping cart) has changed. The observers then receive this notification and refresh themselves in order to display the updated information. This sequence of actions is displayed on figure 5.25.

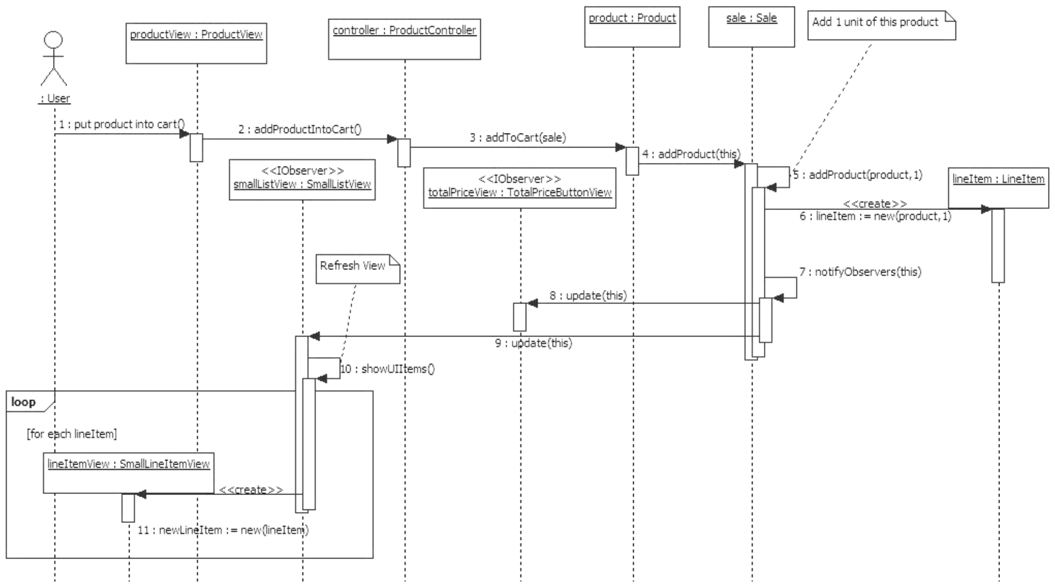

productView : ProductView
controller : ProductController
product : Product
sale : Sale
Add 1 unit of this product
: User
1 : put product into cart()
2 : addProductIntoCart()
3 : addToCart(sale)
4 : addProduct(this)
5 : addProduct(product, 1)
lineItem : LineItem
<<IObserver>> smallListView : SmallListView
<<IObserver>> totalPriceView : TotalPriceButtonView
<<create>>
6 : lineItem := new(product, 1)
Refresh View
7 : notifyObservers(this)
8 : update(this)
9 : update(this)
10 : showUIItems()
loop
[for each lineItem]
lineItemView : SmallLineItemView
<<create>>
11 : newLineItem := new(lineItem)

Figure 5.25: MVC version of "Add Product into Cart" functionality

Chapter 6

Analysis

After developing the functional prototypes, they have been analyzed in two different ways. Firstly, their designs have been compared and the differences were discussed. After that, they have been analyzed according to the usability heuristics that were proposed by Nielsen [38] and are described in section 2.3.4.

6.1 Differences between GRASP and MVC Prototypes

In this section, the differences between the GRASP and MVC prototypes are discussed. This discussion is divided into two parts: the first one discuss about the differences in the business logic and the second part discuss about the changes in the user interface.

6.1.1 Business Logic layer

The design using MVC pattern has three main differences on the business logic when compared with the GRASP-based design. They are:

- **Usage of GRASP controller:** in the MVC design the GRASP controller, which is the element that centralizes the communication between the business logic and the user interface, is not useful as the communication between user interface and business logic needs to be made through each set of model, view and controller.

- **Sending of messages from Business Logic to User Interface:** in the first prototype the classes on the user interface implement an interface that is inside the business logic (called IUserInterface) and has a reference inside the ShoppingAssistant controller, which redirects the messages that are sent by the other business logic entities. However, on the MVC prototype, the user interface classes implement another interface (called IObserver) and, each object from the model receives a "subscription" from each view and controller that are related to this specific object. When an object needs to inform the user interface that it has been updated, it sends a message to each one of the subscribed objects (views and controllers) that it has been updated. Based on that, each view and controller takes the necessary action, for example refreshing themselves. For comparison, figure 6.1 shows the case of *add a product into the shopping cart* where we can see the differences that exist between the two designs.

- **"Add Product into Cart" functionality:** in the GRASP prototype, the cart is asked to add a specific product and, in the MVC prototype, the product is being asked to add itself into the cart. This difference comes from the fact that in the MVC prototype, the order to add a product into the shopping cart comes from a product view and, therefore, this request is handled by the *ProductController*, which passes this request to the product model object. The sequence of action in the two designs can be seen on figure 6.1.

6.1.2 User Interface

When comparing the two developed designs in terms of user interface, four main differences were found. They are:

- **Granularity of the user interface:** the Model-View-Controller design has a finer granularity of user interface classes when compared with the GRASP design. It comes from the fact that in MVC, there is the necessity of having one view object for each model object whose information is displayed on the screen. In this way, there are a larger amount of interactions inside the system in the MVC prototype. Figure 6.2 shows this difference. This granularity brings an immediate drawback, which is the larger amount of method calls. However, the design becomes easier to implement, analyze and maintain because in the MVC design, as the implemented methods are smaller, they are simpler than the ones in the GRASP design.

- **Back action:** for implementing the *Back* action, the prototypes operate in different ways. In GRASP prototype, in every screen the elements that are responsible for the *Back* functionality knows which is the previous screen and it is enough for executing this action. For example: when the system is showing the characteristics of a product, the *Back* action is supposed to take the user into the list of products inside the product's category. However, in the

