I

INFORMATION SOCIETIES TECHNOLOGY (IST) PROGRAMME

STATUS

"Software Architecture for Usability"

WORKPACKAGE: 3 Study of the usability/software architecture relationship

DELIVERABLE 3.4. Techniques, patterns and styles for architecture-level usability improvement

Version: 1.0.

Submission Date: 29/4/03

Authors: Natalia Juristo, Ana M. Moreno, Maribel Sanchez

Partners: UPM

Stage:	Confidentiality:
[ ] Draft	[ ] Public - for public use
[ ] To be reviewed by WP participants	[ ] IST – for IST programme participants
[ ] Pending of approval by next consortium meeting	[X] Restricted – for STATUS consortium and PO
[X] Final / Released to CEC	

DOCUMENT CONTROL

Registration of Changes

Date	Version	Author of Changes	Comments
22/10/02	0.1	Ana M. Moreno, Natalia Juristo	Initial structure, first draft of section 2 and 3
27/10/02	0.2	Ana M. Moreno, Natalia Juristo	First draft of section 4.
5/12/02	0.3	Ana M. Moreno, Natalia Juristo	Second draft of section 4.
2/1/03	0.4	Ana M. Moreno, Natalia Juristo	Adjustments to section 4
21/1/03	0.5	Maribel Sanchez	Inclusion of examples of patterns in section 4
4/2/03	0.6	Ana M. Moreno, Natalia Juristo	Adjustment of deliverable with respect to the concept of architectural usability pattern
10/2/03	0.7	Maribel Sánchez, Camilo Calderón	Inclusion of examples of patterns and architectural usability patterns in section 4 Inclusion of Annex E
28/04/03	0.8	Ana M. Moreno , Natalia Juristo	Last reveiw
29/4/03	1.0	Ana M. Moreno , Natalia Juristo, Maribel Sanchez	Changes on command aggregation

List of STATUS Related Documents

Document Name	Version
Technical Annex	1.0
Deliverable D.2.	1.0
Deliverable D.3.5.	1.0

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Purpose

This document forms deliverable D.3.4. *Techniques, patterns and styles for architecture-level usability improvement* corresponding to Task 3.4..

As specified in the Technical Annex, the work to be done in WP 2 *Usability Attributes Affected by Software Architecture* and WP 3 *Relationship between software usability and software architecture* is oriented to providing a solution to the relationship between software architecture and software usability. The main contribution of Task 3.4. is to provide developers with a set of proposals to improve the usability of the applications they build. The solutions we propose in Task 3.4. should be as general as possible and be able to be instantiated for different kinds of specific applications. One such instantiation will be carried out in WP 4 for applications in the e-commerce domain.

For this reason, the results described in this deliverable focus on providing a set of general design solutions, which, if accounted for during system architecture definition, can improve the usability of the software constructed. These design solutions have been generated for every usability pattern defined in D2, thereby extending the description of usability pattern employed in WP 2 with new attributes related with software architecture aspects.

1.2 Document Structure

This document is structured as follows. Section 1 presents an introduction that sets out the objectives and purposes of this deliverable. Section 2 focus on the concept of usability pattern, defined in WP 2, and how such concept needs to be extended in order to incorporate design solutions to the usability mechanisms represented by such patterns. This section also studies the differences between the usability pattern concept used in STATUS and the usability pattern concept that can be found in literature. As part of the related work studied, this section also details the relationship between the STATUS work and the SEI work about usability and software architecture. It finishes presenting the detailed list of usability patterns that will be used from now in STATUS. This definitive list has suffered some adjustments from the one presented in WP2, those adjustments are exhaustively detailed.

Section 3 presents the inductive process that has been followed to provide design solutions for each of the usability patterns outputted from section 2. Section 4 lists a catalogue of the whole description of usability patterns which includes a full set of aspects that cover the information needed by developers in order to use such patterns. Section 5 sets out what implications the usability patterns have for application analysis. Finally, Section 6 includes the conclusions drawn from Task 3.4.

Much information used in D.3.4. has been included in annexes to make easier the reading of this deliverable. Annex A contains a detailed description of each of the usability patterns. Annex B contains the original requirements specifications for the cases used to generate the design solutions for the usability patterns. Annex C covers the first iteration in the process of induction to abstract the design solutions, which corresponds to the Restaurant Management system case. Similarly, Annex D contains the second iteration of the process of induction, which corresponds to the Amusement Park Control system case. Annex E contains the comprehensive development of an Intranet for Advertising Company Maintenance, which has served to validate the design solutions proposed for the usability patterns in a real project. Finally, Annex F sets out the catalogue of the usability patterns.

2 USABILITY PATTERNS: THE STATUS APPROACH TO IMPROVE USABILITY FROM ARCHITECTURE

2.1 The STATUS Concept of Usability Pattern

The formal concept of pattern comes from the definition given by Alexander in the context of construction of buildings and cities [Alexander, 77]. "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

In software development, the concept of pattern is described as a set of principles and idiomatic solutions that guide developers in the creation of software [Larman, 98]. Accordingly, a software pattern focuses on the pragmatic value of using the pattern as a vehicle for presenting and remembering solutions to a problem by providing useful software engineering principles to the developer. Thus, the main advantage of patterns is the experience in reusing instead of rediscovering potential solutions to a problem.

The most commonly used pattern in software development is the design pattern, and it is particularly used in the object-oriented field. In this context, a design pattern provides a solution to a concrete problem by describing classes and objects that work together to solve such problem [Gamma, 89]. Note that these patterns show a solution to a problem obtained by its use in different applications, but it could not in any case be seen as a unique solution.

At this point in STATUS, the usability patterns proposed in D.2. are described as mechanisms to be applied to the design of the architecture of a system in order to address a particular usability property. We need to complete this definition with information about how such mechanisms will be reflected in the system architecture, that is, what effect the use of some of these usability mechanisms will have on a system's architectural components. In other words, we need to provide the design solution to the problem specified by the usability pattern. Design solutions will be a description of the design components and their intercommunication to provide a solution to a specific mechanism to be applied to the design of a system in order to address a particular usability property.

Like in design patterns, the design solutions provide for usability patterns will not necessarily be the only solution. Design solutions for usability patterns are not solutions to specific problems, but should be able to be applied to solve a number of different problems in a number of different systems in accordance with the principle of software reuse.

Thus, the final objective of this deliverable is to provide a set of architectural recommendations to improve the usability of software systems. These recommendations will be summarised in an usability pattern catalogue.

2.2 Related Work on Usability Patterns

The concept of usability pattern has already been used in the literature. Generally, this concept can be defined as "a description of solutions that improve usability attributes" [Perzel, 99]. The usability aspects dealt with by these patterns refer basically to user interfaces, and they are also known as user interface patterns [Casaday, 97] or interaction design patterns [Tidwell, 98]. As indicated by authors such as Welie and Troetteberg [Welie, 00], although there are several pattern collections, an accepted set of such patterns has not yet emerged. There appears to be a lack of consensus about the format and focus of user interface patterns.

Examples of some of these interface patterns are:

- Feedback
- Wizard

- Provide the user with all information needed in the same window
- Mark required fields when filling a form
- You are here
- Grid layout.

The differences between the usability patterns proposed by the STATUS project and the classic usability or interface patterns existing in the literature are basically related to two aspects:

- **Differences in the list of patterns**. Among the usability patterns identified in the literature, we find some that are oriented to improving the interface, like for example "you are here", and others oriented to improving interaction, like "wizard". The patterns proposed by STATUS are mainly focused on improving the interaction between the user and the system without considering interface issues. Therefore, some of the patterns traditionally included as usability patterns in the literature are not considered in the STATUS catalogue of usability patterns. In STATUS we consider patterns directly related to the interaction between the user and the system, and not taken into account in the classic literature on user interface patterns, such as shortcuts, aggregation of commands or the user interface patterns of provision of actions for multiple objects.
- **Differences in the solutions provided for patterns**. Although there are patterns that are common to the two approaches (traditional and STATUS), for these patterns the difference lies in the solution provided in each one. The classic usability patterns are mainly implemented during the interface design phase and generally affect low-level components like pseudo-code (where to place the different icons, how to put together the screen information, etc). On the other hand, the solutions proposed by STATUS for the identified patterns will affect the system architecture, trying to evaluate and consider usability aspects in the early stages of the development process. For example, the solution proposed by [Welie, 00] for the feedback pattern is "provide a valid indication of progress. Progress is typically the remaining time for completing, the number of units processed or the percentage of work done. The progress can be shown using a widget such a progress bar. The progress bar must have a label stating the relative progress or the unit in which is measured". This solution will not be taken into account until the detailed design phase through the pseudo-code As will be seen later, the solution proposed by STATUS (through the respective architectural solutions) will be based on the establishment of how to provide feedback to the user at the architectural level (modules, interaction among modules, etc).

2.3 Usability Patterns versus Usability Scenarios

As far as the consortium has been able to ascertain from the literature, the only work with similar goal than STATUS project is the research by Bass, John and Kates at SEI [Bass, 01]. Their aim was to identify the relationship between usability and software architecture through the definition of a set of 26 scenarios: "A scenario describes an interaction that some stakeholder (e.g. user, developer, system administrator) has with the system under consideration from a usability viewpoint". The above-mentioned scenarios were identified by the SEI: through literature surveys, from the personal experience of the investigators and by consulting colleagues. The complete list of the scenarios is shown in the Table 2.1. As we will see later, although there are some differences, some of these scenarios can be considered equivalent to some of the properties and usability patterns taken into account in STATUS.

Account for human needs and capability when interacting	Maintain device independence
Keep coherence through multiple views	Allow searching by different criteria
Define upgrades similar to previous ones	Make views accessible
Support international use	Provide reasonable set of views
Predict task duration	Cancel
Verify resources before beginning an operation	Use applications concurrently
Present system state	Allow quick switch back and forth between different tasks
Check for errors	Aggregate commands
Undo	Aggregate data
Minimize user recovery work due to systems errors	Provide test points for evaluation
Provide alternative secure mechanisms	Reuse information
Provide good Help	Design easily modifiable interfaces
Novice interfaces for user in unfamiliar contexts	Quick navigation into a view

Table 2.1. Usability scenarios proposed by Bass et al [Bass, 01]

The SEI researchers use a bottom-up specification to specify the benefits that a software system can have in terms of usability as a result of the identified scenarios. These benefits have been classified in Table 2.2.

SEI Benefits	STATUS Attributes
Increases individual user effectiveness	
Expedites routine performance Accelerates error-free portion of routine performance Reduces the impact of routine user errors (slips)	Efficiency, Reliability
Improves non-routine performance Supports problem-solving Facilitates learning	Learnability
Reduces the impact of user errors caused by lack of knowledge Prevents mistakes Accommodates mistakes	Efficiency
Reduces the impact of system errors Prevents system errors Tolerates system errors	Reliability
Increases user confidence and comfort	Satisfaction

Table 2.2. Relationship between SEI usability benefits and STATUS usability attributes

Table 2.2 shows how SEI usability benefits generally refer to the usability attributes considered in STATUS and detailed in D2. Note that there is a main difference with regard to system errors. The SEI researchers consider the prevention of errors as one of the usability benefits. However, as discussed in D2, this aspect is not considered as a feature of usability attributes in the usability field. Following the usability works, the usability attribute related to error management, reliability, refers to user error and not system error prevention and recovery. This is the approach taken by STATUS.

Both pieces of research agree on the idea of relating the different aspects of usability to architecture through architectural diagrams. These diagrams show how scenario (SEI approach) or usability pattern (STATUS approach) can be represented at an architectural level. As in the STATUS approach, the objective of the SEI is to establish a relationship between the final software system and software architecture. The main differences between the two works come from the approach taken for the research.

The SEI has taken a bottom-up approach from the informal identified scenarios, while STATUS has taken a top-down approach from usability attributes (identified in the literature), through usability parameters, to finally identify the usability patterns. Accordingly, the usability patterns are the final links in the chain and give examples of how to achieve some usability requirements. Besides, the users

of STATUS results have a procedure for developing new usability patterns for applications to which the results are applied. This process will be applied in practice in WP 4 *Proposal of Architecture for Usability in E-commerce*, where the results of WP 3 will be adapted to the e-commerce environment. Finally, the architectural solutions are not justified in the SEI research, whereas the architectural patterns provided by STATUS have been derived through an inductive process applied in several case studies.

In sum, a general difference between the approach taken by the SEI and ours is that STATUS project provides a whole approach for designing for usability, including usability assessment techniques and usability improvement techniques on design models (D.3.5. *Usability-centric software architecture design method* will detail such design process).

2.4 Adjustments to WP 2 usability patterns

Table 2.3 below shows the usability patterns that we identified in D.2. As we mentioned in that document, the list shown in Table 2.3 is preliminary since it is the result of a first approximation. We were sure at that time that when working on WP 3 looking for design solutions for usability patterns we would refine this preliminary list arriving to a consolidated one.

Progress indication	Macros
Alerts	User profile
Status indication	User modes
History logging	Shortcuts
Undo	Context-sensitive help
Form or field validation	Wizard
Model/View/Controller separation	Selection indication
Emulation	Cancel
Workflow model	Multi-tasking
Actions for multiple objects	

Table 2.3. Preliminary List of Usability Patterns from D.2.

Analysing the preliminary set of patterns has led to modify the D.2 patterns in several ways:

1. Some usability patterns have been added as a result of further research conducted during Task 3.4.
2. Some patterns have been removed because they do not match the consolidated concept of usability pattern in WP 3.
3. Some usability patterns have been redefined with the aim of providing a more precise description and/or a more meaningful name.
4. A new usability pattern has been added derived from the identified scenarios [Bass, 00], once our list has been compared with the SEI proposal.

Below, we describe the above modifications.

The new usability patterns incorporated from further research are:

Different Languages

This pattern emerges to address the property of internationalisation identified in WP2, Different Languages. In D.2. this property had been identified but it did not had a pattern that deals with it. Internationalisation means the capability of the software to interact with users in different languages, so this is the concept we have used for the pattern.

Different Access Methods

In the same way than internationalisation, the property of accessibility (multichannel, disabilities) identified in D.2. did not had a pattern to address it. Access method means the capability of the software of being accessed from different types of physical devices. So, this attribute will make the system easier to access not only from the desktop or laptop, but also using devices like WAP, Web, and interactive TV, for example, as well as other devices used by visually impaired or disabled people.

Standard help

This and the tour pattern emerge to add to the different help types that contribute to addressing the property of guidance identified in D.2. This pattern deals with the standard help. The system must provide users with enough information and task help for all the activities they need to do with the system.

Tour

This pattern addresses a different kind of help than the standard help. In it, the system must provide users with specific information that shows exactly how to do a particular task.

The patterns we have removed due to the better concretion and definition of the usability pattern concept are:

Progress Indication

A progress indicator is described in D.2. as "a mechanism that can be used to indicate how much of the current task has been completed and how long it will take to finish". This description is a particular case of another usability pattern defined in D.2., status indication, We defined in D.2. status indication as "The user should be provided with information pertaining to the current state of the system".

Model/View/Controller Separation

As described in D.2, this pattern represents a specific design solution that can be used to implement different usability patterns, like, for example, patterns related to the provision of different information views, the pattern we called provision of views.

Emulation

This pattern is defined in D.2 as "A system can be made to emulate the appearance and/or behaviour of a different system". This is a very specific pattern for some applications and it is not common to general-purpose software systems. Additionally, it has not been proposed as a pattern or heuristic in usability literature.

User modes

This pattern, as specified in D.2., enables the system to provide different modes for different feature sets required by different types of users, for example, simple or advanced modes. Thus, this pattern is a particular case of the Workflow Model, which enables different users to be provided with only the tools or actions that they need in order to perform their specific tasks.

Selection indication

The user and the system will often have one object or a set of objects that have a special status. They are the objects that will be acted upon the next time that a command is issued. These objects should be indicated using some sort of highlighting. This usability aspect is not directly reflected in the system architecture, but in the system low-level design.

The usability patterns we have redefined are:

Preview

This pattern is defined in D.2 as "A user may wish to see what the results of an action (possibly a resource consuming one) will be before executing the command." It can be considered as a particular case of a more general pattern called Provision of Views, which we have now defined as "Allow users to have different alternatives to see the data they are working with at any time".

Macros

This pattern has been renamed *Command Aggregation* according to the respective scenario considered by the SEI. The new pattern can be defined as "The system should provide the capability to allow the users to do different actions through just one command". The generation of a macro is an example of this pattern for some particular applications, whereas this pattern could be implemented by scripts, batch programs, etc., for other applications.

Finally, the pattern adopted from SEI work is the only aspect not covered by our usability patterns but addressed by SEI:

Reuse information

This pattern was not originally considered in the D.2 usability patterns. This pattern enables the user to move data from one part of a system to another. So, users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system.

Table 2.4 shows the final list of usability patterns used in STATUS. The first column shows the usability patterns already defined in D.2. The second column shows the final architectural usability pattern used in D.3.4., showing the new patterns in boldface and the redefined patterns in italics. The last two columns, which will be detailed below, show the relationship between the architectural usability patterns used in STATUS and the SEI scenarios. The relationships we have found are:

- Content. Achieving a particular STATUS usability pattern implies achieving a particular SEI scenario. For example, properly implementing the Provision of Views patterns implies the SEI Make views accessible.
- Instantiation. A STATUS usability pattern is a special case of a SEI scenario. For example, the STATUS Standard Help is a case of the SEI Help.
- Similarity. A STATUS pattern and a SEI scenario are considered similarly in both approaches, for example, Cancel.
- Generality. A SEI scenario is a special case of a STATUS usability pattern. For example, the SEI Novice Interfaces for Users in Unfamiliar Contexts is a special case of provide the STATUS pattern Workflow model.

As we can see from Table 2.4., some SEI scenarios have not been considered in WP3. The specific reason for every scenario is explained below:

- Account for Human Needs and Capabilities when Interacting, Keep Coherence through Multiple Views, Define Upgrades Similar to Previous Ones, Provide Test Points for Evaluation and Design Easily Modifiable Interfaces are the results of specific actions to be taken during the development process and do not fit in with the definition of usability pattern considered in STATUS. Considerations of this nature have been taken into account in STATUS WP5, which deals with the general development process and not only with the architectural design phase like WP3.
- Minimize User Recovery Work due to System Errors refer to errors made by the software system and not by the users. As explained earlier, system errors are not considered by classical usability attributes, which is why this feature has not been considered in STATUS.
- Allow Searching by Different Criteria is a functional requirement that is specific to particular applications, and is not, therefore, really a usability architectural pattern as it is considered in this project.
- Provide Alternative Secure Mechanisms is a security requirement and not a usability architectural pattern as defined in STATUS.

D.2. USABILITY PATTERNS	D.3.4. USABILITY PATTERNS	RELATION	SEI SCENARIOS
	Different Languages	similarity	Support international use
	Different Access Methods	generality	Maintain device independence
Alert	Alert	generality	Verify resources before beginning an operation
Status Indication Progress Indication	Status Indication	similarity generality	Present system state Predicting task duration
Shortcuts (key and tasks)	Shortcuts (key and tasks)		
Form/Field Validation	Form/Field Validation	similarity	Checking for errors
Undo	Undo	similarity	Undo
Context-Sensitive Help	Context-Sensitive Help	instantiation	Provide good Help
Wizard	Wizard	instantiation	Provide good Help
	Standard help	instantiation	Provide good Help
	Tour	instantiation	Provide good Help
Workflow Model	Workflow Model	generality	Novice interfaces for user on unfamiliar contexts
History Logging	History Logging		
Preview	*Provision of Views*	content similarity content	Make views accessible Provide reasonable set of views Quick navigation into a view
User Profile	User Profile		
Cancel	Cancel	similarity	Cancel
Multi-Tasking	Multi-Tasking	content content	Use applications concurrently Allow to quick switch back and forth between different tasks
Macros	*Commands Aggregation*	similarity	Aggregate commands
Action for Multiple Objects	Action for Multiple Objects	similarity	Aggregate data
	Reuse Information	similarity	Reusing information
			Provide test points for evaluation
			Design easily modifiable interfaces
			Allow searching by different criteria
			Minimize user recovery work due to system errors
			Provide alternative secure mechanism
			Account human needs and capability when interacting
			Keep coherence through multiple views
			Define upgrades similar to previous ones

Table 2.4. Relationship between STATUS architectural usability patterns and SEI scenarios

Table 2.5 shows the final relationship between the usability properties defined in D.2. and the consolidated list of architectural usability patterns that we will use henceforth in STATUS.

Usability Property	**Usability Patterns**
Natural Mapping	
Consistency (Functional, Interface, Evolutionary)	
Accessibility (Internationalisation)	Different languages
Consistency Accessibility (Multichannel, Disabilities)	Different access methods
Feedback	Alert
Error Management Feedback	Status indication
Explicit User Control Adaptability (User Expertise)	Shortcuts (key and tasks)
Error Management (Error Prevention)	Form/field validation
Error Management (Error Correction)	Undo
Guidance Error Management	Context-sensitive help
Guidance Error Management	Wizard
Guidance Error Management	Standard help
Guidance Error Management	Tour
Minimise Cognitive Load Adaptability Error Management (Error Prevention)	Workflow model
Error Management (Error Correction)	History logging
Guidance Error Management (Error Prevention)	Provision of views
Adaptability (User Preferences)	User profile
Error Management Explicit User Control	Cancel
Explicit User Control	Multi-tasking
Minimise Cognitive Load Error Management (Error Prevention)	Commands aggregation
Explicit User Control	Action for multiple objects
Minimise Cognitive Load Error Management (Error Prevention)	Reuse information

Table 2.5. Relationship between usability properties and architectural usability patterns

It should be noted that the properties of Natural Mapping and Consistency cannot be arranged around specific usability patterns. The reason is that these properties require the performance of different tasks and activities throughout the entire development process rather than the application of particular solutions at the architectural level. For example, the provision of natural mapping between the user tasks and the tasks to be implemented in the system calls for software requirements to be elicited during the analysis process bearing in mind this objective and they must be designed according to these requirements. The same goes for consistency, which involves different activities throughout the lengthy development process of the original system or new versions.

Annex A contains a detailed description of the usability patterns upon which this WP is based. That is, Annex A is a second version of section 4 in D2, but refined as explained in this section. Design solutions for these patterns are presented in the next section.

3 DESIGN SOLUTIONS FOR USABILITY PATTERNS

Now we are going to describe the procedure followed to identify the design solution for each architectural usability pattern. The procedure can be divided into two phases:

- **PHASE 1. Generating Architectural Design Solutions**

  Application of induction to abstract the architectural solutions from particular designs for some applications developed by both researchers and practitioners. For this purpose, we took the following steps:

  STEP 1. We asked designers to build the design models (represented by class diagrams and interaction diagrams) for two systems *not including the usability mechanisms* specified by the usability patterns.

  STEP 2. Once the designs without usability mechanisms were complete, we asked designers to *modify their earlier designs* for each usability pattern to include the functionality corresponding to the usability mechanisms under consideration.

  STEP 3. For each usability mechanism, we *abstracted the respective architectural design solution* by generalising the modifications made by the different developers to their designs.

  This process was carried out on two different applications developed by different researchers and practitioners: Restaurant orders and tables management; Ride control and maintenance at an amusement park. For ease of reading, we are going to illustrate in this section the process only for one usability pattern, showing just one design example with and without usability mechanisms for the two applications. The process followed for the rest of usability patterns has been placed in Annexes C and D.

- **PHASE 2. Validating Architectural Design Solutions in a Real Development**

  Although the inductive process applied to a couple of applications developed by several developers might well have been sufficient to be sure of the design solutions, we wanted to validate these architectural solutions from the more comprehensive viewpoint of a full development. That is, in Phase 1, the researchers and practitioners knew they were participating in a research project and what the aim of this project was, they were familiar with the applications (either because they had developed them or had used them for training, etc.). Therefore, when asked to build design models with and without usability mechanisms, the developers exclusively addressed the parts of the system affected by the inclusion of the mechanisms in question. In other words, all the work carried out in Phase 1 was done as an exercise, not in the environment of a full and real development. By contrast, this validation phase was run within a development project for an MSc thesis. The development team was building and about to implement a system, having completed the design phase without considering usability patterns. The requirements were then changed, stating that the system should incorporate all the usability mechanisms set out by the usability patterns. The team then had to modify the design to consider these mechanisms. Again, we were able to observe what changes were made to the design because of this addition, and we were able to check whether the changes were equivalent to the instances of the solutions found in Phase 1. Again, for ease of reading, section 3.2 gives only one example of a use case with its respective associated interaction diagram. In Annex E the full development can be seen.

3.1 Phase 1: Generating Design Solutions for Usability Patters

3.1.1 First Iteration: Finding Design Solutions for Architectural Usability Patterns

Below, we show how Phase 1 was conducted for the Restaurant orders and tables management application (the requirements for this application are listed in Annex B). This application was taken as the first step in the process of induction for identifying the architectural usability patterns and, therefore, has been referred to as the first iteration. Each of the steps identified above has been taken for each usability pattern:

- STEP 1. Design models (class diagrams and interaction diagrams) without the usability mechanisms specified by the usability pattern.
- STEP 2. Design models (class diagrams and interaction diagrams) with the usability mechanism specified by the usability pattern.
- STEP 3. Abstraction of the design solution for the usability pattern.

As explained above, for clarity's sake, this section will only show the first iteration for the pattern Different Languages, whereas Annex C includes the first iteration for each one of the usability patterns from Table 2.4. This will illustrate the first iteration concerning how the design solution for the usability pattern is abstracted.

STEP 1. Design solution without the Different Languages usability pattern

Figure 3.1 shows the class diagram of the application where the pattern Different Languages has not been considered. In this case, no interaction diagram is shown as this functionality was not considered originally.

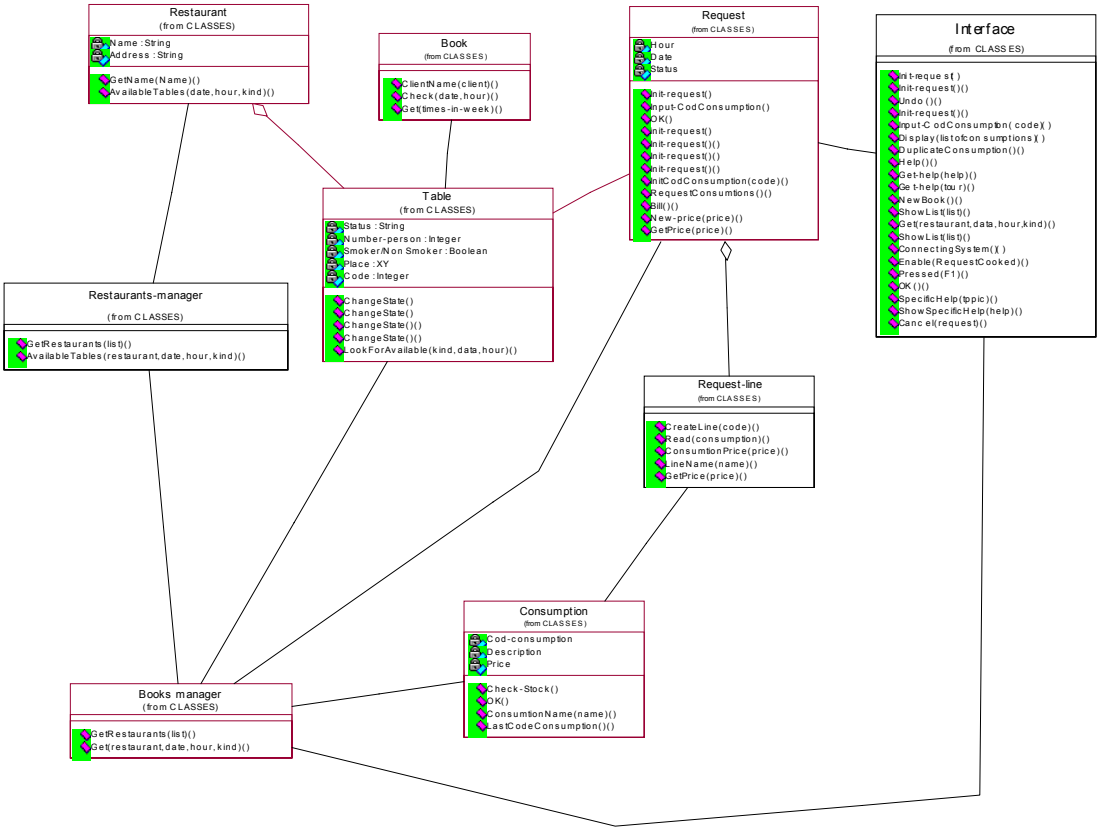


Figure 3.1 Class Diagram for the first application without the Different Languages mechanism

STEP 2. Design solution with Different Languages usability pattern

With the aim of incorporating the usability aspect of Different Languages, a new requirements has been given to the developers: “When the user is booking a table from the terminal, the system should be able to understand the date, time and table time irrespective of the language used by the user. The interaction diagram does not show the full booking for reasons of visibility on the model”.

The inclusion of this requirement provokes some changes in the design models as shown in Figure 3.2 and Figure 3.3.

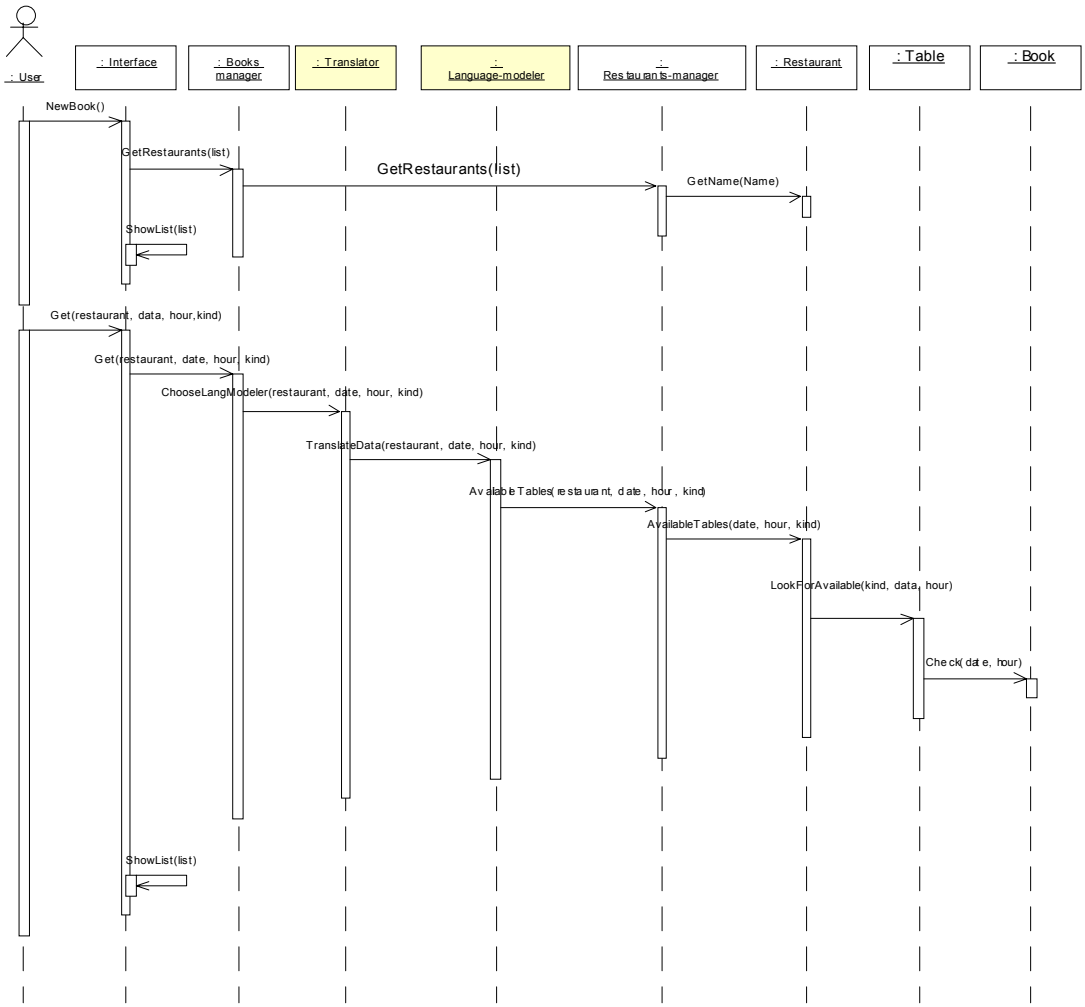


Figure 3.2 Sequence diagram for the first application with the Different Languages mechanism

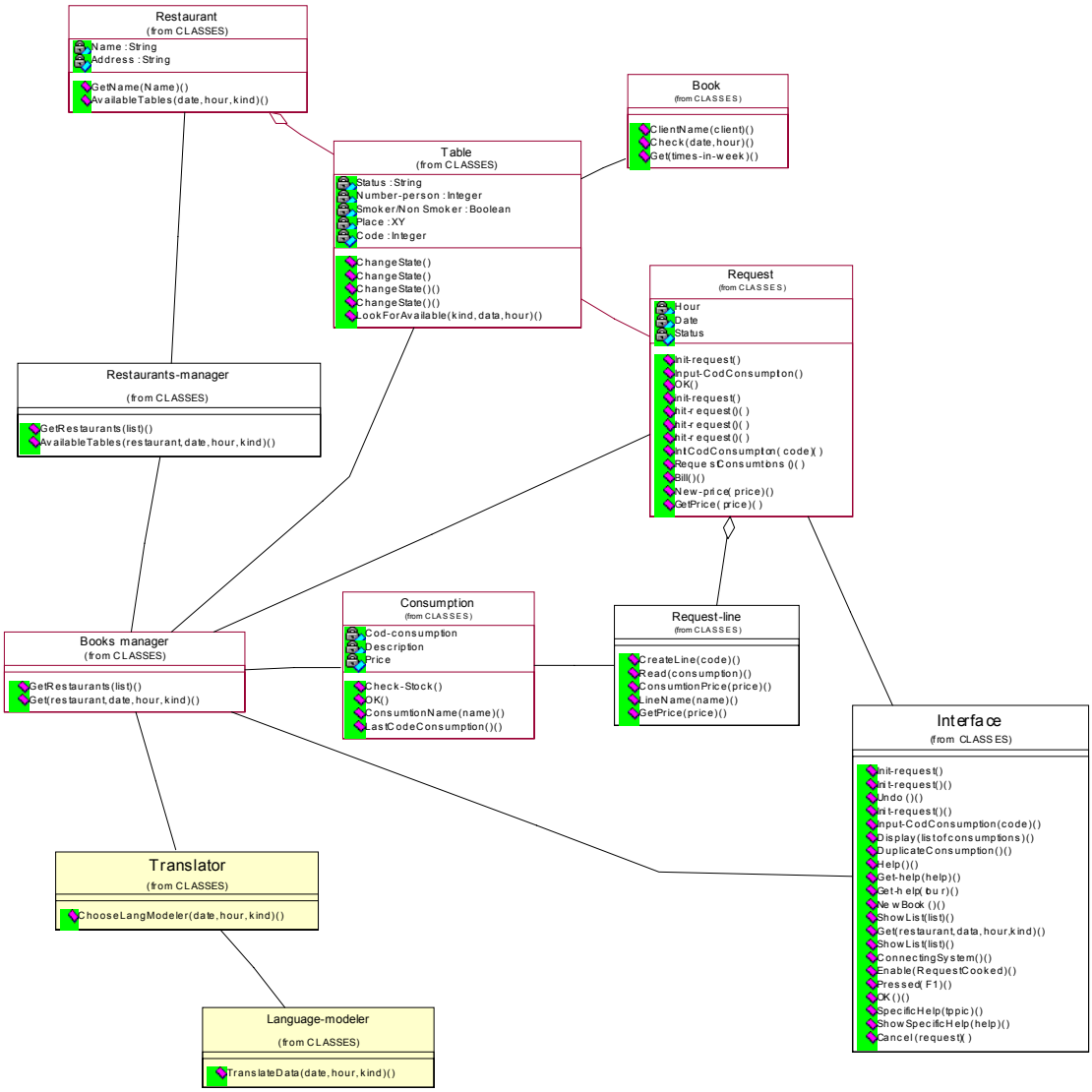


Figure 3.3 Class diagram for the first application with the Different Languages mechanism

STEP 3. Abstraction of the design solution for Different Languages

From the different solutions provided by the different developers (Figure 3.2 and Figure 3.3 are just one of them) it is possible to abstract a more general solution, as shown in Figure 3.4. The different participants in this solution are:

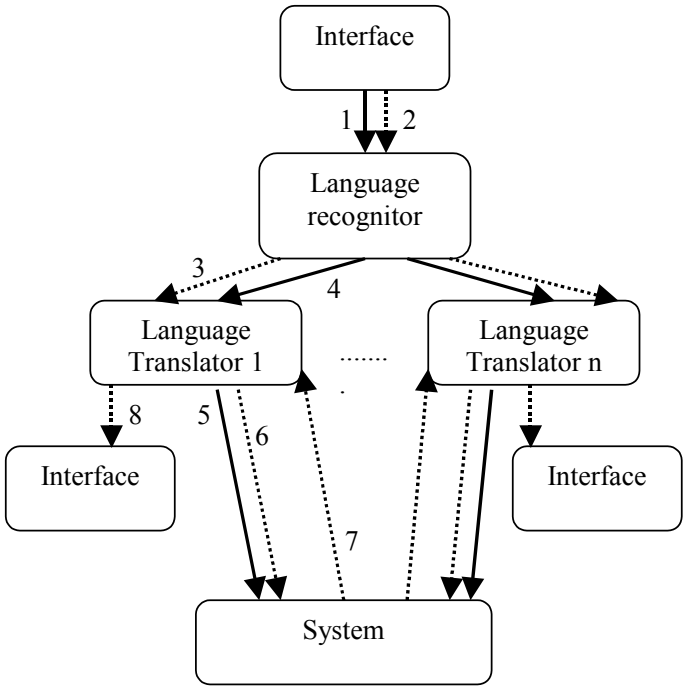


Figure 3.4. Abstract design solution for the Different Languages usability mechanisms

- Interface: collects the operation to be performed and any associated data, which it sends to the Language-recogniser (1) (2). Additionally, once the respective functionality has been processed, the interface receives the data to be displayed to the user from the Language-translator in the language that originated the request (8).
- Language-recogniser is a recogniser, not a translator, which determines the language in which a the respective functionality is requested and sends the data and the functionality request to the respective Language-translator (3) (4).
- Language-translator (i): there may be one for each language that the system is capable of recognising. If there is one for each language, which would be advisable for reasons of system modularity, each Language-translator translates the functionality and any data it receives from the Language-recogniser (3) (4) to a common language understood by the system. Once they have been translated to the common language, it sends them to the system (5) (6). Once the functionality has been processed in the system, it again receives the response data for the executed functionality (7), and again translates them from the common language to the specific language in which the user requested the functionality. After translating, it sends the data to the user (8) through the interface.
- System: it performs the functionality requested by the Language-translator (i), in the common language (5) (6), and returns the respective response to the language-translator in the common language (7).