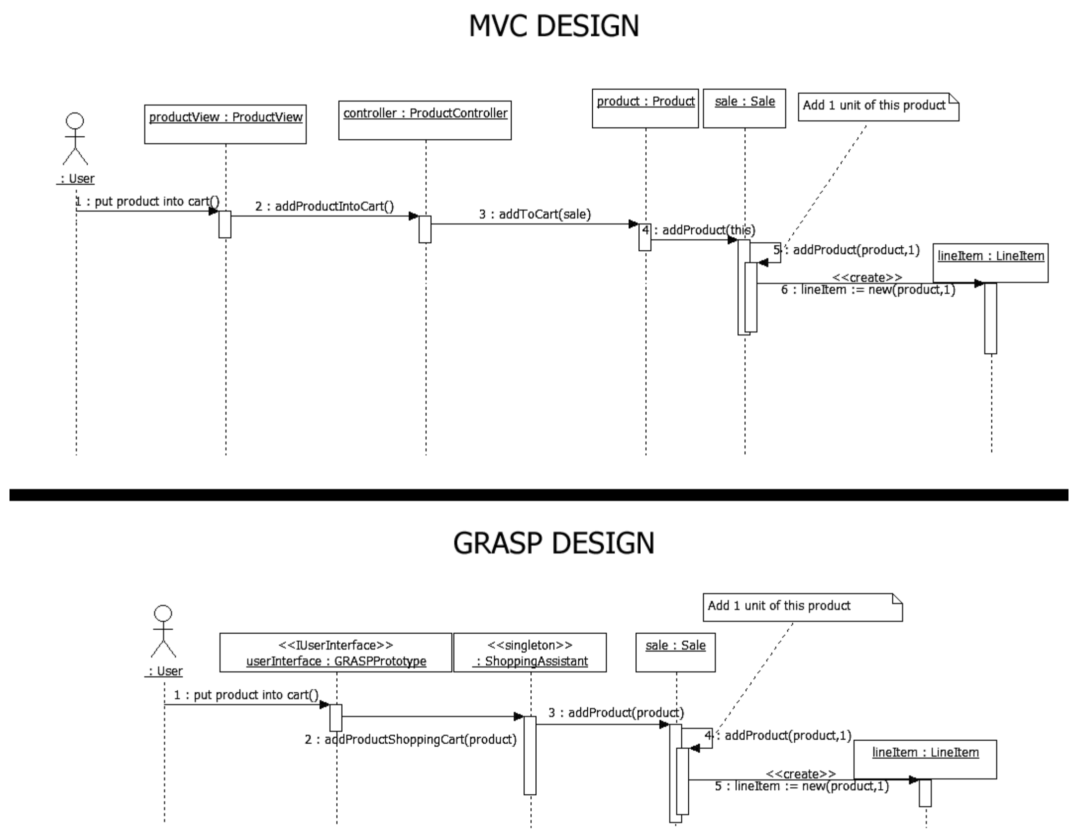

MVC DESIGN
: User
productView : ProductView
controller : ProductController
product : Product
sale : Sale
Add 1 unit of this product
1 : put product into cart()
2 : addProductIntoCart()
3 : addToCart(sale)
4 : addProduct(this)
5 : addProduct(product,1)
lineItem : LineItem
<<create>>
6 : lineItem := new(product,1)
GRASP DESIGN
: User
<<IUserInterface>>
userInterface : GRASPPrototype
<<singleton>>
: ShoppingAssistant
sale : Sale
Add 1 unit of this product
1 : put product into cart()
2 : addProductShoppingCart(product)
3 : addProduct(product)
4 : addProduct(product,1)
lineItem : LineItem
<<create>>
5 : lineItem := new(product,1)

Figure 6.1: Comparison of *Add Product into Cart* actions on business logic

MVC pattern, this approach is not easy to take. The solution that has been used is putting all the views that have been shown into the main panel into a stack and, when the user asks to go back, the system gets the first view on the stack and shows it on the screen. Figure 5.23 shows a sequence of screens with a representation of the stack in each moment. The last step (select *Back* option) removes the top element of the stack and puts it on screen.

- **Reference to Shopping Cart:** in both prototypes, the reference to the *Shopping Cart* (class *Sale*) is kept in a specific object of the system. This reference is then used by the other objects, if necessary. In the GRASP prototype, this reference is kept inside the *ShoppingAssistant* controller, which centralizes the communication between user interface and business logic. On

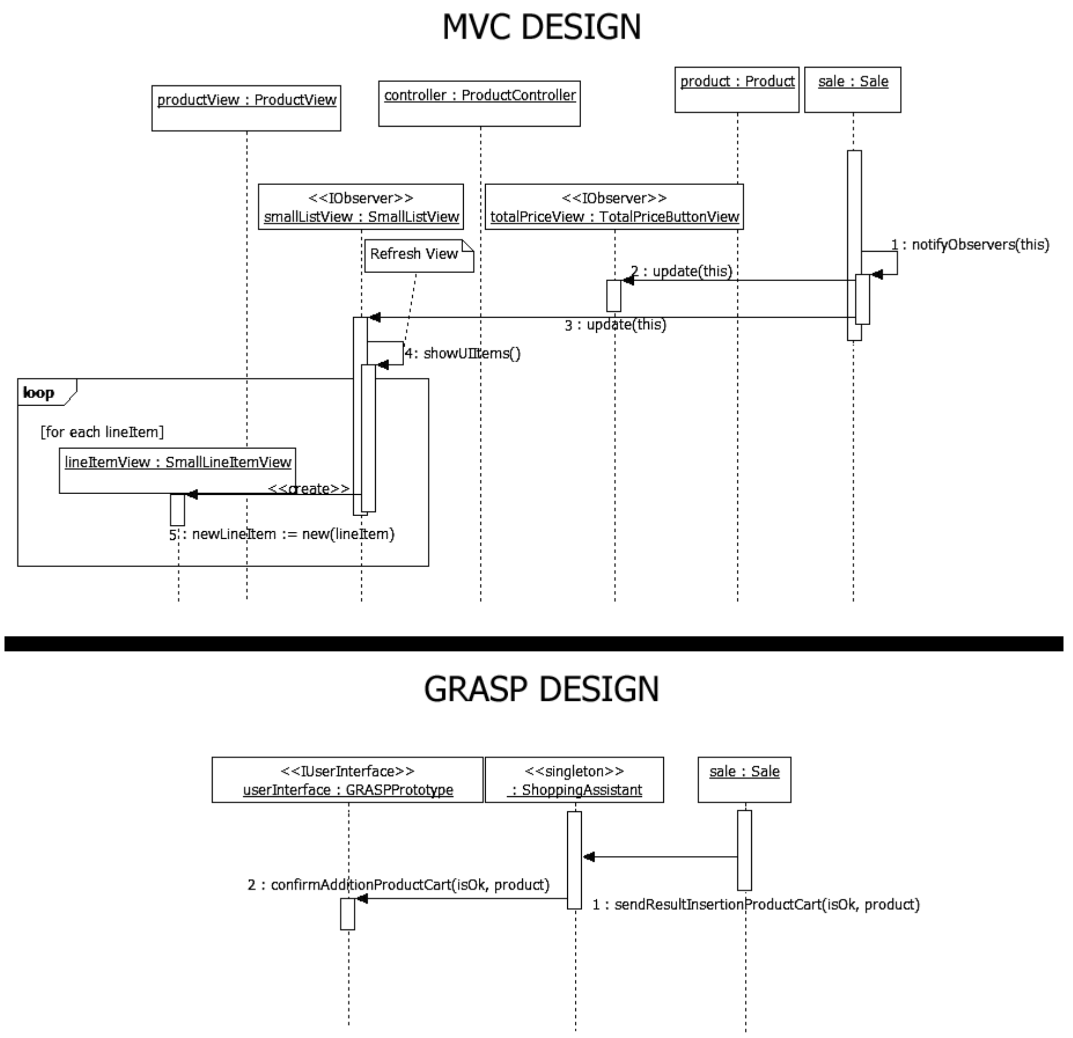
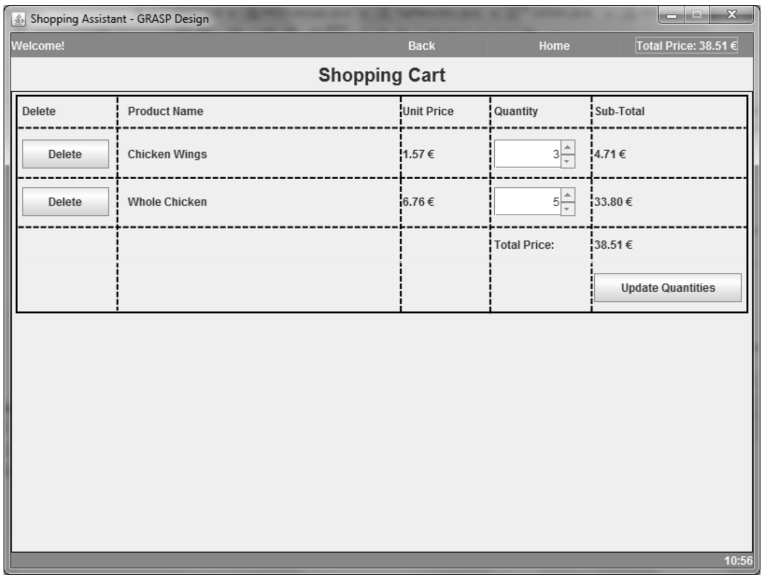