3.1.2 Second iteration: Checking Design Solutions

Having completed the first iteration to get a design solution for usability patterns, the whole process was repeated with a second application, Amusement park management (whose requirements are shown in Annex B). Similarly to the process presented in section 3.1.1., we have carried out again the three above-mentioned steps for each usability pattern.

In the case of the amusement park system, it was not possible to carry out the second iteration for the following usability patterns:

- Undo, Cancel and Provision of Views because, as it is basically a control system, it was not very sensible to introduce this type of requirements that are more related to management systems.
- Actions for multiple objects, because this system does not perform simultaneous tasks on several objects.

As explained above, for the sake of clarity, this section only includes the second iteration for the pattern Different Languages, whereas Annex D includes the second iteration of each of the usability pattern.

STEP 1. Design models without Different Languages

In the original version of the system there were not an interaction diagram regarding to the Different Languages functionality as it is a new functionality. Figure 3.5 shows the class diagram before including the mechanism of Different Languages

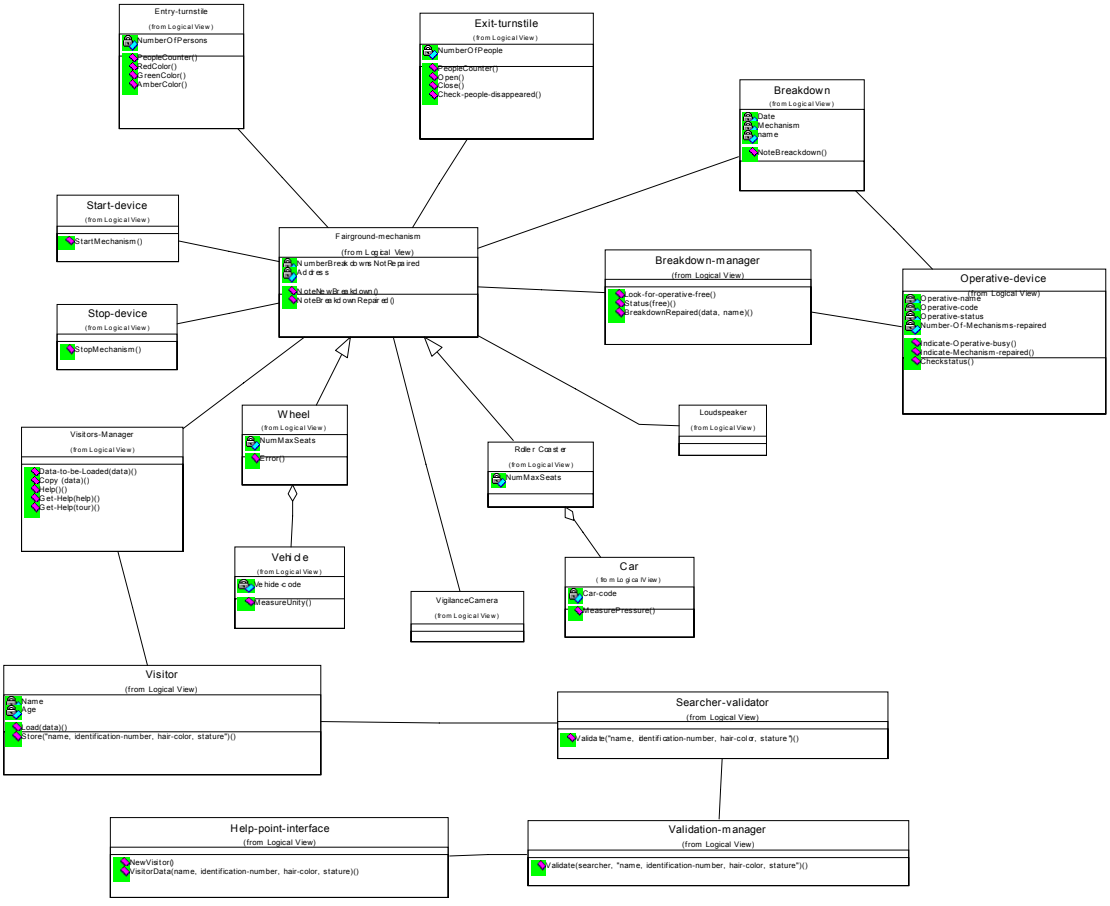


Figure 3.5 Class diagram for the second application without the Different Languages mechanism

STEP 2. Design models with Different Languages

With the aim of including in the application the usability pattern of Different Languages a new requirement has been giving to developers: “The park visitor enters the details of the person who wants to register in any language and the system is capable of translating this to an exchange language so the surveillance system then operates identically when searching for a given subject irrespective of the language in which the subject’s details were entered”.

In order to consider such requirement, a version of the design models has been produced. In particular Figure 3.6 and Figure 3.7 shows the interaction diagram and class diagram respectively produced with this new functionality.

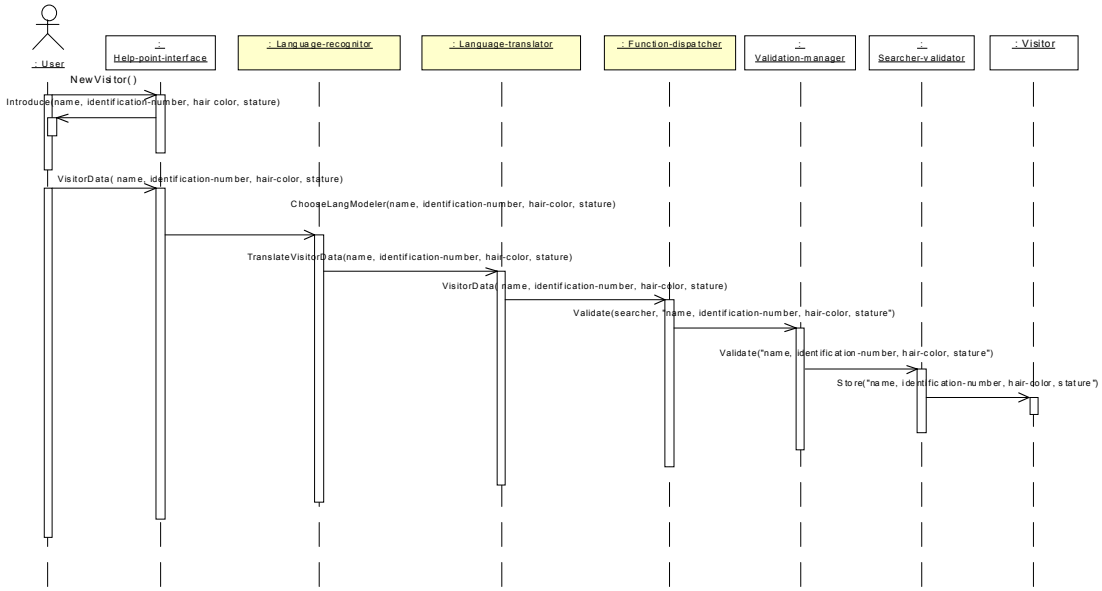


Figure 3.6 Sequence diagram for the second application with the Different Languages mechanism

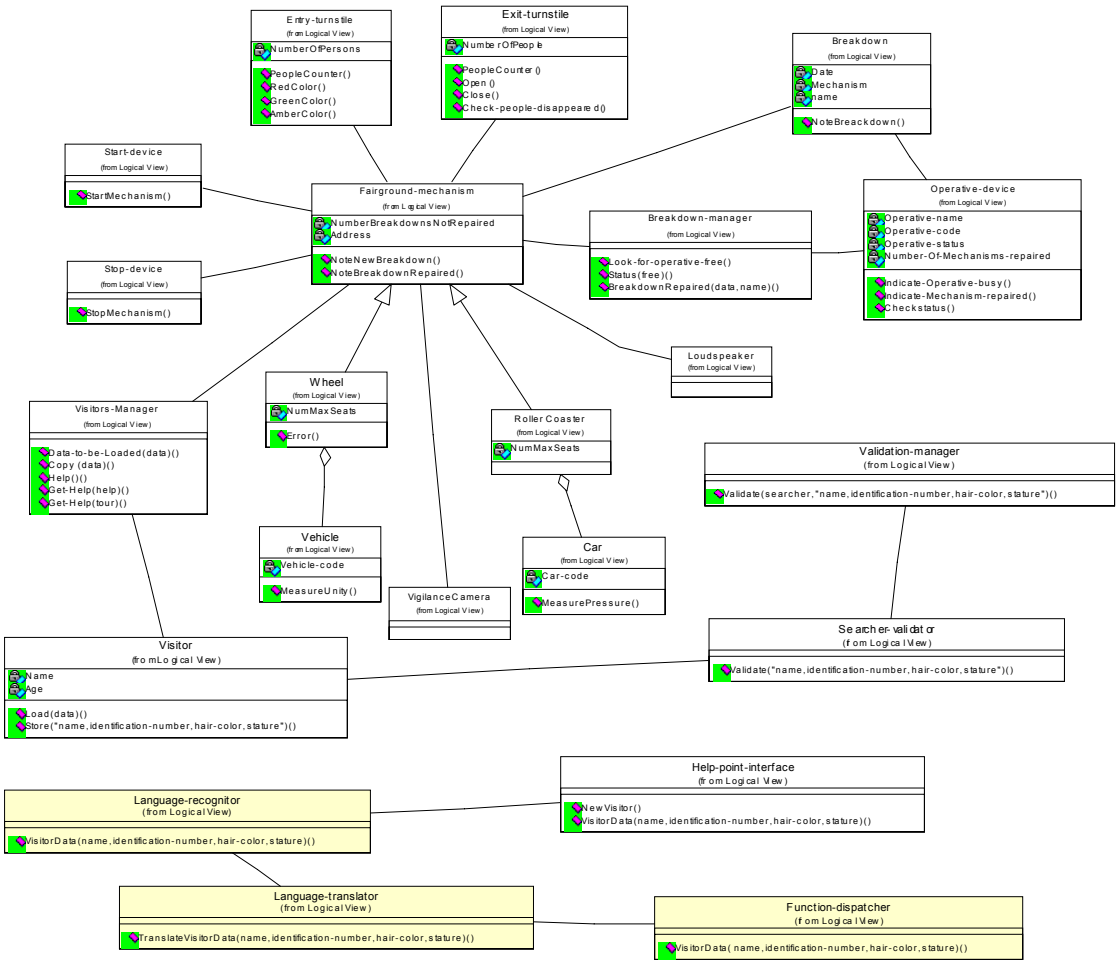


Figure 3.7 Class diagram for the second application with the Different Languages mechanism

STEP 3. Abstraction of the design solution for Different Languages

The abstraction of this solution provides the same design solution shown in Figure 3.4.

3.2 Phase 2. Validating design solutions for patterns with practitioners

The above two iterations had provided a preliminary design solution for the usability patterns. This solution was obtained from software designs made by project researchers. The objective of this third step is to validate these design solutions by practitioners not involved with STATUS. This was done by taking the steps 1 and 2, used in Phase 1 for an Intranet application for managing an advertising company. The application and the results of the two steps have been described in Annex E. The changes to the original development models due to the consideration of the usability patterns are highlighted in each model. Moreover, the solution for the usability pattern abstracted in the previous two iterations can be confirmed.

For reasons of readability, this section only includes a small part of the system, which illustrates what impact the introduction of the Different Languages usability pattern had on one of the system use cases. The remainder of the application is documented in Annex E.

Table 3.1 shows the use case for the functionality that permits the user to query a given contract. This use case has been called Query contract. The table includes all the elements that are useful for defining a use case in expanded format. The requirement P11 is highlighted in the references section, which indicates the inclusion of the Different Languages pattern within the system requirements. This pattern had not been accounted for in the first version of this project which did not include usability patterns:

- **Req P11:** The system should detect the client operating system language and, depending on this, show the messages and labels in the detected language. If the detected language is not one of the parameterised languages or cannot be detected, the messages and labels will be shown in English.

Additionally, note that the description of the use case within system responsibilities now includes step (2) as a result of the inclusion of the Different Languages usability pattern.

Actor	Marketing_administrator, Marketing_user	
Type	Primary and essential	
Purpose	Query the information on a contract with the information requested by marketing.	
Overview:	The user selects the code of the contract to be queried and asks the system for the information.	
References:	*Req* P5, P10, P11, P13, P14	
Typical course of events	**Actor action**	**Typical course of events**
	1. The use case starts when the user enters the query contract application.	2. The system recognises the language of the user and displays the information in the respective language **(UP-Different languages).**
	3 The administrator fills in the data required to identify the contract and asks the system to display the information using a button or abbreviated method (UP-Shortcuts).	4 The system retrieves the information on the contract and this information, if any, is displayed. The user is informed if the contract cannot be displayed (UP- Status Indication).

Table 3.1 Query contract use case

The next step will be to define the system sequence diagram for the Query contract use case. This diagram is illustrated in Figure 3.8. The first system input, Assign Language, has been highlighted in a different colour, as it stems from the consideration of the Different Language usability pattern.

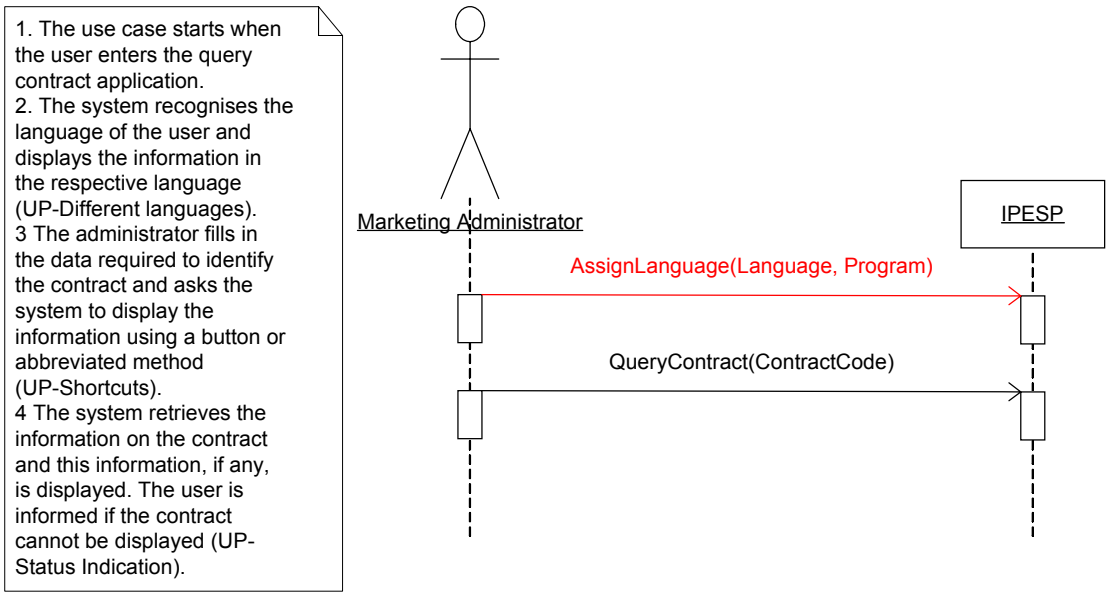


Figure 3.8 System sequence diagram for the Query contract use case

For each of the system inputs represented in the diagram illustrated in Figure 3.8, the respective operation contracts are described in Tables 3.2 and 3.3 below. The operation contract for the AssignLanguage input (Table 3.3) has been highlighted to indicate that it appears as a result of the inclusion of the Different Languages usability pattern.

Name:	***QueryContract(ContractCode:int)***
Responsibilities:	Retrieve and display data on a contract
Cross-references:	Requirements P2, P10 Use case: Query contract
Notes:	
Exceptions:	The contract does not exist
Output:	
Pre-conditions:	
Post-conditions:	The contract data have been retrieved and displayed

Table 3.2 Operation contract for QueryContract

Name:	***AssignLanguage (Language:String, Application)***
Responsibilities:	Assign the language in which the application is to be viewed.
Cross-references:	Requirements: P11 Use case: Update contracts
Notes:	

Exceptions:	The language cannot be assigned.
Output:	
Pre-conditions:	
Post-conditions:	The default application language has been assigned.

Table 3.3 Operation contract for AssignLanguage

A sequence diagram needs to be built for each operation contract defined above in Tables 3.2 and 3.3. Figure 3.9 shows the sequence diagram built for the QueryContract operation contract from Table 3.2. Figure 3.10 shows the sequence diagram built for the AssignLanguage operation contract from Table 3.3.

The classes that emerge as a result of having taken into account the Different Languages usability pattern are shaded in both diagrams. We can check whether the design including usability patterns matches the design for the applications described in Phase 1 (Annexes C and D) for both the example included in this section and the remainder of the application described in Annex E.

As we can see, the classes identified in the sequence diagrams of Figure 3.8 and Figure 3.9 show that the designer has not taken into account as much granularity as was accounted for in the Restaurant (Annex C) and Amusement Park (Annex D) systems, but, in all cases, the design solution for Phase 2 is similar to the solution applied in the applications taken into account in Phase 1. We can conclude, therefore, that the abstractions materialised as architectural patterns in Phase 1 are valid as guidelines for application in other projects.

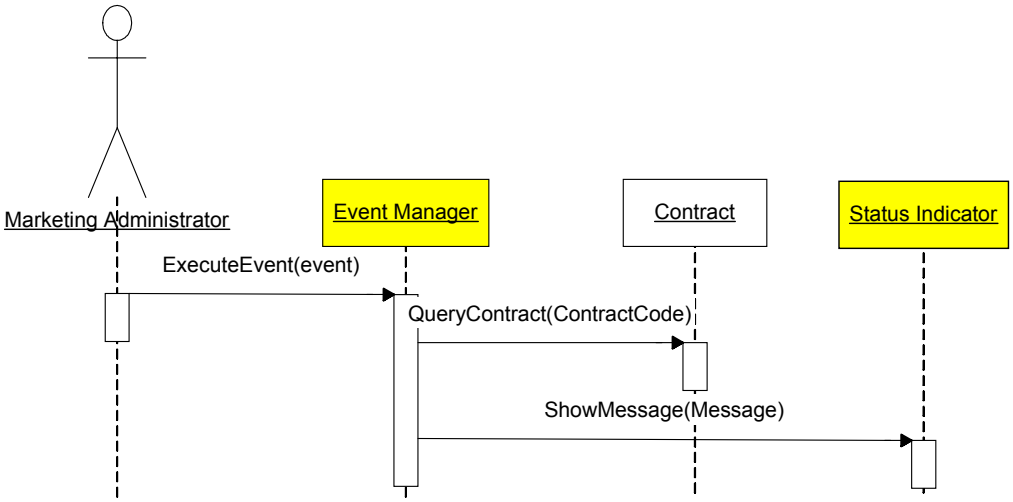


Figure 3.9 Sequence diagram for QueryContract contract

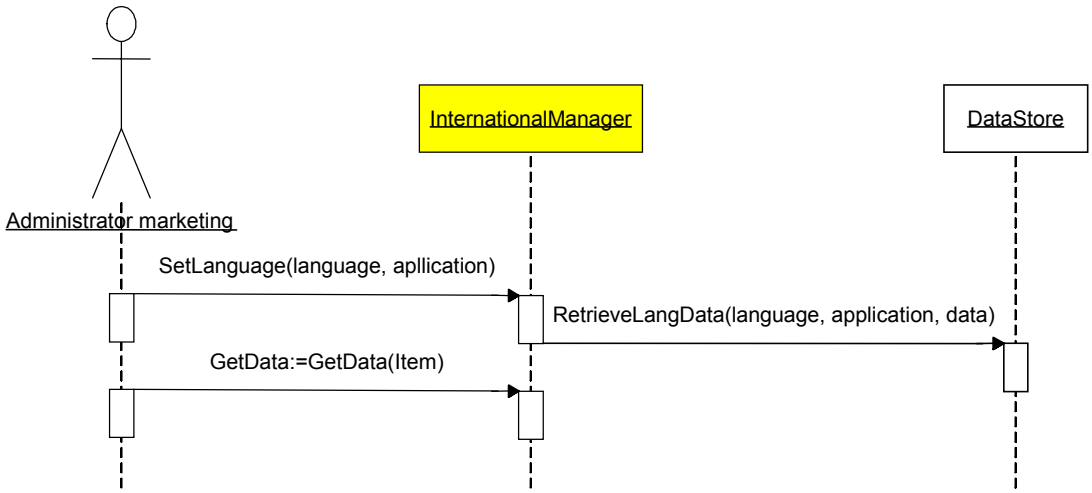


Figure 3.10 Sequence diagram for AssignLanguage contract

4 ARCHITECTURAL USABILITY PATTERNS CATALOGUE

The above process has output general design solutions that can be given to developers to be followed or to serve as inspiration when incorporating a given usability pattern. Annex F describes all the architectural usability patterns output by the entire process. These patterns are described according to the following parameters.