MVC DESIGN
productView : ProductView
controller : ProductController
product : Product
sale : Sale
<<IObserver>>
smallListView : SmallListView
<<IObserver>>
totalPriceView : TotalPriceButtonView
Refresh View
1 : notifyObservers(this)
2 : update(this)
3 : update(this)
4: showUIItems()
loop
[for each lineItem]
lineItemView : SmallLineItemView
<<create>>
5 : newLineItem := new(lineItem)
GRASP DESIGN
<<IUserInterface>>
userInterface : GRASPPrototype
<<singleton>>
: ShoppingAssistant
sale : Sale
2 : confirmAdditionProductCart(isOk, product)
1 : sendResultInsertionProductCart(isOk, product)

Figure 6.2: Comparison of *Add Product into Cart* actions

the MVC design, this reference is kept on the UIHelper class, which is the class that has been created to ease the development of the user interface classes and is accessible by any user interface object.

- **Shopping cart visualization:** in GRASP prototype, all items could have been inserted into a single table, with each field in a separated column as shown in Figure 6.3 . However, when we use a MVC design, each item on the shopping cart has to be shown in a separated view (Figure 6.4).

  As we can see, in the first figure, the limits of each column are the same for


Shopping Assistant - GRASP Design

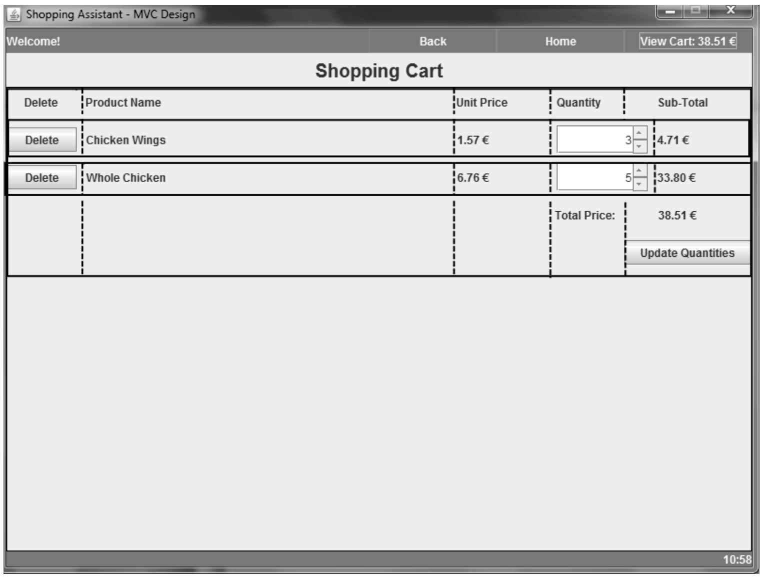
Welcome! Back Home Total Price: 38.51 €

Shopping Cart

Delete	Product Name	Unit Price	Quantity	Sub-Total
Delete	Chicken Wings	1.57 €	3	4.71 €
Delete	Whole Chicken	6.76 €	5	33.80 €
			Total Price:	38.51 €
				Update Quantities

10:56

Figure 6.3: Alignment in GRASP prototype


Shopping Assistant - MVC Design

Welcome! Back Home View Cart: 38.51 €

Shopping Cart

Delete	Product Name	Unit Price	Quantity	Sub-Total
Delete	Chicken Wings	1.57 €	3	4.71 €
Delete	Whole Chicken	6.76 €	5	33.80 €
			Total Price:	38.51 €
				Update Quantities

10:58

Figure 6.4: Alignment in MVC prototype

each row automatically. For example: if the product on the first row has a bigger name, it will push the remaining columns (on its right) *in all rows* to the right. But, in the MVC design, as each product is represented in a different object, in the same example, if the product on the first row has a bigger name, it will push the remaining columns (on its right) *in its own row* to the right, keeping the other rows without changing - this will cause a misalignment on the user interface.

The solution that has been found for this specific case is keeping the columns with a fixed width. In this way, if a product has a bigger name, its name will be split into two or more rows, keeping the columns width and the alignment intact.

6.2 Heuristics Evaluation

In order to figure out the impact of the usage of Model-View-Controller into the general usability of a system, we have conducted a software design evaluation based on usability heuristics, where the objective is analyzing the software designs that have been developed based on the usability heuristics that have been described on section 2.3.4.

6.2.1 UH1: Visibility of System Status

The visibility of the systems status can have its analysis divided into two parts. The first of them is related to the status of the system from a general point of view. The second is the update of the status of the system due to an action that has occurred in the system.

- **General status of the system:** in order to retrieve the status of the system in a general view, both prototypes worked in the same way. Taking a closer look into the designs we were not able to find any particularity that might be useful to this feature.

- **Result of an action:** this case is related to the updates that the user interface shall have in order to reflect the actions that the user has executed. The usage of the *Observer* pattern made this feature possible in the Model-View-Controller pattern and the GRASP prototype used a similar mechanism to make this update possible. However, the Model-View-Controller pattern seems to provide a more organized way of implementing those updates, as the updates are focused in user interface elements and not in the whole user interface at once. The GRASP design allowed us to make the same updates, but the way of communicating was from the business logic layer to the whole screen and not just a single user interface element, which is the case of the Model-View-Controller design.

6.2.2 UH2: Match between System and Real World

When developing a system, the match between the real world and the application that is being developed is useful for making the users comfortable on using the application. For this purpose, one of the GRASP principles, the controller, provides a good mechanism to help on mimicking the real world, as the controllers may simulate the actions that are related to entities in the real world and, by this, allowing the software to mimic the real world in an easier way.

6.2.3 UH3: User Control and Freedom

In order to meet the expectations from this heuristic, the system needs to have its behavior predictable and provide the user alternative paths for navigation in a system, being the most common: back and restart the process. In order to better discuss about them, those three actions are discussed below:

- **Back:** this feature has been developed in different ways in the prototypes. In the GRASP prototype, the back functionality is based on displaying the previous element that has been displayed. The previous element can be the "product's category" (when the actual screen shows the information of a product), "parent category" (when the actual screen shows the information of a category) or "element that was being displayed" (when the actual screen shows the shopping cart). However, in the MVC prototype, this approach was not possible, forcing us to implement this feature in another way, which is pushing all screens that are being visited into a stack and popping the previous element of the stack when the user calls the *back* functionality. Section 6.1.2 discusses this issue with more details. Comparing the two approaches, we believe that the one that has been used in the Model-View-Controller prototype provides a more flexible solution that eases changes in the navigation. It is important to mention that this solution could be implemented into the GRASP prototype as well. However, the restriction that came from the usage of Model-View-Controller raised an alternative design compared to the solution employed in the GRASP prototype.

- **Restart the process:** this feature is the possibility of going to the beginning of the navigation and has been implemented into both prototypes in a similar way.

6.2.4 UH4: Consistency and Standards

There are three different types of consistency in a software:

- **Visual consistency:** this type of consistency is based on how the user interface is designed and does not receive any impact from the software architecture.

- **Functional consistency:** for achieving this heuristic, the system needs to provide a consistent way of executing actions. There are three issues that need to be addressed:

  1. The same action needs to be called in the same way: on one point of view, the usage of Model-View-Controller can be helpful as if a specific

functionality of the system (*View Cart* in the prototype, for example) is called through a button, this button can be a view that is instantiated many times on the system. However, this improvement would still be possible in the GRASP prototype, by creating a method for creating this button or even by creating an extra class.

2. The system needs to provide the same way of inputting the same type of data into the system: in the prototype, there is only one place for data input, which is in the list of products inside the shopping cart, where there is a component *JSpinner* that is responsible for handling the input of data according to arbitrary rules. In this case, the rule applied is that the data is in Integer number format, between 0 and 100, with increments of 1. The usage of components with arbitrary data formats and rules can enforce this characteristic and, as in both implementations this could be done without any difference, we believe that this feature does not suffer any impact from the software architecture that is being used.
3. Validation of input data: given the finer granularity of the user interface in the MVC design and the existence of a specific entity (the controller) for the validation of the data input, it is expected that the MVC leads to a possible better treatment of data validation.

- **Evolutionary consistency:** this type of consistency is based on a top-level view of the consistency of a software, aiming that the applications inside a software family are consistent among themselves. Given the fact that the Model-View-Controller design is more flexible and organized, it is expected that it will ease the evolutionary consistency.

6.2.5 UH5: Error Prevention

The main objective of this heuristic is preventing the users from making mistakes in the data input. In this case, the best way of avoiding it is through the usage of user interface components that are customized to receive the data in an unambiguous way. This customization is done entirely in the user interface and, given the small scope of a single user interface component, it is independent of the software architecture. In this way, we will not consider this heuristic in the study.

6.2.6 UH6: Recognition Rather than Recall

As this heuristic is closely related to which information is given to the user in the system, this heuristic does not suffer any impact from the software architecture that has been chosen. Even in Model-View-Controller where the representation of information is made through views, which are supposed to represent only one model each one, the arrangement of views can be done in order to fulfill the requirements of the system

6.2.7 UH7: Flexibility and Efficiency of Use

Efficiency is the key value in this heuristic and can be achieved through many ways, being the most common:

- **Shortcuts:** The usage of shortcuts in a system aims on providing to an advanced user the possibility of doing his/her tasks in a more efficient way. Those shortcuts can be through the input device (for example keyboard) or through visual shortcuts (for example buttons for the most common functionalities). In the implemented application, keyboard shortcuts are not useful as it is supposed to be used without a keyboard, which is not present in the real world scenario. In the case of visual shortcuts, they can be implemented easily without a considerable impact on the software design as it is composed by buttons that will activate specific actions on the system.

- **Skip steps in a process:** in a process that is composed by a set of steps, some of them may be skippable. In this situation, the possibility of skipping the unnecessary steps is welcome. In the developed application, the process does not have skippable steps.

  However, one step further in the prototypes would be implementing a list of important offers. In this feature, the option of skipping steps in a process is welcome and, when implementing this feature, the flexibility that the *Back* functionality brought in the MVC prototype is useful, as the users would be able to go back from one of the important products to the listing of important offers by pressing the back button. The implementation of this feature in the GRASP prototype would be problematic as the definition of *previous screen* would need to be based on the situation and not on the actual implementation, which is the definition of a product.

- **Input buffering:** input buffering would be applicable in cases where the sequence of executing the internal logic, gathering the results and presenting the user interface is slow enough for encouraging the user to start inputting data before the screen is presented. In the application that has been developed this situation does not appear.

- **Multiple Views:** one possible feature for a next version of the prototype would be the possibility of having customized user interfaces for different types of users. This feature has been highlighted by some works in the literature review and, in this case, the usage of Model-View-Controller design would bring an advantage on implementing it when compared with the GRASP design. One reason for that is the nature of Model-View-Controller where different views may be implemented using the same controller for handling the user data input.