- **Pattern Name**: Patterns must have suggestive names, which give an idea of the problem addressed and the solution in a word or two.
- **Usability Mechanism**: Describes the usability mechanism to be incorporated in the software architecture.
- **Architectural Design Solution**: This describes the elements that make up the architecture, their relationships, responsibilities, etc. The solution does not describe a definite design, as a pattern can be seen as a template that can be applied in many different situations. Particularly, the solution for a specific pattern will be specified from.
  - **Diagram**: A figure that represents the components of the architecture and their iterations. Numbered arrows between the different components will represent the iterations. The arrows with solid lines specify the data flow, while the dotted lines represent the control flow between the components.
  - **Participants**: A description of the components that take part in the proposed solution and the iterations (represented by arrows) to determine how they are to assume their responsibilities.
- **Usability benefits**: Description of which usability aspects (usability properties) can be improved by including this pattern.
- **Usability rationale**: A reasonable argumentation for the impact of pattern application on usability, that is, what usability attributes have been improved, and which ones may get worse. Initially, this feature will be completed with information coming from other authors or from the experience of the consortium members. However, once the patterns have been applied to real applications in WP6, this field will be refined with empirical experience.
- **Consequences**: Impact of the pattern on other quality attributes, like flexibility, portability, maintainability, etc. Table 4.1 present a preliminary relationship among those attributes that, will be checked with the results of empirical experience.
- **Related patterns**: Which architectural patterns are closely related to this one, and what differences there are.
- **Implementation of the pattern in OO**: The architectural patterns provided are patterns that can be applied in any development paradigm. However, as these patterns have been obtained and refined for OO applications, we will provide guides tending to address pattern application in this paradigm. Basically, we will describe the classes deriving from the pattern's main components. These guides are illustrated in the example shown in the following section.
- **Example** of the application of the pattern in question.

	Reliability	Reusing	Learnability	Efficiency	Memorability	User Satisfaction	Performance
Reusing Information	+	+					
Standard Help			+	+			
Tour			-	+	+		
Different Languages	+		+	+		+/-	-
Different Access Methods						+/-	-
Alerts				+		+/-	-
Status Indication						+	
History Logging	+		+				
Undo	+		+				
Form or field validation	+			+			
Provision of views				+		+	
Workflow Model	+			+			
User Profiler			-		-	+	+
Shortcuts			-	+	-		
Context Sensitive Help	+		+	+			
Wizard			+	-	+		
Cancel	+			+			
Multi tasking	-			+			
Commands Aggregation							+
Actions for Multiple Objects	+			+			

Table 4.1. Relationship among quality attributes

5 CAN USABILITY PATTERNS HELP FOR EDUCING USABILITY REQUIREMENTS

Examining the process we followed to get the design solutions for the usability patterns, we realised that the inclusion of a usability mechanism has provided developers with a specific software requirement that considers the respective usability pattern. This modus operandi led us to think that the usability mechanisms to be included in the design should be discussed with users in the requirements analysis phase. In other words, our usability pattern catalogue can serve as guidance for developers during the discussion with users of what usability requirements they would like the system to satisfy.

This is an original idea as compared with the recommendations given by the traditional approaches to usability, which specify two tasks for analysis: establish some usability level requirements (preferably quantitative) and study users and how they perform the task so that there is a natural mapping between the system and user reality. Everything related to usability mechanisms is dealt with as design heuristics, design principles, etc., in the traditional approaches. Our proposal is that the usability mechanisms (not the design solutions) should be brought forward in the software development process and be included in the analysis phase. Accordingly, not only developers but also the users would be able to state their opinions on what usability mechanisms are most critical for a given application, as well as participate in the trade-off between usability and other quality attributes, from which some usability mechanisms may detract.

As Task 3.4 deals with design rather than analysis, we postpone discussing any further findings until we have applied the results of Tasks 3.1, 3.2, 3.3 and 3.4 to a real case. However, a brief overview of how to integrate the entire development process for usability is given in D.3.5. “Usability-centric software architecture design method”.

6 CONCLUSION

Task 3.4 focused on examining the possible design solutions for including usability aspects into each stage. For this purpose, it was necessary to extend the concept of usability pattern that was set out in D.2, adding design-related aspects, like, for example, what effect the inclusion of each of these usability mechanisms has on modules or interactions between modules.

The process followed to provide the design solutions for each usability mechanism was based on induction from several cases studies and on the generalisation of the design solutions set out in each case.

Finally, we developed a catalogue of usability patterns (Annex F) with their respective design solutions, as well as a series of aspects that aim to provide developers with information on how to use these patterns in their development projects.

Both the design solutions provided for each usability pattern and other aspects of the catalogue will be improved in the remainder of the STATUS project, as they are used by the industrial partners in the project context.

7 REFERENCES

[Alexander, 77] Alexander C., Ishikawa S., Silverstein M. A PatternLanguage –Towns-Building-Construction. Oxford University Press, 1977.

[Bass, 01] Bass L., Bonie E. John, Jesse Kates. Achieving Usability Through Software Architecture. Technical Report. CMU/SEI-2001-TR-005, March 2001.

[Casaday, 97] Casaday G Notes on a Pattern Language for Interactive Usability, Proceedings of the Computer Human Interface Conference of the ACM, Atlanta, Georgia, 1997.

[Gamma, 98] Gamma E., Helm R., Johnson R., Wlissides J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1998.

[Perzel, 99] Perzel, , Kane D. (1999) Usability Patterns for Applications o the World Wide Web. PloP'99

[Tidwell, 98] Tidwell, J. Interaction Design Patterns. Pattern Languages of Programming 1998, Washington University Technical Report TR 98-25.

[Welie, 00] Welie M, Troetteberg H. Interaction Patterns in User Interfaces. PloP'00

Anexo A: Detailed Description of Usability Patterns

A.1 Different Languages

Description
Internationalisation refers to the capability of the software to interact with users in different languages.

Relationship with usability properties
This pattern improves system accessibility for users who speak different languages. It supports error prevention by providing a better understanding of the options and tasks that can be performed using the system.

Example:
In internet-based systems, it is quite common to find architectures where the functionality and/or information is duplicated in different languages. In these cases, users usually select the language in which they want to interact with the system.

Other non-internet-based systems also implement the internationalisation attribute to allow to users to manually or automatically select the language. For example, word processing system users can select the spelling and grammar function in the selected language, but also have the option to select another language, depending on the language they used in the document. Microsoft Word is actually able to recognise the language that the user is using and adapt the Autocorrect function to the language in question.

A.2 Different Access Methods

Description
Access method refers to the capability of the software to be accessed using different types of physical devices. Therefore, this attribute will ease system access not only from a desktop or laptop but also using means such as WAP, Web, and interactive TV, for example.

Relationship with usability properties
This pattern improves system accessibility by users using different devices.

Example
Internet weather forecasts can be accessed from a desktop/laptop, but this information can also be obtained using interactive TV or a mobile phone.

A.3 Alerts

Description
An alert is a message from the system to the user that a change of state has occurred that the user ought to know about.

Relationship with Software Architecture
To support the provision of alerts to the user, there needs to be component that monitors the behaviour of the system and sends messages to an output device.

Relationship with Usability Properties
Alerts help to keep the user informed about the status of the system.

Example
If a new e-mail arrives, the user may be alerted by means of an aural or visual cue.

If users make a request to a webserver that is currently off line, they will be presented with a popup window telling them that the server is not responding.

A.4 Status Indication

Description
Users should be provided with information pertaining to the current status of the system.

Relationship with Software Architecture
To support the provision of status information to the user, there needs to be component that monitors the behaviour of the system and sends a message to an output device.

Relationship with Usability Properties
Giving an indication of the system's status provides feedback to the user about what the system is doing at this time and what the result of any action they take will be.

Example
The status bar at the bottom of the screen in Microsoft Word shows the current page number, the position of the cursor on the screen in rows and columns, whether certain modes, such as overwrite, are currently active, and the current language.

A.5 Shortcuts

Description
A shortcut allows an experienced user to activate a feature that may be hidden "under the surface" of the interface with one quick manoeuvre.

Relationship with Software Architecture
To allow shortcuts, several different user interface manoeuvres need to be able to be mapped to the same underlying action.

Relationship with Usability Properties
The provision of shortcuts allows the system to match the user's level of expertise. An experienced user will use the shortcut, whereas a novice will navigate a longer path through the user interface, perhaps receiving more guidance.

Example
Almost all Windows applications provide keyboard shortcuts for commonly accessed items from menus.

Websites may provide "deep links" to pages many clicks away on the front page, if (especially combined with a user profile) they expect the user to want to jump to that page as a result of previous experience.

A.6 Form or Field Validation

Description
If a user is entering multiple items of data on one screen, it is possible to check that each field contains valid data either all at once when the "submit" or "ok" button is pressed (form validation), or individually each time a data item is entered (field validation). With form validation, one invalid entry may lead to the whole form having to be filled in again.

Relationship with Software Architecture
In a web situation, form versus field validation often equates to doing the validation check on the server or on the client, respectively.

Relationship with Usability Properties
This pattern relates to a provision for error management.

Example
These techniques are often employed in forms on websites where the user has to enter a number of different data items, for example, when registering for a new service, or buying something.

A.7 Undo

Description
The ability to undo an action and return to the previous state.

Relationship with Software Architecture
For undo implementation, there must be a component that can record the sequence of actions carried out by the user and the system, as well as enough details about the state of the system in-between each action to go back to the previous state.

Relationship with Usability Properties
Providing the capability to undo an action helps users to correct errors if they make a mistake. It helps the user to feel that they are in control of the interaction.

Example
Microsoft Word provides the ability to undo and redo (repeatedly) almost any action users can take when working on a document.

A.8 Context-Sensitive Help

Description
Context-sensitive help monitors what the user is currently doing and supplies information relevant to the completion of the task in question.

Relationship with Software Architecture
There needs to be provision in the architecture for a component that tracks what the user is doing at any time and targets a relevant portion of the available help.

Relationship with Usability Properties
The provision of context-sensitive help can give the user guidance.

Example
Microsoft Word includes context-sensitive help. Depending on what feature the user is currently using (entering text, manipulating an image, selecting a font style), the Office Assistant will offer different pieces of advice (although some users feel that it is too forceful in its advice).

Depending upon what the mouse cursor is currently pointing to, Word will pop up a small description or explanation of the feature in question.

A.9 Wizard

Description

The wizard pattern presents users with a structured sequence of steps to carry out an operation, which it guides them through one by one. The task as a whole is separated into a series of more manageable subtasks. The user can go back and change earlier steps in the process at any time.

Relationship with Software Architecture

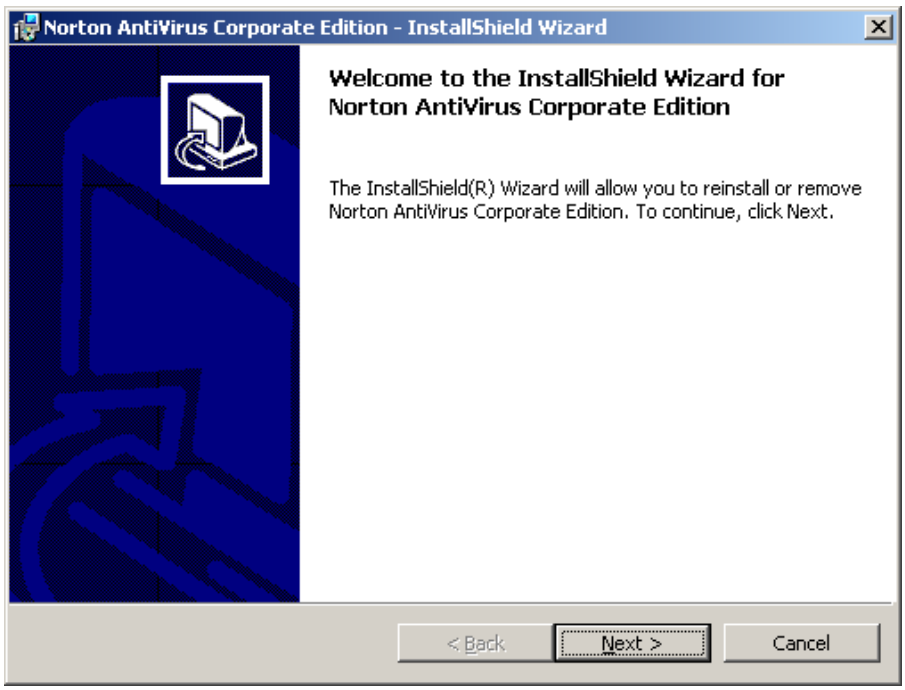
There needs to be provision in the architecture for a wizard component, which can be connected to other relevant components, that is, the component that triggers the operation and the component that receives the data collected by the wizard.

Relationship with Usability Properties

The wizard helps with guidance, showing the user what each consecutive step in the process is.

Example

The install wizard used by most Windows programs guides users through several options for installation



A.10 Standard Help

Description

The system must provide users with help on tasks at any time.

Relationship with Usability Properties

The provision of help will give the user guidance and will improve error management, both error detection and error correction.

Example

It is now usual practice to present users with interactive books, providing search facilities, indexes and even troubleshooting for information searching and problem solving.

A.11 Tour

Description
A tour presents users with information explaining how to do routine system tasks, providing step-by-step guidance.

Relationship with usability properties
The provision of a tour will give the user guidance and will improve error management, both error detection and error correction.

Example
A system that provides presentation facilities could provide a tour of the application, including animated explanations, besides displaying text instructions on how to easily perform such presentations.

A.12 Workflow Model

Description
Modelling workflow provides different users with only the tools or actions that they need to perform their particular tasks.

Relationship with Software Architecture
A component or set of connectors that model the workflow is required, which describe where the data flows. A further model of each system user will be required so as to provide the actions that they need to perform on the data (see A.15 user profile).

Relationship with Usability Properties
Targeting the user interface specifically to each user, depending on the tasks that they need to perform in the workflow, minimises the user's cognitive load.

Example
Logic IS has developed software that models workflow.

A.13 History Logging

Description
Logging the actions that users (and possibly the system) take means that users (or the system) can look back over what was done previously.

Relationship with Software Architecture
To implement this feature, a repository must be provided which can store information about actions. Consideration should be given to how long the data are required for. Actions must be able to be represented in a suitable format for recording in the log.

Relationship with Usability Properties
Providing a log helps users to see what went wrong if an error occurs and may help them to correct that error. Being able to refer to actions that were carried out previously may help with "recognition rather than recall".

Example
Web browsers create a history file detailing all the websites that the user has visited. Databases typically write a log of the transactions that are completed.

A.14 Provision of Views

Description
The system must provide users with different views so that they can see what data they are working on at any time.

Relationship with usability properties
Having data-specific views available at any time provides the user with guidance and will contribute to error prevention.

Example
Before printing a specific document, the system can provide the user with an image of the document as it would be printed.

A.15 User Profile

Description
The software system builds and records a profile of each user, so that specific system attributes (concerning the layout of the user interface, the data or options to show, etc.) can be set and reset each time that a different user accesses the system. Different users may have different roles and require different things from the software.

Relationship with Software Architecture
A repository for user data needs to be provided. This data may be added to or altered either by the user setting a preference or by the system.

Relationship with Usability Properties
Providing the facility to model different users allows a user to express preferences.

Example
Many websites recognise different types of users (e.g., customers or administrators) and present different functionality depending on who is using the site.

Amazon.com builds detailed profiles of each of its customers in order to recommend products that it thinks the user might be interested in on the front page of the site.

A.16 Cancel

Description
Users should be allowed to cancel a command that has been issued if they realise that they have done the wrong thing before an error state is reached. This is different from being able to undo an action after it has finished to return to the previous state.

Relationship with Software Architecture
There needs to be provision in the architecture for the component monitoring the user input to run independently of and concurrently with the components that process actions. The action processing components need to be able to be interrupted.

Relationship with Usability Properties
Being able to cancel commands helps with error management, as if users realise that they have done the wrong thing then they can interrupt and cancel an action before the error state is reached. It also gives users the feeling that they are in control of the interaction.

Example
In most web browsers, if the user types in an incorrect URL and the web browser spends a long time searching for a page that does not in fact exist, the user can cancel the action by pressing the “stop” button before the browser presents them with a “404” page or a dialog saying that they server could not be found.

A.17 Multi-Tasking

Description
Multi-tasking describes the situation where the system (and the user) can manage several tasks at the same time, allowing switching from one task to another as is most conducive to efficiently and effectively doing the work.

Relationship with Software Architecture
A system should be designed so that it can be used along side any other system without interference. It may also be useful for the system to be able to manage more than one set of data at once, for example, a word processor that can hold multiple documents open simultaneously). All of these things have architectural considerations.

Relationship with Usability Properties
Providing a multi-tasking environment gives users the feeling that they are in control of the system, as at any point they can switch to the task that is of most interest to them.

Example
Windows is a multitasking environment which enables the user to run a web browser showing a useful reference website, while writing a document in a word processor, or switch to check an e-mail when one arrives.

Mircosoft Excel allows multiple spreadsheets to be opened at the same time without replicating the controls.

A.18 Command Aggregation

Description
The system should provide the capability to allow users to perform different actions by means of a single command. Macro creation would be an example of this pattern.

Relationship with usability properties
Providing the ability to group a set of commands into one higher level command reduces the users' cognitive load, as they do not need to remember how to execute the individual steps of the process once they have created a macro, they just need to remember how to trigger the macro.

Example
All of Microsoft's office applications provide the ability to record macros or to create them using the Visual Basic for Applications language.

Emacs allows the user to execute strings of commands that can be assigned to special key combinations.

A.19 Actions for Multiple Objects

Description
The same action often needs to be applied to a number of different objects. Providing the user with the possibility of grouping the objects and applying one action to them all "in parallel" will be of help in completing such a task more quickly and accurately. Errors are more likely to be made if each object has to be dealt with separately.