6.2.8 UH8: Aesthetic and Minimalist Design

This heuristic is related to the definition of which functionalities will be presented to the users and how they will be accessed and it depends entirely on the user interface design, being totally independent from the software architecture that is being used. Therefore, this heuristic does not suffer any impact from the software architecture.

6.2.9 UH9: Help Users Recognize, Diagnose and Recover from Errors

An error message in a software is a feedback from the system to the users which is related to some error that has occurred inside the system. Given the fact that the inner sections of the system have a way of communicating with (i.e. sending messages to) the user interface, how the error messages (and the polishment of their content) will be displayed to the user depends entirely on the user interface.

6.2.10 UH10: Help and Documentation

Given the characteristics of the system, it does not have a built-in help system. Moreover, documentation does not impact the software design. In this way, this heuristic is not considered in our study.

6.2.11 Summary

Table 6.1 provides a summary of the discussion about the usability heuristics.

Usability Heuristic		**Comments**
UH1:	Visibility of system status	
	1. General status of the system	It did not have any relationship with the software architecture of the prototypes.
	2. Result of an action	In the MVC prototype, the updates of the screen could be focused in a single UI element.
UH2:	Match between system and real world	The GRASP controller facilitates the mimic of the real world into the system.
UH3:	User control and freedom	
	1. Back	The arrangement of the user interface into views drove the implementation of this feature in the MVC design, leading to a more flexible solution when compared to the GRASP prototype.
	2. Restart the process	It has been implemented in similar ways in both prototypes.
UH4:	Consistency and standards	
	1. Visual consistency	Improvements could be implemented in both prototypes.
	2. Functional consistency	The validation of input data is an feature where the usage of MVC can bring advantages, as the validation is focused on the controller.
	3. Evolutionary consistency	The flexibility and organization of MVC may ease the evolutionary consistency of a system.
UH5:	Error prevention	It does not have any relationship with the software architecture of the system.
UH6:	Recognition rather than recall	It does not have any relationship with the software architecture of the system.
UH7:	Flexibility and efficiency of use	
	1. Shortcuts	It would not have any relationship with the software architecture of the prototypes.
	2. Skip steps in a process	The flexible *Back* functionality of the MVC prototype may ease the implementation of this feature in a future version of the prototype.
	3. Input buffering	This situation does not appear in the developed prototypes.
	4. Multiple views	The usage of MVC eases the development of this feature, as discussed in the literature review.
UH8:	Aesthetic and minimalist design	It does not have any relationship with the software architecture of the system.
UH9:	Help users recognize, diagnose and recover from errors	It does not have any relationship with the software architecture of the system.
UH10:	Help and documentation	Not considered in this study.

Table 6.1: Usability heuristics analysis summary

Chapter 7

Conclusions

Usability is one of the most important quality attributes in the software products that have been developed lately. This fact can be explained by the amount and variety of computational systems that are available to the population. Until a few years ago, usability was considered only a quality attribute whose depth of impact was restrict only to the user interface of a system [25, 5]. Recently, a higher attention has been given to the relationship between usability of a computational system and its software architecture and studies have been performed around this concept.

These studies are approaching the relationship between usability and software architecture in different ways: some studies analyze the relationship that usability has with other quality attributes as the studies from Harrison et al. [23], Seffah et al. [41] and Folmer et al. [16]; other studies provide architectural solutions implementing specific usability mechanisms as the ones from Folmer et al. [20], John et al. [25] and Carvajal et al. [10].

Even though MVC pattern is one of the main patterns used in computational systems, there is not much presented in the literature about its relation with usability. In most cases, the literature limits itself on a single comment about this relationship. One exception is the work presented by Junior et al. where an application has been redesigned with usability concerns in mind and the new design has been made based on the MVC architecture [26].

However, none of the analyzed studies provided a deep analysis focused on the mutual impact that the usage of MVC may have with the usability of a system. This study then, provides an analysis this relationship through a case study.

The comparison of the two developed prototypes has been divided into two parts: one of them is based on the usability heuristics proposed by Nielsen [39] and the other is based on a direct comparison of the software designs. The outcomes from the comparison are presented below.

7.1 Usability Heuristics Evaluation

Comparing the developed prototypes, the following advantages have been discovered:

- The updates on the user interface, when using MVC, are exactly focused on the view that is related to the model that has been updated. In the case of GRASP-based design, the whole user interface would receive the notification of the update from the model. This difference brings an easier implementation in MVC for presenting to the user the results of an action that has been taken.
- The *Back* functionality has been implemented in a more flexible way in the MVC-based prototype. This is caused by the fact that the restrictions on the user interface did not provide the situation necessary for implementing a simpler mechanism for this feature. However, the result is a more flexible operation for this functionality. This flexibility is useful for implementing *Skipping Steps* functionality for advanced users, as it becomes possible for coming back from the skipped step into the beginning of the process.
- The *Validation of Input Data* can be implemented in an easier way using MVC. This is based on the fact that the implementation for data validation is focused in the controller, and, this fact may help on providing a better implementation for data validation.
- The usage of MVC brings a more flexible and organized design, which can help on providing *Evolutionary Consistency* in a software application by making the updates easier.
- One advantage that has been already pointed in the literature survey is that the usage of MVC eases the implementation of *Multiple Views*. This is feature has not been implemented, but it is part of possible increments for the developed prototype, as it could provide different user interfaces for different types of customers in the prototype.

The usage of MVC, however, brought one drawback to the prototype that has been developed. In some situations, later changes on the user interface may become more complex when the application has been developed using Model-View-Controller. Due to the division of the screen into views and the tight relationship between view, controller and model, some changes on the user interface that would not cause changes in the business logic in other architectures may cause changes on the models that are related to the views that are being modified.

In general, the usage of MVC brought advantages to the usability of the developed prototypes when compared with the prototype with GRASP patterns. Some advantages are originated by the development of a more organized software design and not totally by the usage of MVC into the design, which is the case of the *Back* functionality. Even if these issues may be solved in any software architecture, these advantages still need to be considered.

7.2 Software Design Comparison

From the comparison of the software designs, it was detected that the main sources of difference between the MVC and the GRASP design in our case study come from the finer granularity in the MVC-based solution, leading to the following specific conclusions.

Message Passing Between Business Logic and User Interface

There are two main differences in the communication between the business logic and the user interface. They are:

1. One GRASP controller may be used to join N business logic classes and M user interface classes. This situation does not happen in MVC, because in MVC it is not possible to gather the actions that involves different business logic classes and belong to the same context (for example: use case) into a single container.

2. Larger amount of interactions between business logic and user interface. On the other hand, the implementations for those interactions will be simpler thus achieving higher scalability.

More Elaborated Designs in MVC

The granularity of the user interface leads to more elaborated designs when they use the Model-View-Controller pattern when compared to the designs that are based on the GRASP prototype. As a consequence, it may lead to creation of more flexible implementations in MVC-based applications.

Container for"Common Elements"

As the user interface has finer granularity in MVC, the elements that are used by many user interface objects may need a special mechanism for centralizing the provision of those elements.

Alignment of User Interface Objects

Due to the finer granularity of the user interface in MVC, a listing of information (for example: listing of products in a shopping cart) that could be grouped into one single object in the user interface of a GRASP design may be forced to be displayed as a list of distinct objects in the user interface of an MVC-based design. This issue should be further investigated in order to establish specific guidelines for user interface design in MVC.

7.3 Limitations and Future Work

The fact that both prototypes were developed by the same person may interfere in the validity of this study as the first implementation may influence the design of the second. This could be taken in consideration for a future work and in order to overcome this issue some setups are proposed: one possible approach would be to give the implementation to two different teams; another possibility would be to give the implementation to two different teams and after that, change the pair team-implementation (for example: firstly, Team1 implements PrototypeA and Team2 implements PrototypeB, after that, Team1 implements PrototypeB and Team2 implements PrototypeA. This second case would address possible differences between Team1 and Team2).

Furthermore, a future work should consider other case studies which involves different types of interactions with the user. In this way, the case studies can complement each other to provide a more refined answer to this topic.

As mentioned this study presents only a single analysis about the relationship between usability and MVC and to provide a better answer to this research topic, further studies need to be made. Also, the studies that are being conducted about a more comprehensive topic, the relationship between usability and software architecture, will bring more information to this specific topic as well.

Bibliography

[1] ISO 9241-11. *Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability.* International Organization for Standardization, 1998.

[2] Apple. Cocoa fundamentals guide. `http://developer.apple.com/mac/library/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html`, 2009.

[3] Apple. iphone os reference library. `http://developer.apple.com/IPhone/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html`, 2009.

[4] Len Bass and Bonnie E. John. Supporting usability through software architecture. *Computer*, 34(10):113–115, 2001.

[5] Len Bass and Bonnie E. John. Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software*, 66(3):187–197, 2003.

[6] Len Bass, Bonnie E. John, and J. Kates. Achieving usability through software architecture. `http://www.sei.cmu.edu/library/abstracts/reports/01tr005.cfm`, 2001.

[7] Randolph G. Bias. The pluralistic usability walkthrough: coordinated empathies. In *Usability inspection methods*, pages 63–76, 1994.

[8] Steve Burbeck. Applications programming in smalltalk-80: How to use model-view-controller (mvc). `http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html`, 1992.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture Volume 1.* John Wiley & Sons, first edition, 1996.

[10] Laura Carvajal. Usability-enabling guidelines: a design pattern and software plug-in solution. In *ESEC/FSE Doctoral Symposium '09: Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium*, pages 9–12, 2009.

[11] John D. Day and Hubert Zimmermann. The osi reference model. *Proceeding of the IEEE*, 71(12):1334–1340, 1983.

[12] Joseph S. Dumas and Janice C. Redish. *A Practical Guide to Usability Testing.* Intellect Books, revised edition, 1999.

[13] Robert Eckstein. Java se application design with mvc. `http://java.sun.com/developer/technicalArticles/javase/mvc/`, 2007.

[14] Xavier Ferré, Natalia Juristo, and Ana Maria Moreno. Obstacles for the integration of hci practices into software engineering development processes. In *Encyclopedia of HCI*, pages 422–442. IGI Global, 2006.

[15] Eelke Folmer and Jan Bosch. Architecturally sensitive usability patterns. In *Proceedings of the VikongPLoP 2003*, pages 27–47, 2003.

[16] Eelke Folmer and Jan Bosch. Architecting for usability: a survey. *Journal of Systems and Software*, 70(1–2):61–78, 2004.

[17] Eelke Folmer and Jan Bosch. Experiences with software architecture analysis of usability. *International Journal of Information Technology and Web Engineering*, 3(4):1–29, 2008.

[18] Eelke Folmer, Jilles van Gurp, and Jan Bosch. A framework for capturing the relationship between usability and software architecture. *Software Process: Improvement and Practice*, 8(2):67–87, 2004.

[19] Eelke Folmer, Jilles van Gurp, and Jan Bosch. Software architecture analysis of usability. In *Engineering Human Computer Interaction and Interactive Systems*, volume 3425/2005, pages 38–58. Springer Berlin / Heidelberg, 2005.

[20] Eelke Folmer, Martin van Welie, and Jan Bosch. Bridging patterns: An approach to bridge gaps between se and hci. *Information and Software Technology*, 48(2):69–89, 2006.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison–Wesley, first edition, 1995.

[22] S. Greenberg. Working through task-centered system design. In *The Handbook of Task Analysis for Human-Computer Interaction*, chapter 2, pages 49–65. Lawrence Erlbaum Associates, 2003.

[23] Neil B. Harrison and Paris Averigou. Leveraging architecture patterns to satisfy quality attributes. In *Software Architecture*, volume 4758/2007, pages 263–270. Springer Berlin / Heidelberg, 2007.

[24] Bonnie E. John and Len Bass. Usability and software architecture. *Behaviour & Information Technology*, 20(5):329–338, 2001.

[25] Bonnie E. John, Len Bass, Maria-Isabel Sanchez-Segura, and Rob J. Adams. Bringing usability concerns to the design of software architecture. In *Engineering Human Computer Interaction and Interactive Systems*, volume 3425/2005, pages 1–19. Springer Berlin / Heidelberg, 2005.

[26] N.C. Junior, A.J. Abackerli, and L.A. Consularo. User interface design for vcmms: an approach to increase fidelity and usability. In *Virtual Environments, Human-Computer Interfaces and Measurement Systems, Proceedings of 2006 IEEE International Conference on*, pages 130–135, 2006.

[27] N. Juristo, A.M. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *Software Engineering, IEEE Transactions on*, 33(11):744–758, 2007.

[28] Natalia Juristo, Ana M. Moreno, and Maria-Isable Sanchez. Clarifying the relationship between software architecture and usability. In *16th International Conference on Software Engineering and Knowledge Engineering*, pages 2–7, 2004.