Relationship with Software Architecture
Provision needs to be made in the architecture for objects to be grouped into composites or for it to be possible to iterate over a set of objects performing the same action for each one.

Relationship with Usability Properties
Providing the ability to perform the same action on a number of objects at once reduces the time that it will take the user to complete a task, as the system should be much faster in repeating actions than the human user. The number of clicks (or equivalent actions) that the user has to make to complete the task is reduced.

Example
In a vector based graphics package, such as Corel Draw, it is possible to select multiple objects and to perform the same action (change colour, etc.) on all of them at the same time.

A.20 Reusing Information

Description
This pattern enables the user to move data from one part of a system to another. So users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system.

Relationship with Usability Properties
The data is reused in one or more applications, thus minimising the user's cognitive load and reducing errors.

Example
For example, a project management tool must permit the project manager to copy the particular tasks and their characteristics from one project to another one.

ANNEX B: REQUIREMENTS SPECIFICATIONS FOR THE CASE STUDIES

B.1 CASE 1 specifications: restaurant network management

A restaurant chain wants to automate the reservation process, as well as the orders of each table and the amount there is in the kitchen of each of the products handled to make up each dish, which, obviously, needs to be restocked by the warehouse as the products run out.

TABLE RESERVATIONS

The restaurant customers can telephone to book a table, but the restaurant chain is trying to encourage is the use of point of reservation terminals (PRT) located in the street. The advantage of using these terminals is that they offer the possibility of choosing a table depending on its location within the restaurant, which is impossible over the telephone.

All the PRT are owned by the restaurant chain, although other restaurant chains might offer their services over these terminals in the future. At present, only restaurants belonging to this restaurant chain will be able to selected.

When customers connect to one of these PRT, the terminal asks at what restaurant, on what day and at what time they want to book a table. With the support of the Reservations Centre, which has information on the status of all the restaurant chain tables, the terminal checks whether there is a vacant table at the specified restaurant at the requested time. If there is, the Reservations Centre, with the support of the restaurant in question, first sends a plan of the restaurant and then the vacant tables located in their respective place on the plan to the PRT. Thus, the PRT can reconstruct the plan of the restaurant with the tables that are free.

The tables are divided into tables for smokers, marked with S and for non-smokers marked with NS. Additionally, each table is labelled with the number of people that this table can seat.

Users select a table and specify the number of people who are going to occupy the table, the PRT notifies the Reservations Centre, which then checks with the restaurant that everything is still in order. If everything is OK, the terminal asks the user to specify the name in which the table is to be reserved, which the user enters. The terminal then notifies the Reservations Centre, which makes the reservation, and the terminal issues a ticket specifying the day, time, table and name in which the table has been reserved.

If the customer arrives at the restaurant 20 minutes after the time for which the table was reserved, the system will automatically cancel the reservation.

If there are no tables free at the time specified by the user, the PRT notifies customers, also giving them the option of asking the system for suggestions on restaurants available at the time and on the date requested. If customers want a suggestion from the system, the Reservations Centre provides customers, through the PRT, with a list of possible restaurants. Users can select one, in which case the normal reservation procedure applies, except that the PRT already has some of the customers' particulars.

If there are tables available but none is to the customer's liking at the time for which he wants to book, he can ask the system to specify another chain restaurant that also have vacant tables at the time in question.

If, in either case, the user changes his mind, all he has to do is cancel the operation at any time.

When a customer arrives at one of the chain restaurants, he is asked whether or not he has booked.

If he has a reservation, all he has to do it present the ticket, the table goes from being reserved to being occupied and they are seated in their respective place, provided they do not arrive over 20 minutes later than the time for which they had booked the table.

If, on the other hand, they arrive 20 minutes later than the reservation time, the system will have cancelled the reservation in question and the table will have been released for another possible customer. Therefore, they will be dealt with as if they had not booked. In this case, the reservations manager asks the system to display the tables that are free at the time. If there are free tables, he asks the user if he wants a smoking or non-smoking and how many people there are. The user tells him and, if a table is available, the reservation manager books the table and they are seated. If no tables are available, the reservations manager has to ask the system approximately how long it will take for the next table of the characteristics of the table being requested to become available. The system will be able to calculate this from the status of the different table at a given time. These statuses are:

- Free: if it has not been reserved
- Reserved: if someone has booked
- Occupied: if the diners are at the table
- Ordering: if the waiter is taking the order for the table
- Waiting to be served: if they are waiting to be served
- Served: if the diners already have the food on the table
- Waiting for the bill: if the diners have asked for the bill
- Paying: if the diners already have the bill at their table

Additionally, if there are no tables available and the customer so wishes, he should be informed of another or other chain restaurants that do have free tables.

ORDERS

Once the customers have been seated, the waiter gives them the menu and waits for them to order. The waiters have devices that control part of the system, namely, the orders for each table.

This part of the system waits for the waiter to enter a table number.

When the waiter enters the table number that is going to order, the order time and table that is ordering is automatically recorded. The customers can order both food and drinks, which are both considered as foodstuffs. Each foodstuff has a code that the waiter will enter in the system.

If the customer wants what ingredients a given dish contains, he can ask the waiter, who will then consult the system, keying in the code of the foodstuff followed by a question mark.

The order of each table is composed of order lines, where each order line is a foodstuff. This means that if three dishes of pasta and two glasses of beer are ordered, the order will have five order lines.

The waiter enters the code of each foodstuff and presses accept, before being able to enter the next foodstuff code. The system must be able to check that the ingredients required to prepare the dish ordered are available. If not, that is, if the dish cannot be prepared because one or more ingredients are not available, the waiter will tell the customer that it is not available and ask him to order something else. Of course, if this situation is detected, the warehouse should be told to restock each of the ingredients or drinks that are not available.

When the diners finish ordering, the waiter temporarily closes the order, that is, presses end and the table status switches to “Waiting to be served” as long as they do not order anything else. The system automatically advises the kitchen that there is a new order for a given table. At this point, each line of the order is read again, as are the ingredients of each foodstuff and the amount of the respective products in the kitchen is reduced accordingly. If the amount of any product falls below the threshold established for this foodstuff, it is automatically ordered from the warehouse.

The kitchen manager monitors the incoming orders and tells the cooks. When the dishes have been prepared, the kitchen manager sets the status of order of the table in question to cooked and sends a

message to waiter control for the waiter to collect the order for the specified table. The waiter collects the order and takes it to the respective table and specifies that the table has been served.

INGREDIENTS CONTROL

Additionally, as mentioned above, the ingredients are also controlled from the kitchen. As the exact ingredients of each dish are known, once the trays containing the table order have been prepared, the system is told the stock of ingredients that the dishes or foodstuffs contained have fallen and when these stocks drop below the minimum required in the kitchen, the system automatically advises the warehouse to restock the ingredient.

PAYMENT AND TABLE VACATION

When the diners have finished, they ask the waiter for the bill, which is when the waiter finally closes the order for the table in question and specifies the table status as waiting for the bill. The waiter orders the bill, which is composed of each of the order lines, to be printed. Once printed, it is given to the customers who deposit the money either in cash or by credit card. The waiter goes to the central cash desk and specifies that the table is paying. He then returns with the paid bill and specifies the table status as free.

B.2 CASE 2 specifications: amusement park control

The company DIVERTIMENTO S.A. runs several amusement parks all over Spain. The company is most concerned about:

- Ride control and maintenance: as for a company of this type, a mechanical error could cause material and personal losses that would raise serious problems.
- Visitor safety with respect to theft, losses, etc.,: as the visitors must be as relaxed and confident as possible as regards the park being a safe place for adults and especially children.

With the aim of guaranteeing optimum safety in the park, it wants to implement a pilot project designed to assure the above two points.

PERMANENT RIDE CONTROL AND MAINTENANCE

The only way of detecting faults in rides at present is when the operators responsible for maintenance and control perform these activities.

The company intends to computerise its amusement parks and is going to begin by starting up a pilot project, aimed at equipping one of the amusement parks in the chain with an automatic fault detection system for rides.

The system is initially to be designed to manage the big wheel and the roller coaster, but it is planned to eventually use the control system for these and other park rides.

The big wheel has a series of vehicles, each of which is equipped with a detector thanks to which it is possible to establish at any time whether the vehicle is securely enough anchored to the metallic structure of the big wheel. Each vehicle is equipped with the control software and hardware required to be able to detect the state of the anchorage, and the checks should to be run every three seconds.

If the anchorage were found to be deficient, the vehicle concerned would report this to the Fault Reception Centre (FRCS) and also to the ride of which this vehicle is part. Accordingly, when the ride next stops, there will be a record that one of its vehicles has requested maintenance.

The roller coaster is likewise equipped with detector of anchorage to the following car (if there is one). Each car detects whether it is sufficiently anchored to the following car. If anchorage is deficient, it advises the FRCS and the ride, in this case, the roller coaster.

When the FRCS receives an alert, which specifies the possibly faulty car or vehicle and the ride in question, it immediately locates an available maintenance operator. If none are free, it reports to the component in question that its request cannot be satisfied, and the component will emit the possible fault signal until its request is satisfied.

As each maintenance operator receives a bonus depending on the number of faults he attends to per month. Each operator one is monthly assigned a device which:

1. Manages the faults he attends to monthly

2. And, additionally, reports the possible faults to be attended to, irrespective of which area of the amusement park he is in, that is, he can always be located.

When the FRCS receives a request for fault control and finds a free operator, it sends a message to the operator specifying the street of the park on which the ride is located and the number of the vehicle or car with a possible fault.

The status of the operator's device switches to occupied, indicating that the operator is attending to a possible fault. When the operator has finished repairing the fault, he specifies that he is now free for the next fault request it receives. The device then reports to the FRCS and the repaired component that everything has been correctly solved. This component will advise its ride that the requested maintenance operation has been completed so it can be started up again.

CONTROL OF PERSONS ON RIDES

Additionally, the system will have to be capable of counting the number of people who get on and off a ride for two purposes. It has to control, first, that no more people than the ride is capable of accommodating get on and, second, that everyone gets off the ride when it has stopped.

The ride stop and start controller receives a message specifying that the ride is full for it to start up the ride. This message can come from the entrance turnstile, which detects when the maximum ride occupancy has been reached, or from the operator supervising the ride, if, although it is not full, nobody else is waiting to get on and he thinks that enough time has passed for the ride to start up.

When the ride stop and start device detects that the ride has stopped, it sends a message to the exit turnstile for it to prepare to let the people out. What the exit turnstile does before letting the people off the ride is check whether the ride has a missing person search alert. If it does, it will order the speaker there is at each ride to broadcast a message naming the person in question and telling this person to stay the ride entrance until someone comes to pick him up.

Having run the check, the exit turnstile starts to let the people out. This exit turnstile knows how many people there are on the ride thanks to the entrance turnstile, and accordingly knows how many people have to get off the ride.

When the exit turnstile determines that the number of people who have got off the ride is equal to the number of people who got on, it sends a message to the entrance turnstile to reset the counter of people on the ride to zero, unblock and display the green indicator for people to get on the ride. If the exit turnstile has not unblocked the entrance turnstile by 5 minutes after the ride has stopped, it means that someone has not got off and the operator will have to go in to look of him or her.

When the entrance turnstile receives the unblock message from the exit turnstile, it first consults the ride to see if it has any unrepaired fault. This will be reflected in the ride when one or more vehicles or cars request repair. The ride has an unrepaired faults counter and the entrance turnstile will only go green for users to get on if this counter is set at 0. Otherwise, it remain amber, indicating that it is waiting for repair.

PARK SURVEILLANCE MANAGEMENT

The park will be equipped with two SOS terminals by means of which the parents can enter their children. These terminals ask parents the name, ID card number, age, hair colour and stature of the

person to be registered. If parents discover at any time that someone has gone missing, they should go to the nearest SOS terminal and enter the member of the family as missing:

- If the person in question has already been entered as a park visitor, all they will have to do is to enter the ID card no. of the person who has gone missing.
- If the person has not been registered as a park visitor, then they will have to enter the above-mentioned identification particulars and then report the disappearance at the same terminal.

Once the system has all the available data, it sends the data on the missing person to all the surveillance cameras located throughout the park and the rides. These cameras are equipped with an algorithm based on computer vision techniques so that if they detect an individual of the specified characteristics within their field of view, they automatically report to the surveillance centre, which will then send the operators responsible for collecting the person in question to the point indicated by the camera. Additionally, as the cameras are associated with a speaker, the camera that detected the person will be capable of asking its loudspeaker to broadcast the name of the person in question, telling him that he is being looked for and asking him not to move from where he is.

If the individual in question is identified when getting on a ride, this ride will be alerted by its respective camera and when the ride stops, a message will be broadcast by the loudspeaker associated with the cameras, as described in the “Control of Persons on Rides” section.

ANNEX C: PHASE 1 FIRST ITERATION: THE RESTAURANT MANAGEMENT CASE

C.1 Reusing Information First Iteration

STEP 1. Design solution without Reusing Information

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

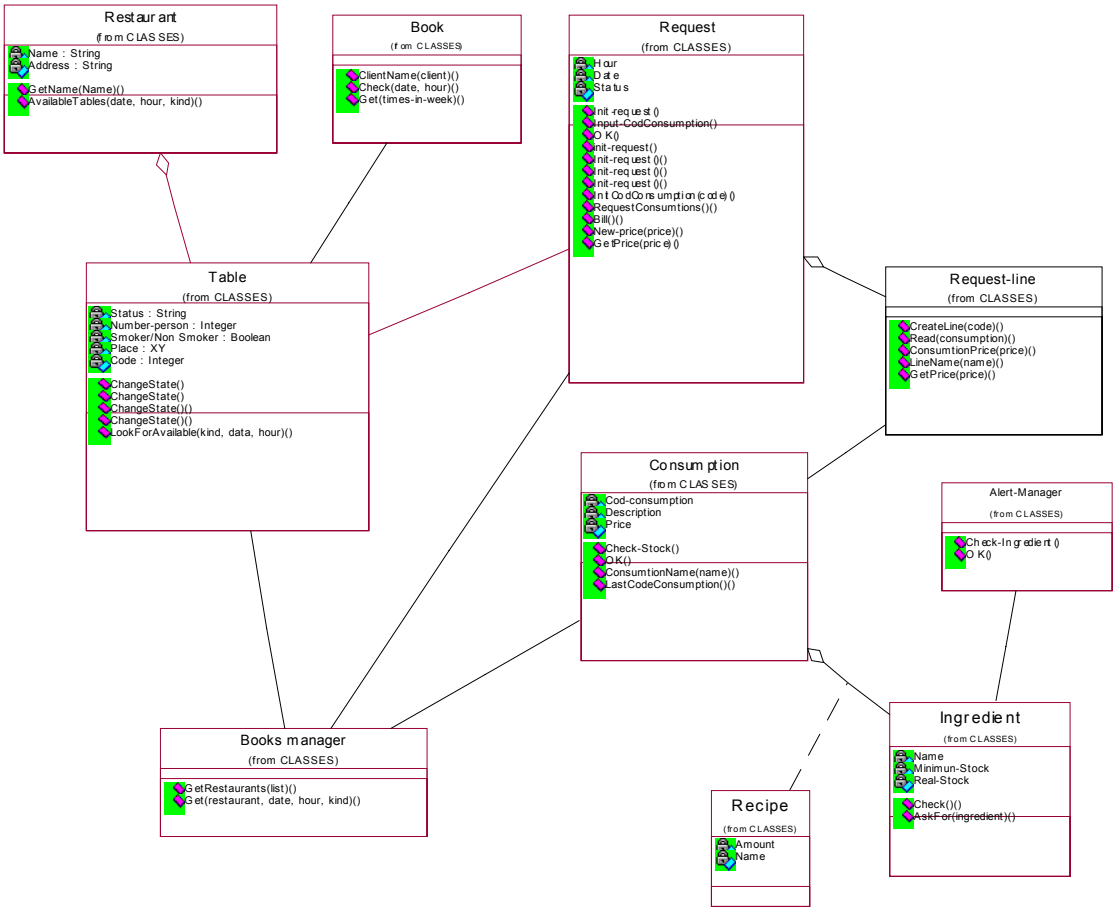


Figure C.1 Class diagram for the first application without Reusing Information mechanism

STEP 2. Design solution with Reusing Information

Requirement: the waiter inputs the foodstuff code and, as the next consumption ordered is the same, the waiter uses the "duplicate last foodstuff" function.

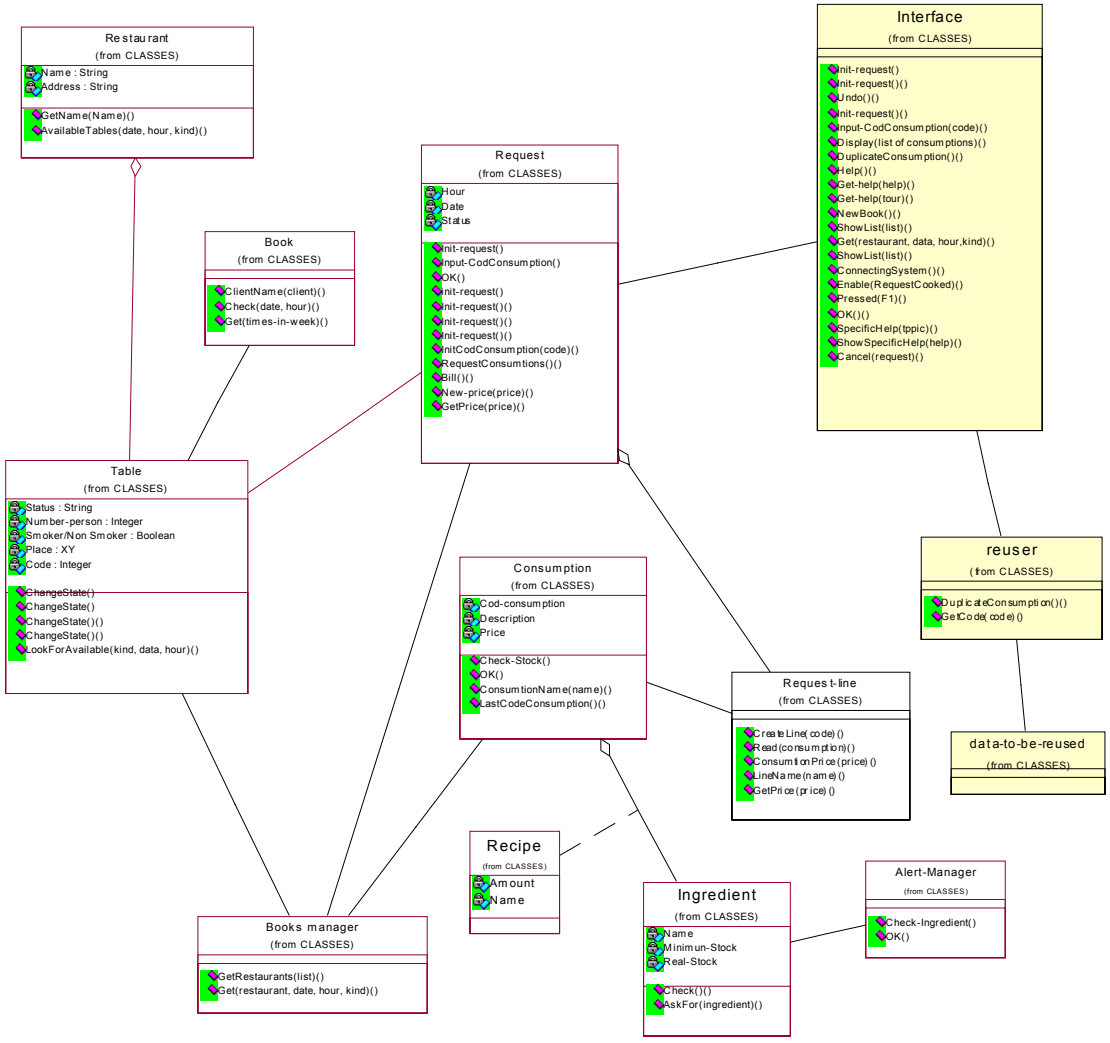


Figure C.2 Class diagram for the first application with Reusing Information mechanism

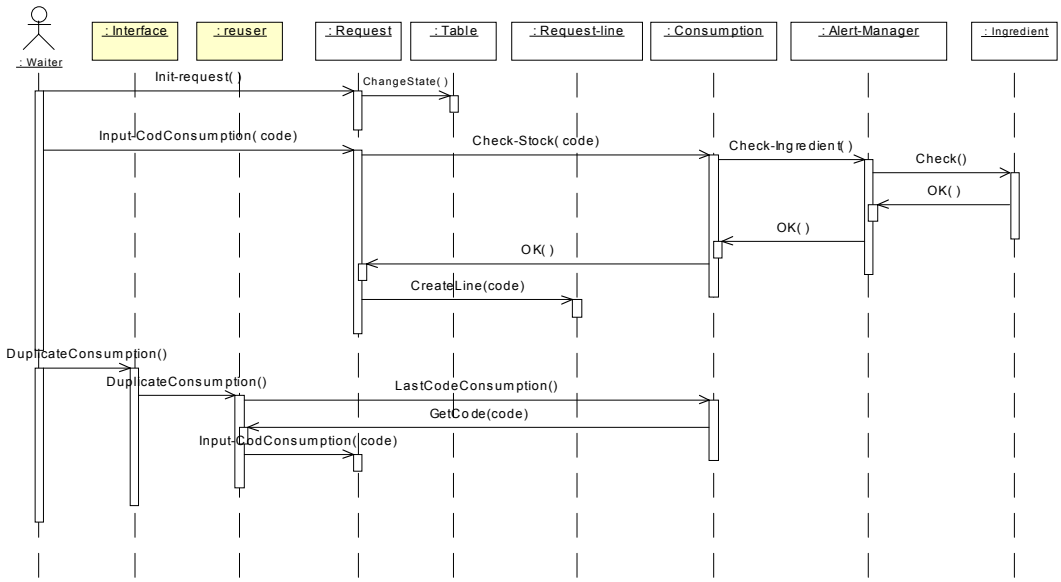
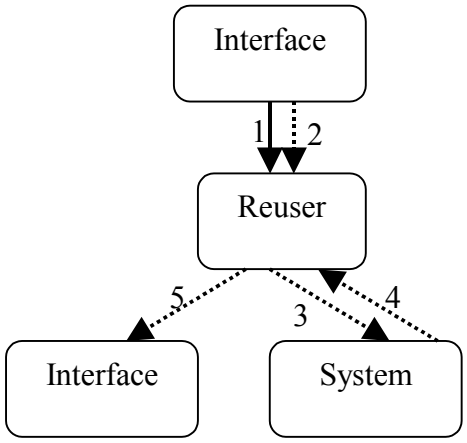


Figure C.3 Sequence diagram for the first application with Reusing Information mechanism

STEP 3. Abstraction of the design solution for Reusing Information

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: collects the data to be processed by the reuser pattern and finally displays the operation results (if the user needs to see the result). Interface sends the data to be processed (1) and the function requested by the interface (2), i.e. copy, paste, move, etc., to Reuser. Also, once the reuser pattern has been applied the results of the requested function will be displayed on the interface (5), unless the requested function was "copy".
    - Reuser: is the module that gathers the information provided by the interface and manipulates these data according to the requested function (copy, paste, move, etc.). Reuser receives the data to be manipulated as well as the function to be executed (1) (2). If Reuser does not store the data to be manipulated internally, it has to send these data to the system (3), as happens, for instance, with the Copy function. Also if Reuser does not store the data internally, it has to ask for these

data from the part of the system where they are stored (4) as happens with the paste or move functions.

- System: this component is optional and is only necessary when the Reuser module does not store the data internally.

C.2 Standard Help First Iteration

STEP 1. Design solution without Standard Help

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

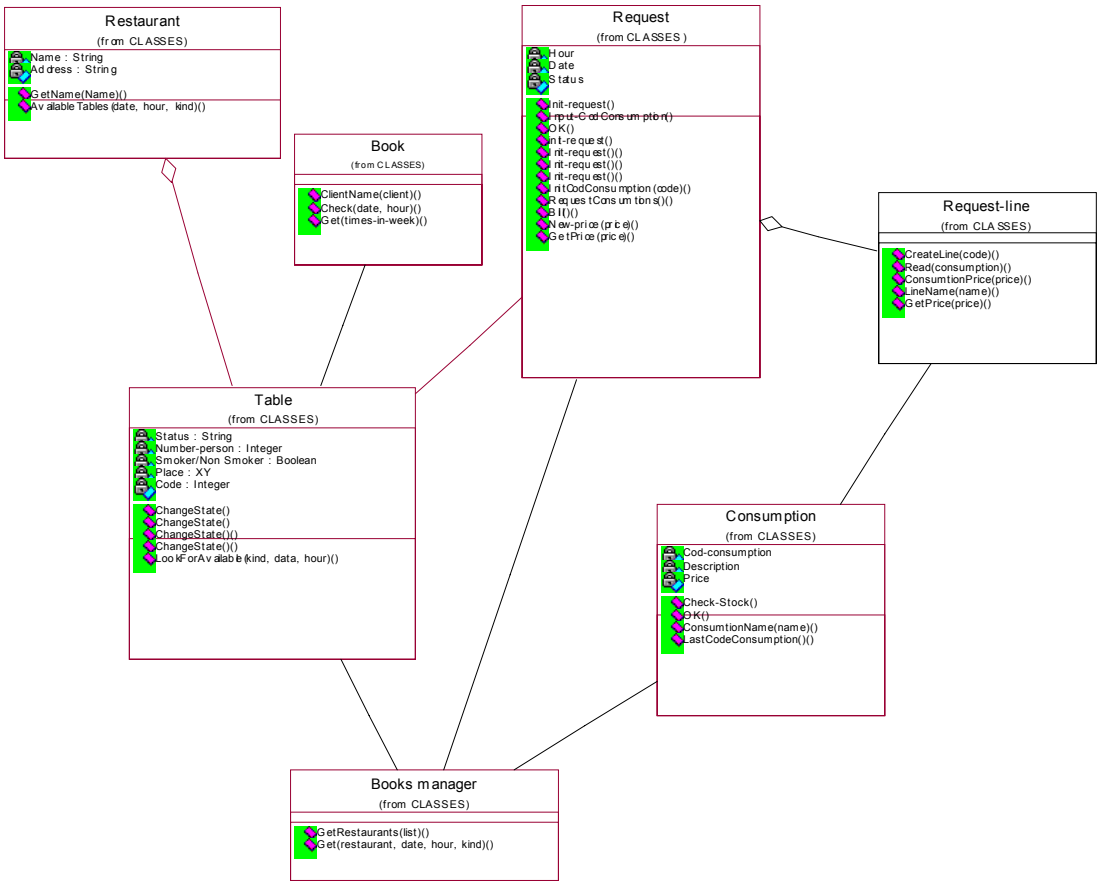


Figure C.4 Class diagram for the first application without Standard Help mechanism

STEP 2. Design solution with Standard Help

Requirement: the user can push the Help button

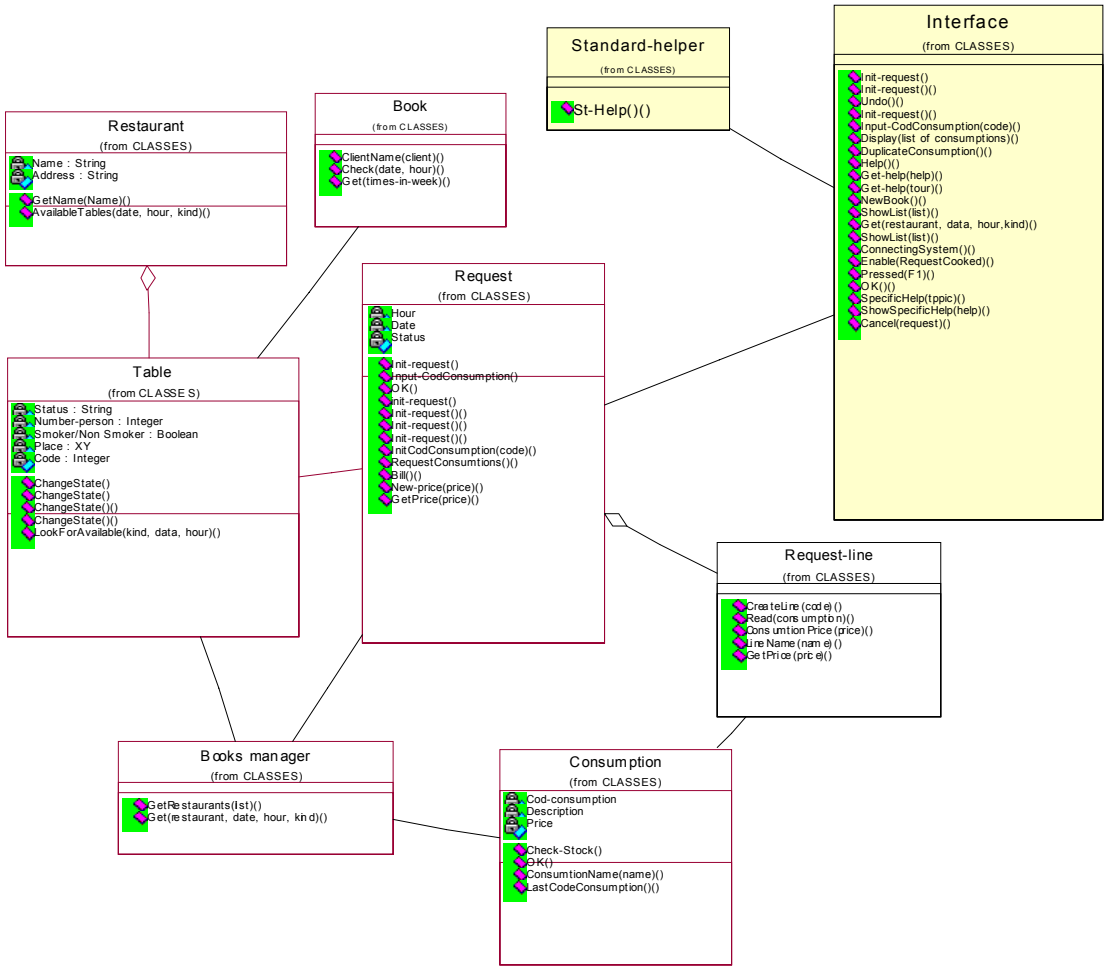


Figure C.5 Class diagram for the first application with Standard Help mechanism

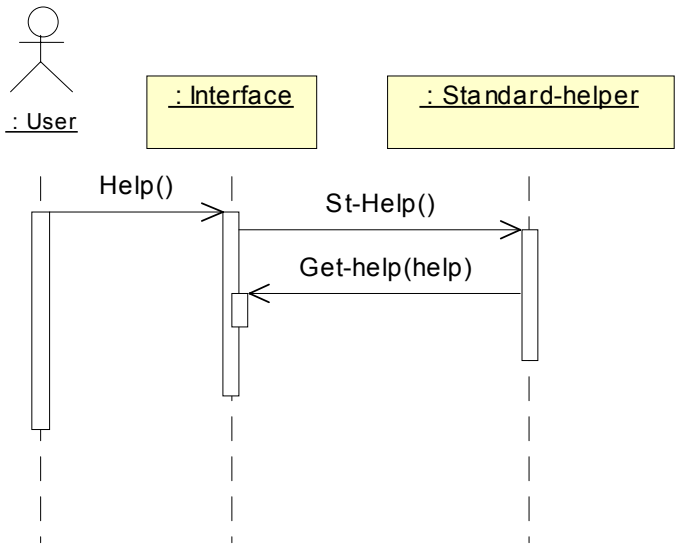
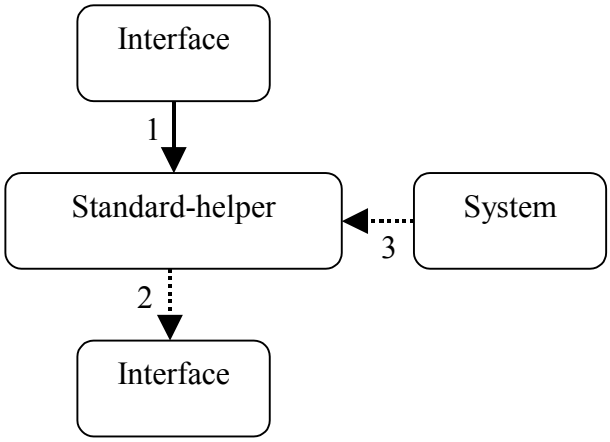


Figure C.6 Sequence diagram for the first application with Standard Help mechanism

STEP 3. Abstraction of the design solution for Standard Help

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: gathers the information from the help application and sends this information to the module which manages the help (1). Also it will show the help information sent by the Standard-helper (2)
    - Standard-helper: will show a general help (that is, not specialised) for the application. This help is usually identified as an html, doc, etc., document. This component receives the application from the interface (1) and sends the respective data to the interface (2). If the help is not stored in this component, the help will be provided for another component using the data flow from System (3).
    - System: this component is optional and represents the part of the system where the help is stored if the Standard-helper does not store the help internally. It will be the system that provides the Standard-helper with the help (3).