[29] Glenn E. Krasner and Stephen T. Pope. A description of the model–view–controller user interface paradigm in the smalltalk–80 system. *Journal of Object Oriented Programming*, 1(3):26–60, 1988.

[30] Craig Larman. *APPLYING UML AND PATTERNS: An Introduction to Object–Oriented Analysis and Design and the Unified Process.* Pearson Education, second edition, 2002.

[31] Clayton Lewis, John Rieman, and Amended J. Blustein. *Task-Centered User Interface Design: A Practical Introduction.* N/A, 1993.

[32] Microsoft. Asp.net mvc. `http://www.asp.net/mvc/`, 2009.

[33] Microsoft. Model-view-controller. `http://msdn.microsoft.com/en-us/library/ms978748.aspx`, 2009.

[34] Sun Microsystems. Designing enterprise applications with the j2ee platform, second edition. `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html`, 2002.

[35] Scot F. Morse and Charles L. Anderson. Introducing application design and software engineering principles in introductory cs courses: model–view–controller java application framework. *Journal of Computing Sciences in Colleges*, 20(2):190–201, 2004.

[36] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 152–158, 1994.

[37] Jakob Nielsen. Usability inspection methods. In *CHI '94: Conference companion on Human factors in computing systems*, pages 413–414, 1994.

[38] Jakob Nielsen and Robert L. Mack. *Usability Inspection Methods*. John Wiley & Sons, first edition, 1994.

[39] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256, 1990.

[40] Marie Christine Roy, Olivier Dewit, and Benoit A. Aubert. The impact of interface usability on trust in web retailers. *Internet Research: Electronic Networking Applications and Policy*, 11(5):388–398, 2001.

[41] Ahmed Seffah, Taleb Mohamed, Halima Habieb-Mammar, and Alain Abran. Reconciling usability and interactive system architecture using patterns. *Journal of Systems and Software*, 81(11):1845–1852, 2008.

[42] Herbert A. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, 1962.

[43] Jacob Somervell, Shahtab Wahid, and D. Scott McCrickard. Usability heuristics for large screen information exhibits. In *Human-Computer Interaction – INTERACT'03*, pages 904–907, 2003.

[44] J. Tidwell. Common ground: A pattern language for human–computer interface design. `http://www.mit.edu/~jtidwell/common_ground.html`, 1999.

[45] M. Welie and H. Traetteberg. Interaction patterns in user interfaces. In *Proceedings of the Seventh Conference on Pattern Languages of Programming*, page N/A, 2000.

[46] M.V. Welie. Amsterdam collection of patterns. `http://www.welie.com/patterns/`, 1999.

[47] Jiajie Zhang, Todd R. Johnson, Vimla L. Patel, Danielle L. Paige, and Tate Kubose. Using usability heuristics to evaluate patient safety of medical devices. *Journal of Biomedical Informatics*, 36(1–2):23–30, 2003.

Appendices

Appendix A

Non-Functional Prototype: Usability Test Script

A.1 User Test Instruction

Imagine you are in the supermarket. You have selected several items and put them in your cart. On your cart, there is a small device called “Shopping Assistant” installed to help you with the shopping experience. It is a system that helps the customers on selecting and buying their products, providing a new concept of shopping.

A.2 Tasks

There are seven tasks that have been used in the usability test. They are:

Task 1: You want to buy beef fillet for the dinner tonight. You do not want to buy beef from Australia because there is now a cow disease over there. You do not remember exactly where it is but you know it must be in the meat section. So now you use the system to locate the beef fillet and check its country of origin.

Task 2: You want to buy a light bulb and you decide to try to locate by typing it, since you are not sure in which category it falls into. When you have found these items, you decide that you wish to buy light bulb Phillips 75W, and in order to get it you want to find its location on the map as well as on the shelf.

Task 3: You want to buy some milk on promotion. Sometimes the milk on promotion runs out of stock so you would like to know its availability without having to come to its location just to check.

Task 4: Find the promotion nearby your current location.

Task 5: Tonight you will go to your best friend’s housewarming party so you need to buy a gift for him. As he just moved in the new flat, you guess he would need some home appliances. With the budget of 50 Euros, you would like to know what thing you can buy in the supermarket.

Task 6: The system suggests you the tableware and the multi-blender as gift ideas. You are interested in the multi-blender and want to have more information about it. You decide to call the assistant to ask for more info.

Task 7: You get everything you want in and to finish the shopping session, you want to view the products in your cart and the total price you are going to pay.

A.3 Data Acquisition Form

Personal info			
Tasks	**Time taken (seconds)**	**Amount of errors**	**Number of clicks: Actual / Expected**
1. Search for beef filet			/ 5
2. Search for light bulb			/ 4
3. Find the milk promotion			/ 3
4. Find the promotion nearby			/ 1
5. Look for gift suggestions			/ 4
6. Call assistant			/ 1
7. View cart and total purchase			/ 1
Questionnaire			
Is the system easy to navigate? (1-very hard, 5 very easy)			
How easy is it to learn the system? (1-very difficult, 5-very easy)			
Do the icons/images adequately represent the tasks they do?			
Is there any option you do not understand/unsure its purpose?			
What is the best thing in the system?			
What is the worst thing in the system?			
Would you like to have this system in your supermarket?			
Would you change your supermarket just because another supermarket provides this service?			
Which software do you usually use?			
Other suggestions?			

Table A.1: Data acquisition form for usability test of the non-functional prototype

Appendix B

Non-Functional Prototype: Pilot Usability Test

Question	**Average**	**Std. Dev.**
Is the system easy to navigate?	4.33	0.57
How easy is it to learn the system?	4.33	0.57
Do the icons/images adequately represent the tasks they do?	4.33	0.57
Would you like to have this system in your supermarket?	4.66	0.57
Would you change your supermarket? (no=1, maybe=3, yes=5)	3.00	2.00

Table B.1: Pilot usability test - satisfaction questionnaire summary

Problems found on pilot test	
Item	**Description**
Top menu	It was difficult to the users to realize what the top menu provided, as there was no icons and no visible distinction on the buttons.
	The description of the buttons (the text that they contained) was not clear enough.
Nearby promotions	The users tried to find the "Nearby Promotions" inside the "Promotions" section of the system, not in the top menu.
Shelf location	The picture that was describing the shelf location of a product was leading to wrong or no interpretations.
Map	The users did not know that the "small" map (inside a description of a product) was clickable.
Big map	The big map was not clear enough as it had a lot of sections, making it difficult to see in a small screen.
Money restriction	The users did not see how to access this functionality.