C.3 Tour First Iteration

STEP 1. Design solution without Tour

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

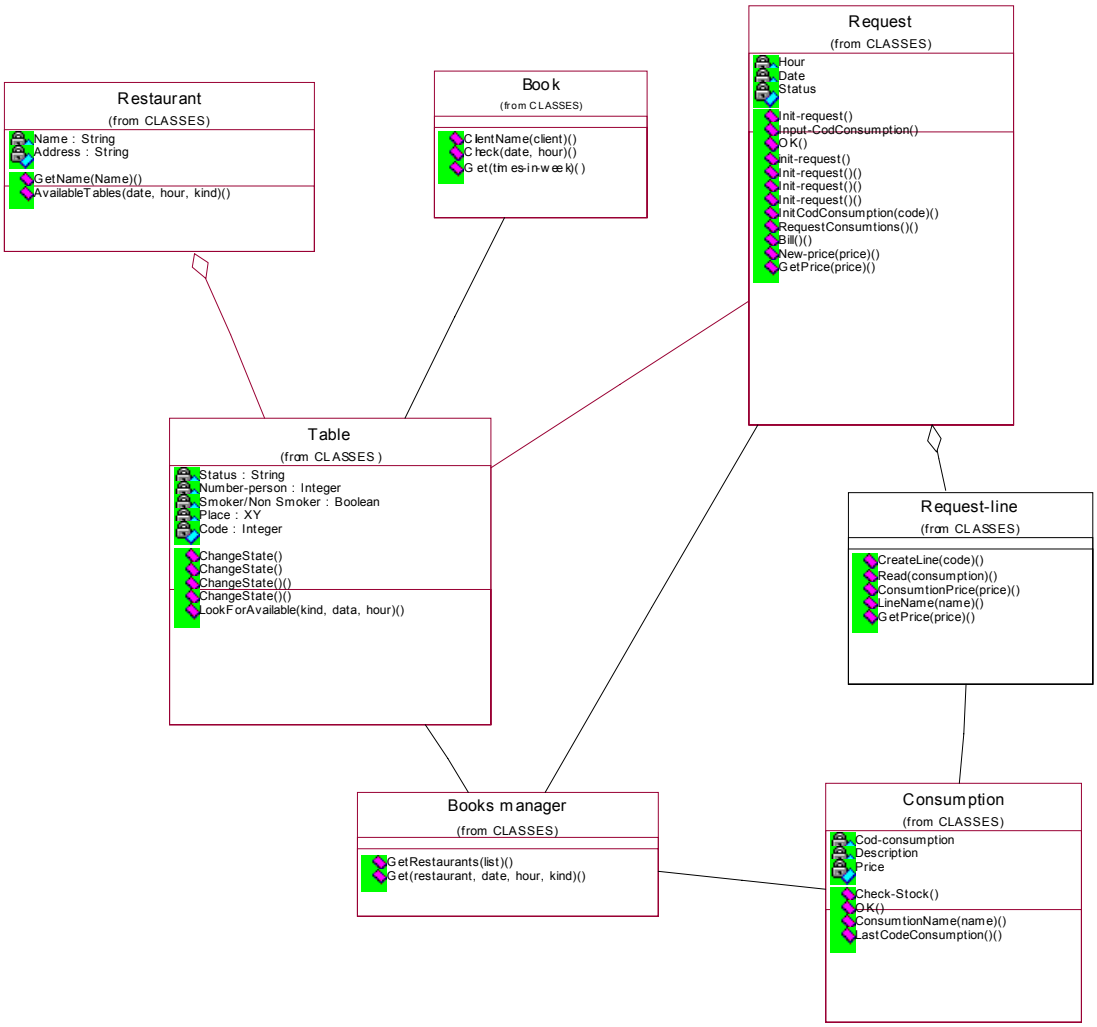


Figure C.7 Class diagram for the first application without Tour mechanism

STEP 2. Design solution with Tour

Requirement: the user can push the Guided Help button

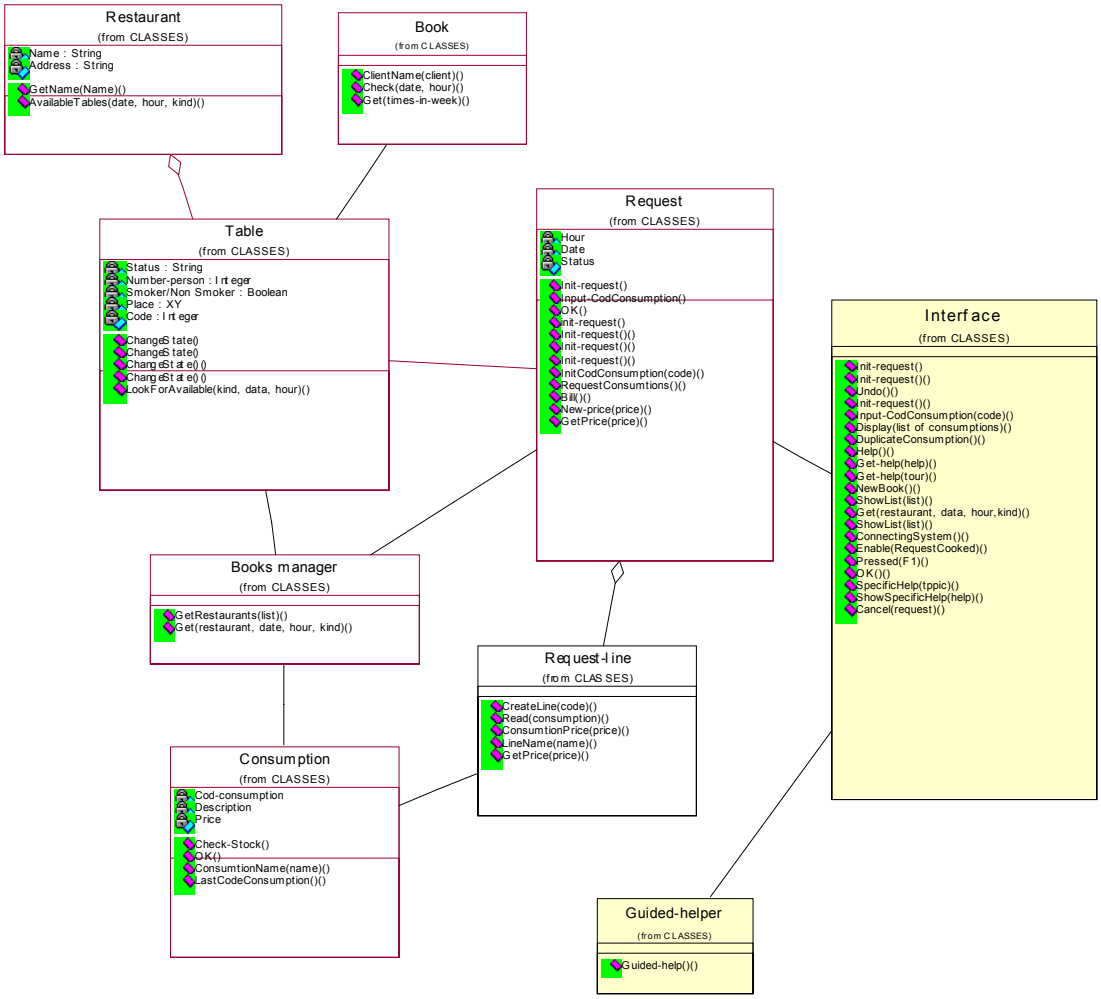


Figure C.8 Class diagram for the first application with Tour mechanism

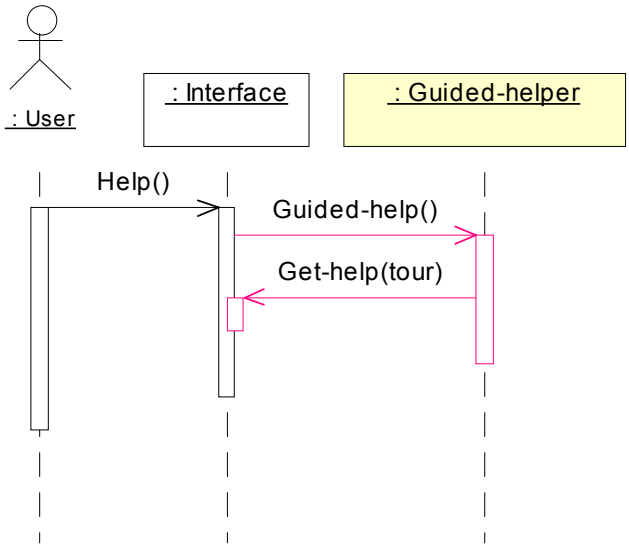
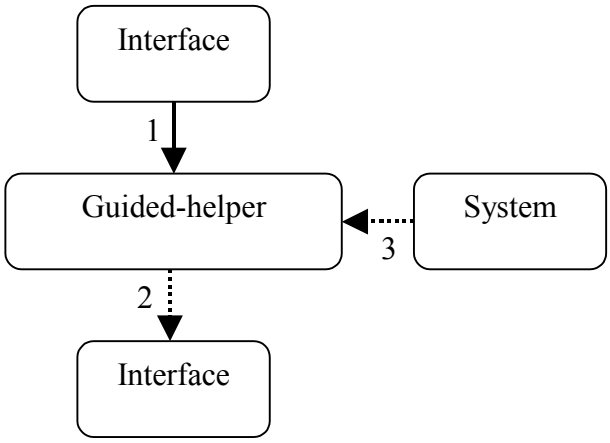


Figure C.9 Sequence diagram for the first application with Tour mechanism

STEP 3. Abstraction of the design solution for Tour

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: collects the guided help request and sends it to the Guided-helper (1). Additionally, it will display the help information it receives from the Guided-helper (2).
    - Guided-helper: displays a guided help for the application for which the help has been described (2). This help can range from a pre-recorded tour of the application, to an interactive tour, which involves the development of a separate application. If the help is not stored internally in this component, this help will be provided by any other part of the system through the information flow from system (3).
    - System: this is an optional component and represents part of the system in which the help will be stored if the Guided-helper does not store the information internally. System will, therefore, be responsible for providing the Guided-helper with the help through (3).

C.4 Different Languages First Iteration

STEP 1. Design solution without Different Languages

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

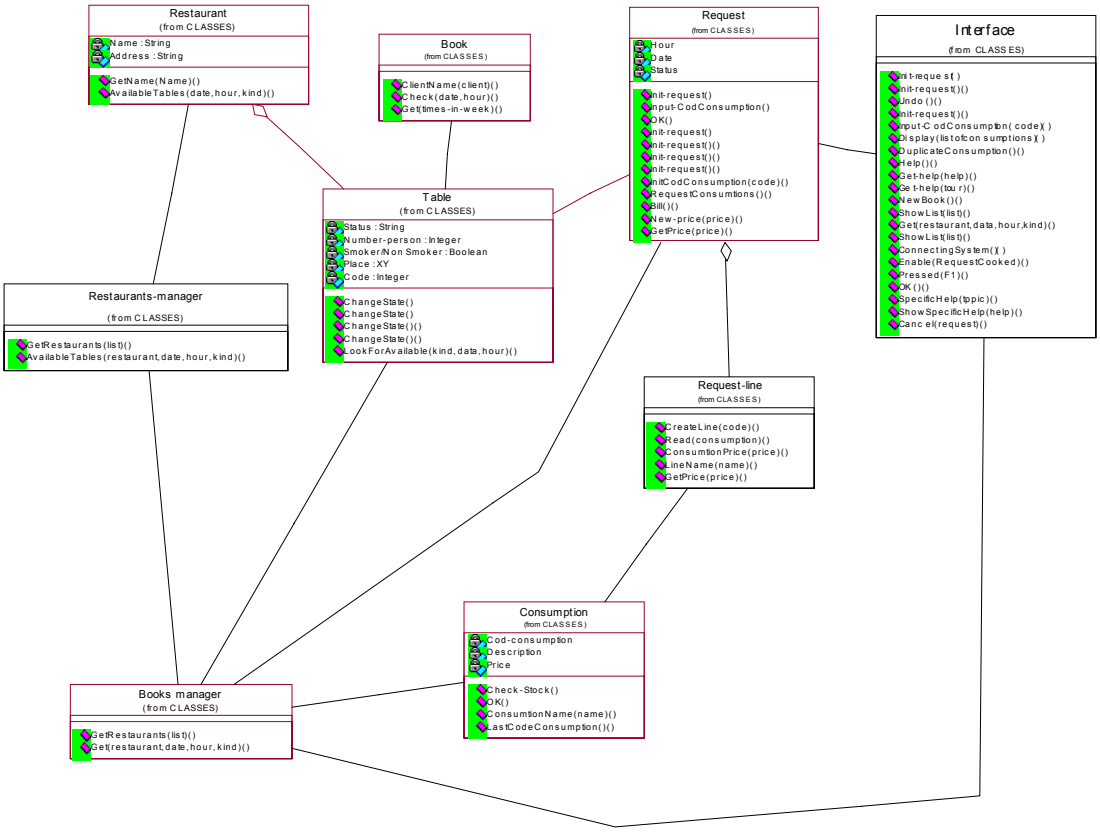


Figure C.10 Class diagram for the first application without Different Languages mechanism

STEP 2. Design solution with Different Languages

Requirement: When the user is booking a table from the terminal, the system should be able to understand the date, time and table time irrespective of the language used by the user. The interaction diagram does not show the full booking for reasons of visibility on the model.

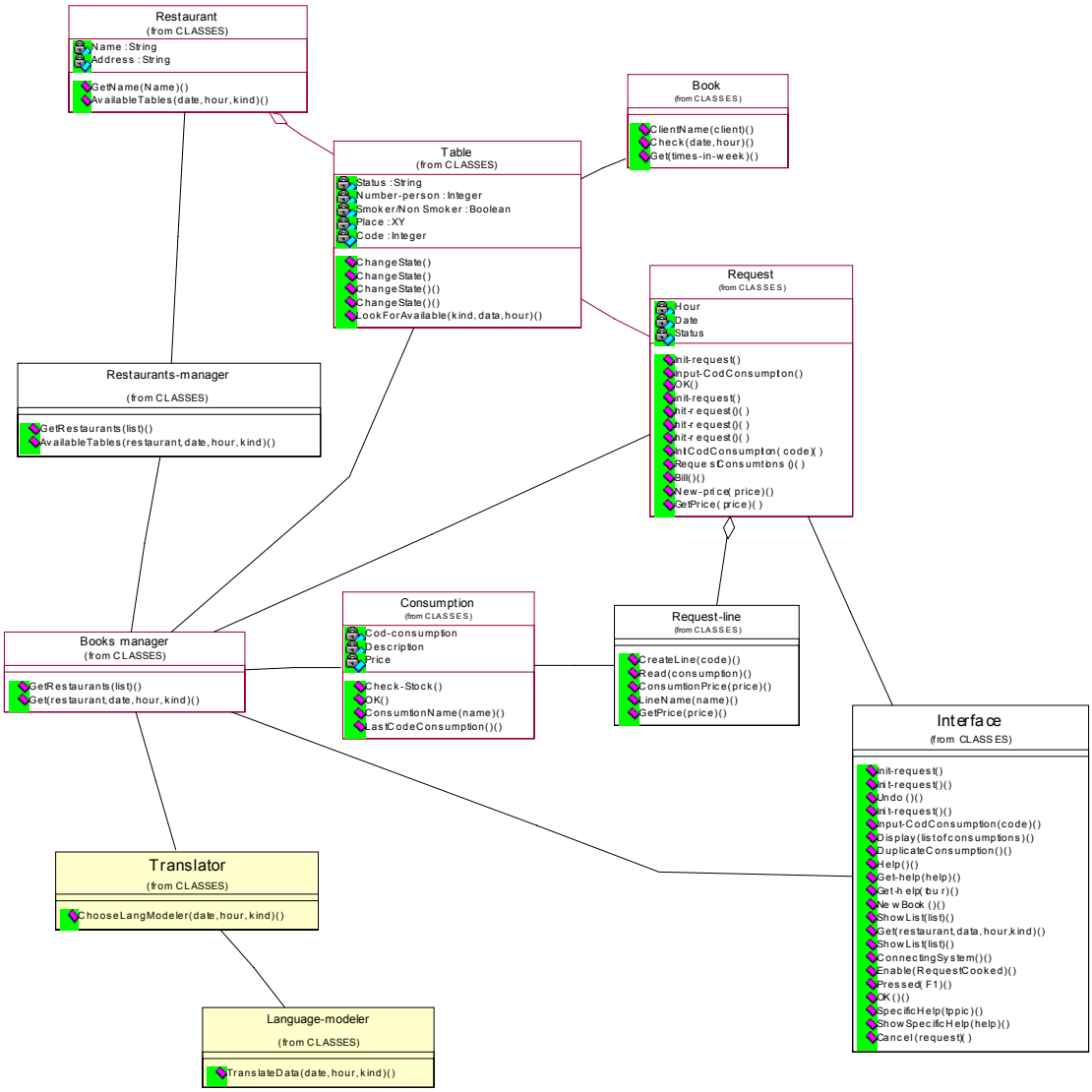


Figure C.11 Class diagram for the first application with Different Languages mechanism

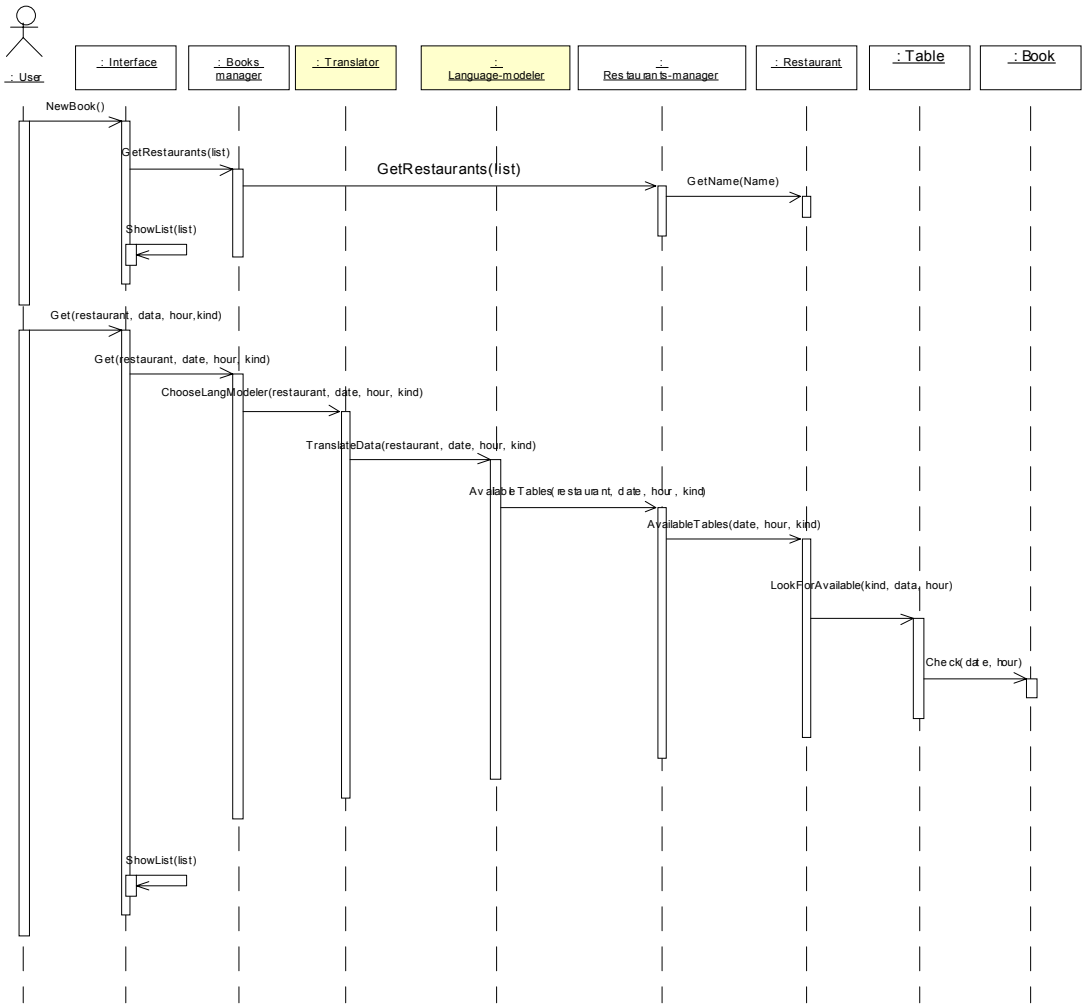
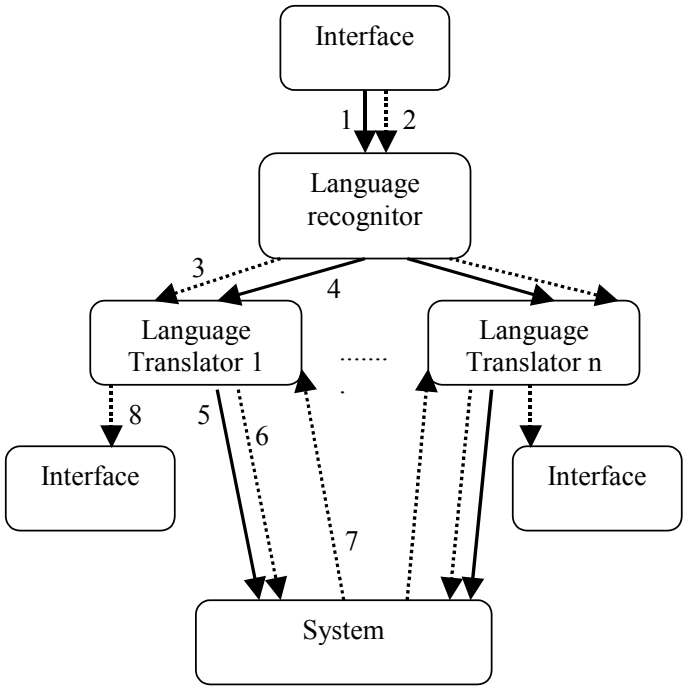


Figure C.12 Sequence diagram for the first application with Different Languages mechanism

STEP 3. Abstraction of the design solution for Different Languages

- **Solution:**
  - Diagram:



- Participants:
  - Interface: collects the operation to be performed and any associated data, which it sends to the Language-recogniser (1) (2). Additionally, once the respective functionality has been processed, the interface receives the data to be displayed to the user from the Language-translator in the language that originated the request (8).
  - Language-recogniser is a recogniser, not a translator, which determines the language in which a the respective functionality is requested and sends the data and the functionality request to the respective Language-translator (3) (4).
  - Language-translator (i): there may be one for each language that the system is capable of recognising. If there is one for each language, which would be advisable for reasons of system modularity, each Language-translator translates the functionality and any data it receives from the Language-recogniser (3) (4) to a common language understood by the system. Once they have been translated to the common language, it sends them to the system (5) (6). Once the functionality has been processed in the system, it again receives the response data for the executed functionality (7), and again translates them from the common language to the specific language in which the user requested the functionality. After translating, it sends the data to the user (8) through the interface.
  - System: it performs the functionality requested by the Language-translator (i), in the common language (5) (6), and returns the respective response to the language-translator in the common language (7).

C.5 Different Access Methods First Iteration

STEP 1. Design solution without Different Access Methods

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

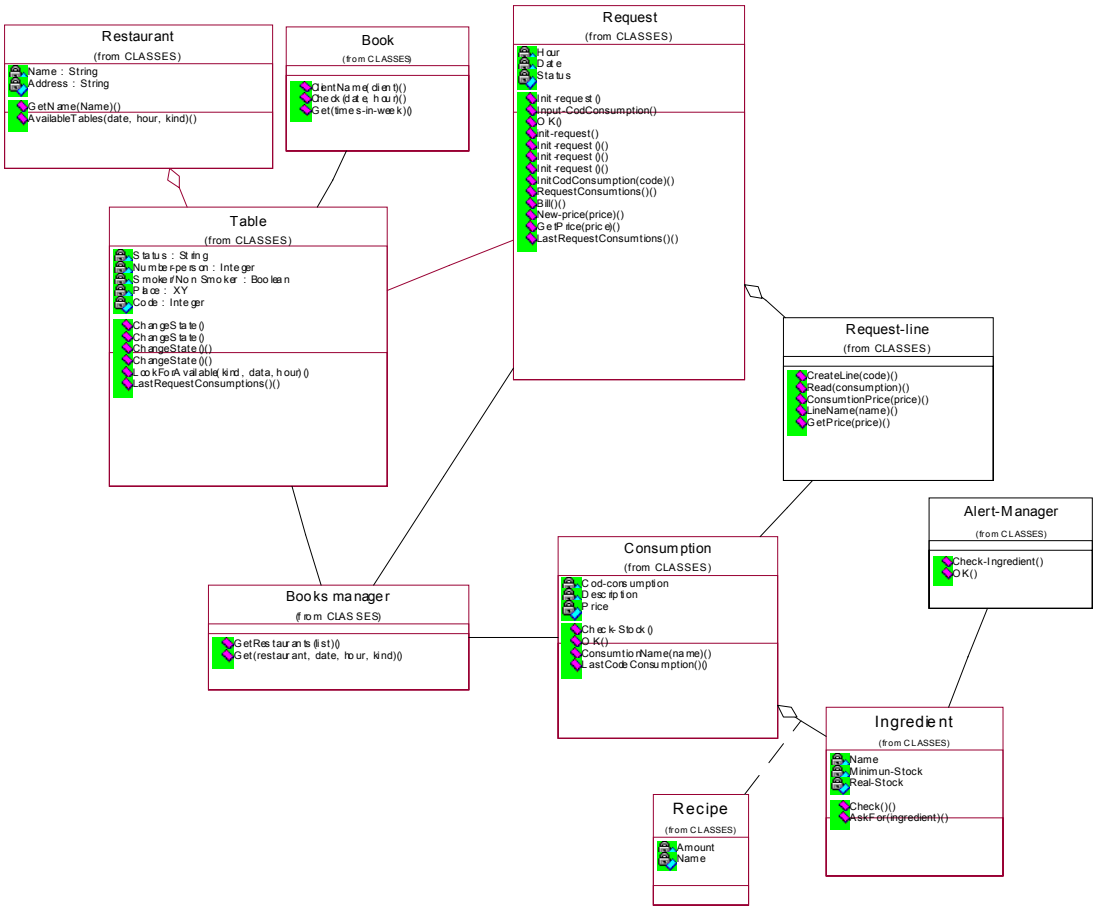


Figure C.13 Class diagram for the first application without Different Access Methods mechanism

STEP 2. Design solution with Different Access Methods

Requirement: The waiter can ask the waiter device what foodstuffs a table has ordered by simply saying “I want to know what foodstuffs table x has ordered”. Additionally, the waiter’s device is capable of verbally reproducing all the foodstuffs.