Table B.2: Pilot test - problems discovered

Improvements after pilot test	
Item	**Description**
Top menu	Change the text of the "buttons" and addition of icons.
Small map	Addition of a text "Click to enlarge" below the small map.
Information of a product	Addition of "Out of stock" text on products that were not available.

Table B.3: Pilot test - problems solved

Appendix C

Non-Functional Prototype: First Usability Test

Question	**Average**	**Std. Dev.**
Is the system easy to navigate?	4.13	0.35
How easy is it to learn the system?	4.00	0.93
Do the icons/images adequately represent the tasks they do?	4.63	0.52
Would you like to have this system in your supermarket?	4.25	0.89
Would you change your supermarket? (no=1, maybe=3, yes=5)	2.75	1.98

Table C.1: First usability test - satisfaction questionnaire summary

Problems found on first test	
Item	**Description**
Selection of type of search (for products)	The pictures are not as clear as they should be.
Top menu	The “Back” button was not as clear as it should be.
Money restriction	The users did not see how to access this functionality.
Nearby promotions	The users tried to find the “Nearby Promotions” inside the “Promotions” section of the system, not in the top menu.
Shelf location	The picture that was describing the shelf location of a product was leading to wrong or no interpretations.
Big map	The big map was not clear enough as it had a lot of sections, making it difficult to see in a small screen.

Table C.2: First usability test - problems found

Appendix D

Non-Functional Prototype: Heuristics Evaluation

1. Visibility of System Status

- **Check if in each screen the user knows the aim of that screen:** yes, if we follow the tasks given along with the prototype each screen describes the aim. It is quite lucid.

- **Check if the feedback is appropriate for the aim of the screen:** the feedback provided is quite sufficient, however at some points improvements can be done. For example when we go for promotions nearby along with the products the specific location of the product can be given and also re-confirming where we are.

- **Check if the name of the labels is according to the information that should be filled:** the information is only filled in the search by category page and we believe it is a good option to have a keyboard for searching by name and there should be a mechanism for correcting the names typed by the user in the end product.

- **Check if the users always knows where he is and what is he doing:** yes, if we follow the tasks described by the developers of prototype we have enough feedback to understand where we are and what we are doing. The prototype keeps the user informed in each step.

2. Match Between System and the Real World

- **If the user has to fill any form, check if the labels have the correct explanation to refer to the information that should be filled:** the user does not have to fill any forms in this particular prototype, as it is not a web-based application. It is a touch screen application therefore there should be and there are few options where user fills in information.

- **Check if the system has a real aim:** the system has quite prudent aim considering the current business trends. It aims at improving the shopping experience of a customer in huge malls where most of the times the customer are lost finding the desired products.

3. User Control and Freedom

- **Check if the system guides the user in a appropriate way:** the system succeeds in guiding the user in most of the cases as mentioned before most of the time; however, sometimes the current location of the user should have been included. We expect it would be present in the subsequent versions of prototype or in the end product eventually. For example in promotions it can be included. Guiding the user is quite crucial for this system and we believe it has succeeded in doing so.

- **Check if the system always tell the user about where they are and what are they doing:** yes, if we follow the tasks described by the developers of prototype we have enough feedback to understand where we are and what we are doing. The prototype keeps the user informed in each step.

- **Check if the system has "back-way" in each screen:** there is a back button for each task, however it is not quite explicit for the non-computer users therefore, we suggest a button which says "BACK" rather than just "¡".

4. Consistency and Standards

- **Check if the system guides the user in a cognitive way:** we think that the system guides the user in a cognitive way. The system is quite easy to use and learn. It performs the task as expected that makes it a success from cognitive perspective. The user does not have flagrant things like a helping creature helping you during the shopping (like we had in Microsoft word) which makes it simple and non-irritating.

- **Check if all information is consistency:** the information and the pages remain consistent all across the prototype.

5. **Error Prevention**

   - **Check if all the text fields to fill are in the correct format:** there is only one field to fill, the search by name, and the format is completely correct.
   - **Check if the system could guide the user in a wrong way:** the answer is no in general. One exception is in Product Search, the icon of search by category is not very clear because one can thing that instead of a simple icon it is composed of several buttons.

6. **Recognition Rather than Recall**

   - **Check if the system force the user to remember too many data in order to complete an operation:** in none of the steps to complete the tasks we felt the need to remember too many data. We didn't find a task were previous data should be introduced again.

7. **Flexibility and Efficiency of Use**

   - **Check if the system has a keyboard access, tool tip-texts, short-cuts:** because this system interacts with the user through a touch screen we consider that this doesn't apply. To help the users, a very well selected set of icons is used.

8. **Aesthetic and Minimalist Design**

   - **Check if the system has any irrelevant information:** this point depends on each of the user and his needs. Sometimes a user is not looking for a promotion or he doesn't want to know any promotions in the nearby. So there is a lot of information that maybe is not relevant for some users. But because the supermarket wants to promote new products or services all the time, this customer will be the first interested in include all of this information. So we think that because of the commercial nature of this system this irrelevant information is completely fine.

Appendix E

Non-Functional Prototype: Second Usability Test

Task	Time Taken	Amount of Errors	Amount of Clicks
Search for beef filet	14.00 ± 3.94	0.00 ± 0.00	5.00 ± 0.00
Search for light bulb	18.16 ± 9.15	0.00 ± 0.00	5.00 ± 0.00
Find the milk promotion	20.00 ± 15.88	0.83 ± 1.60	5.83 ± 3.60
Find the nearby promotions	6.00 ± 9.44	0.00 ± 0.00	1.00 ± 0.00
Look for gift suggestions	16.33 ± 5.43	0.00 ± 0.00	5.00 ± 0.00
Call assistant	2.83 ± 1.83	0.17 ± 0.41	1.00 ± 0.00
View cart and total purchase	3.50 ± 3.89	0.17 ± 0.41	1.00 ± 0.00

Table E.1: Second usability test - results summary

Question	Average
Is the system easy to navigate?	4.33 ± 0.51
How easy is it to learn the system?	4.50 ± 0.55
Do the icons/images adequately represent the tasks they do?	4.83 ± 0.41
Would you like to have this system in your supermarket?	5.00 ± 0.00
Would you change your supermarket?	3.67 ± 0.52

Table E.2: Second usability test - satisfaction questionnaire summary

Item	**Description**	**Occurence**
Big Map	Simple, easy to understand but lack of appeal	3 / 6
Big Map	Users did not realize it was clickable	1 / 6
Budget Limit	Often overlooked, not attracting user's attention	2 / 6
Product Search	One product can be listed in several categories	2 / 6

Table E.3: Second usability test - problems found