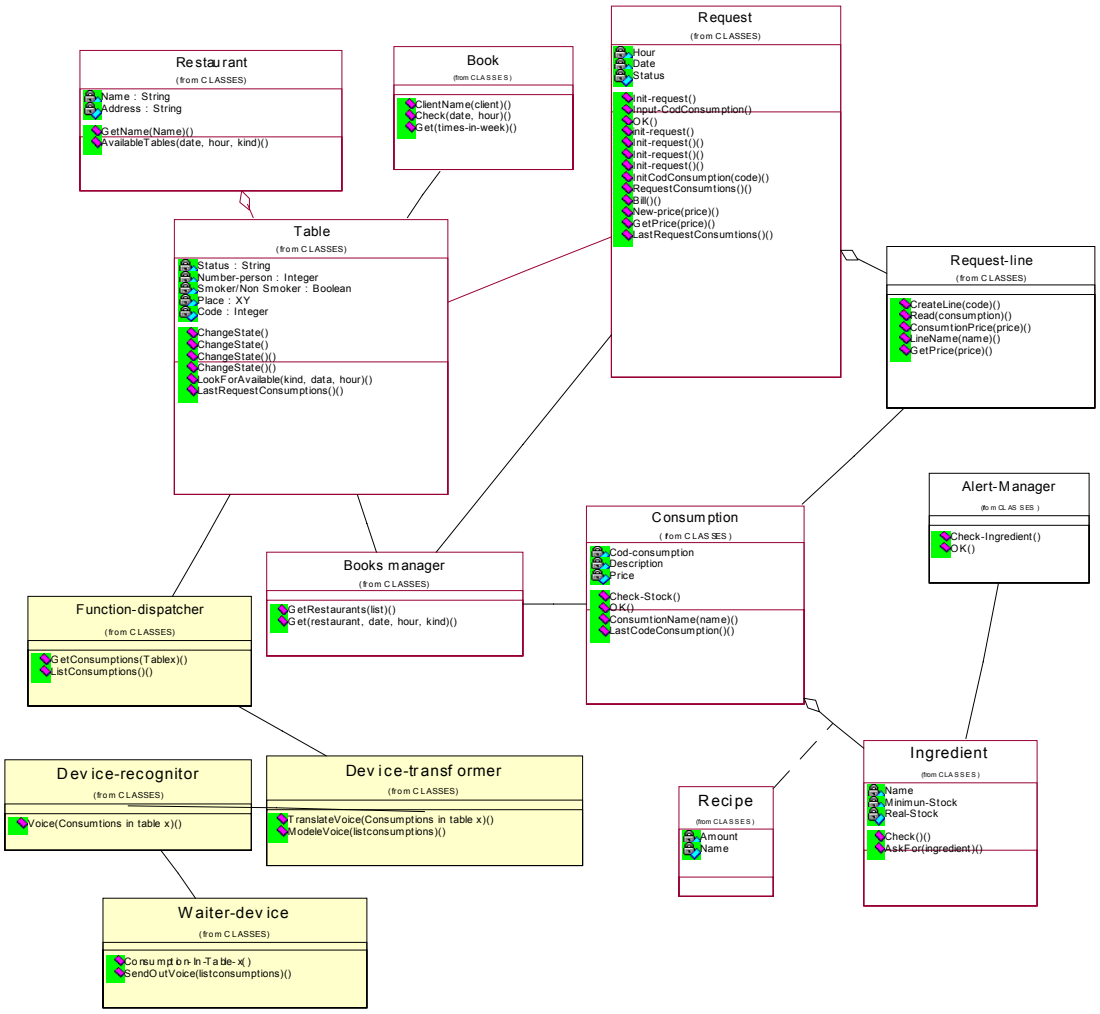


Figure C.14 Class diagram for the first application with Different Access Methods mechanism

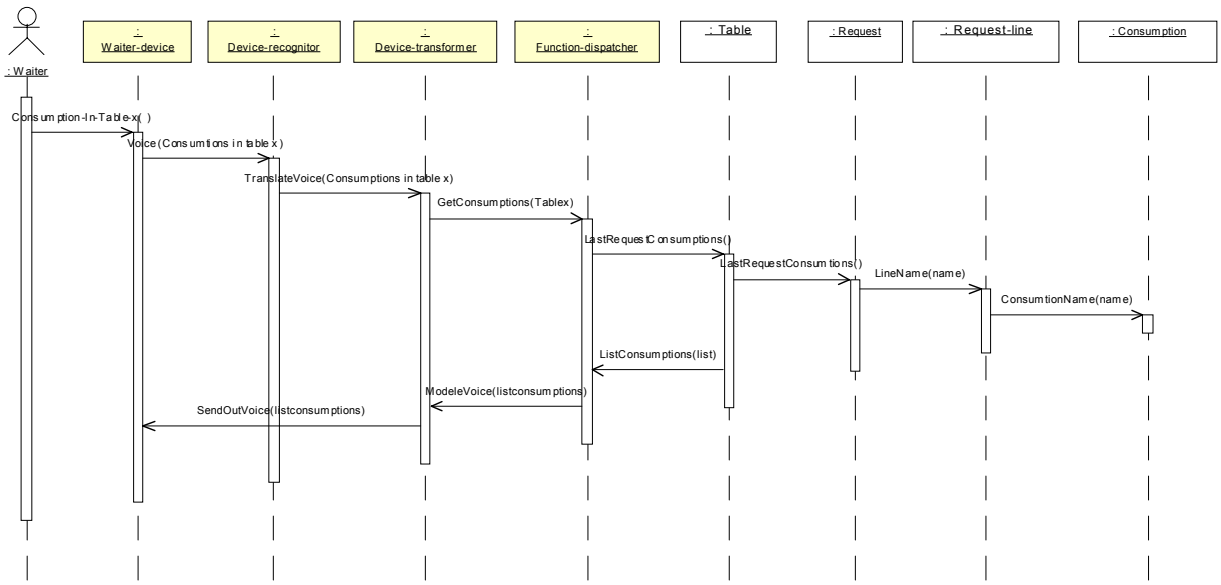
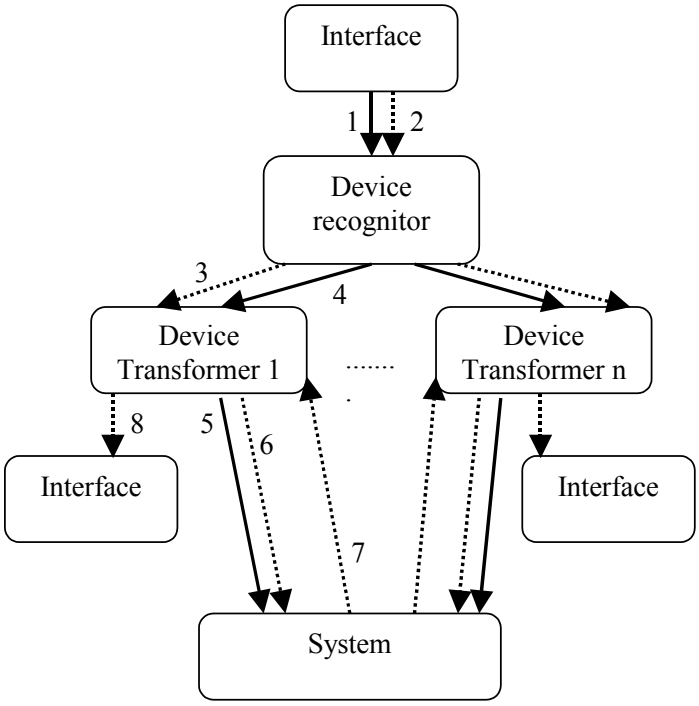


Figure C.15 Sequence diagram for the first application with Different Access Methods mechanism

STEP 3. Abstraction of the design solution for Different Access Methods

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: collects the operation to be performed and any associated data, which it sends to the Device-recogniser (1) (2). Additionally, once the respective functionality has been processed, the interface receives the data to be displayed to the user from the Device-transformer in the format in which the user placed the request (8).
    - Device-recogniser: is a signal format recogniser, which sends the signal to one device or another for interpretation, depending on the type of signal it receives. Additionally, it sends the data and the functionality request to the respective device-transformer (5) (6).
    - Device-transformer: (i) there may be one for each device that the system is able to recognise. If there is one for each device, which would be advisable for reasons of system modularity, each Device-transformer is responsible for converting both the functionality and any data it receives from the Device-recogniser (3) (4) to a general functionality understood by the system. Once the signal has been converted to a functionality and/or data that can be understood by the system, it is all sent to the system for it to perform the respective operation (5) (6). Additionally, once the functionality has been processed in the system, it again receives the response data for the executed functionality (7), which it again translates to the specific signal format in which the user requested the functionality. After translation, it sends the data to the user (8) through the interface.
    - System: it performs the functionality requested by the Device-transformer (i) in the common functionality format (5) (6) and returns the response to the respective device-transformer in the aforesaid common format (7).

C.6 Alerts First Iteration

STEP 1. Design solution without Alerts

Requirement: the waiter starts a table order.

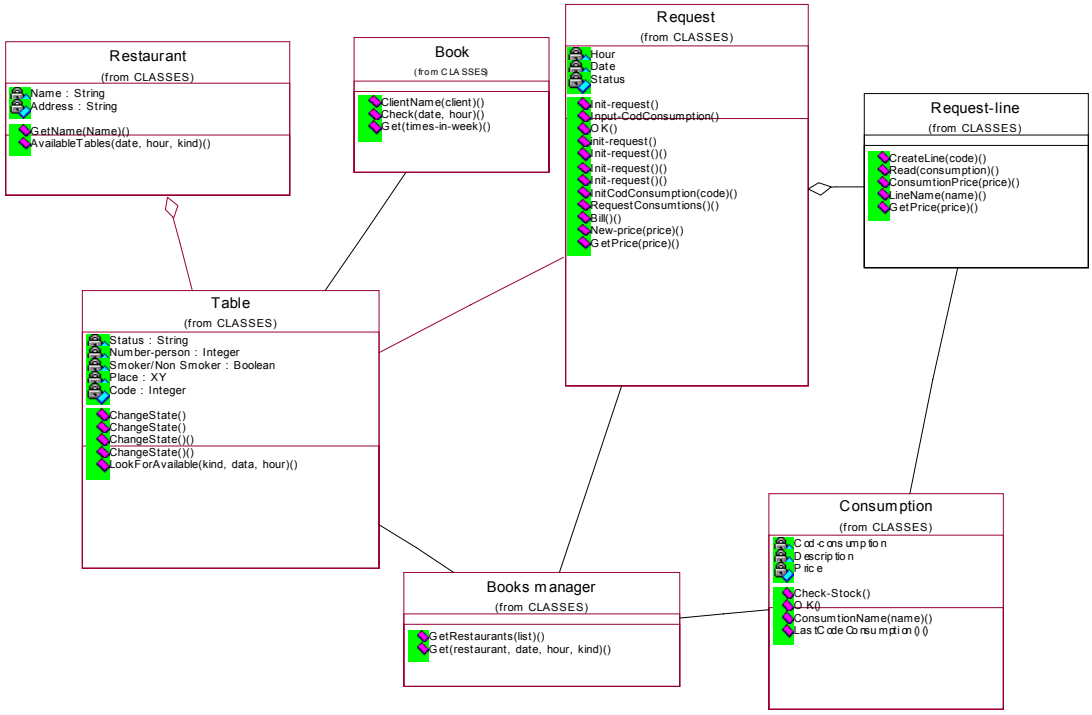


Figure C.16 Class diagram for the first application without Alerts mechanism

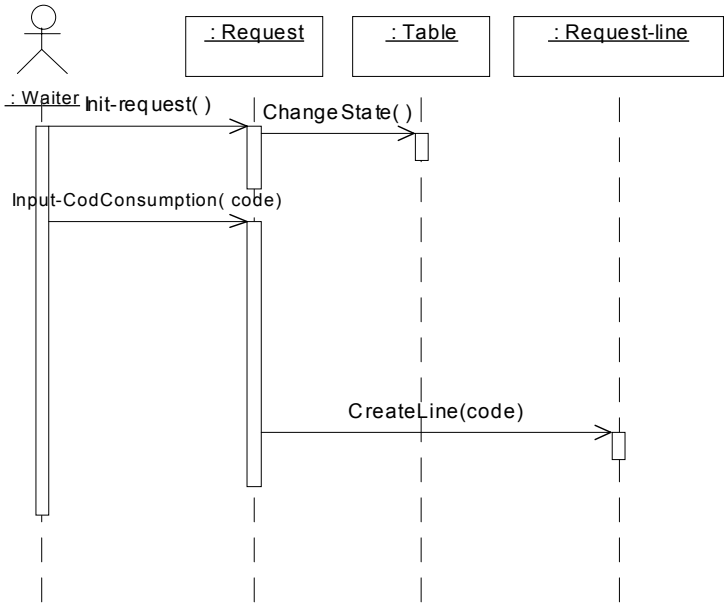


Figure C.17 Sequence diagram for the first application without Alerts mechanism

STEP 2. Design solution with Alerts

Requirement: the foodstuff code cannot be entered until a check has been run of whether there is a stock of all the ingredients for the selected foodstuff.

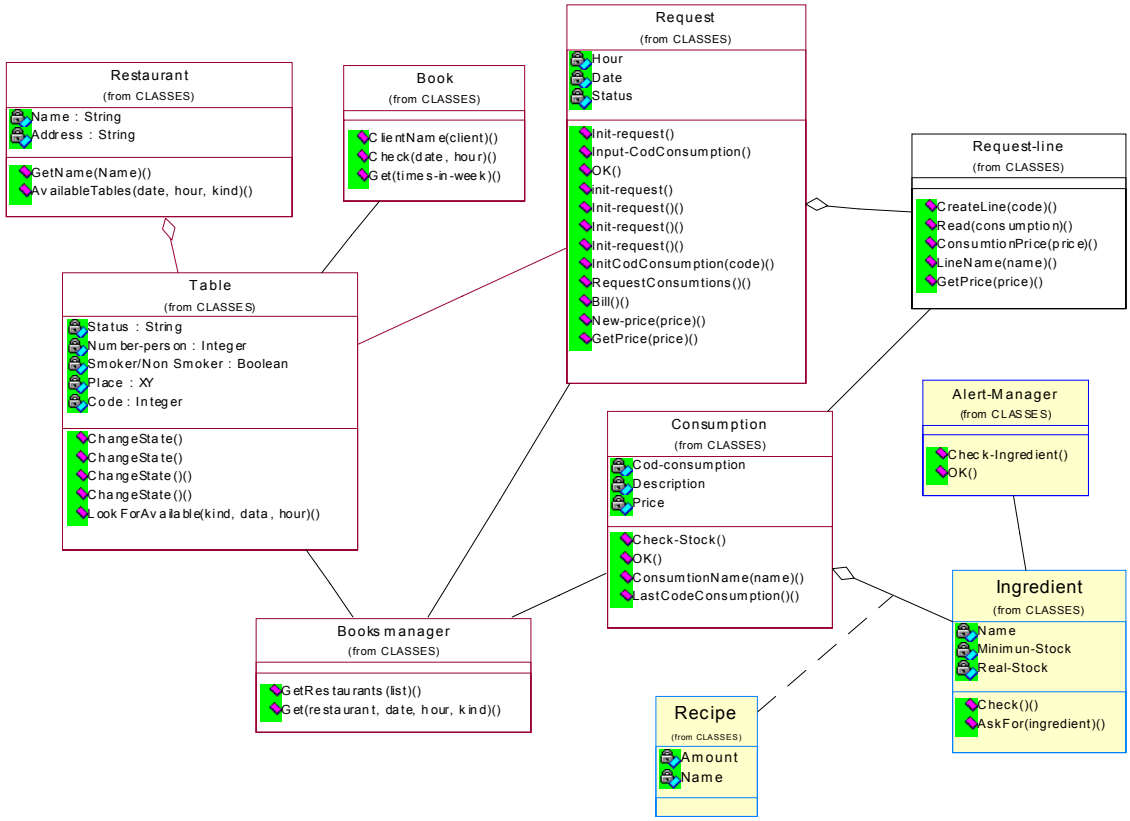


Figure C.18 Class diagram for the first application with Alerts mechanism

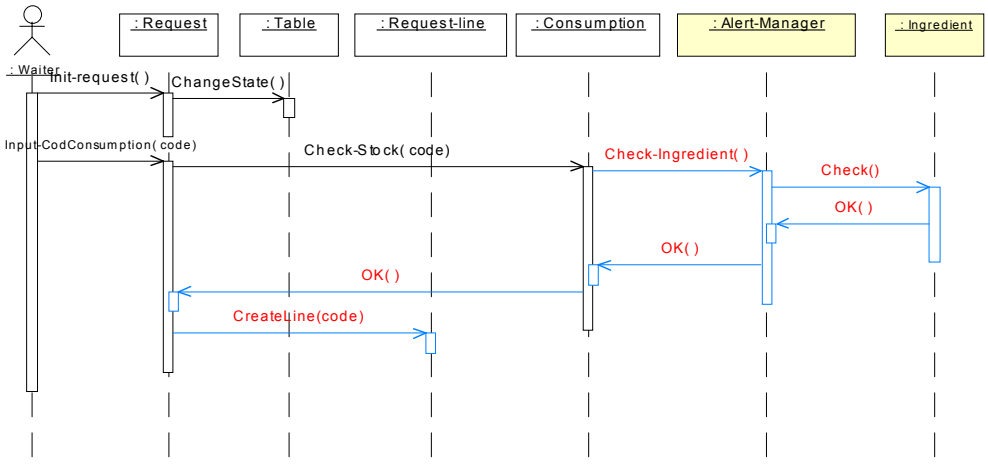
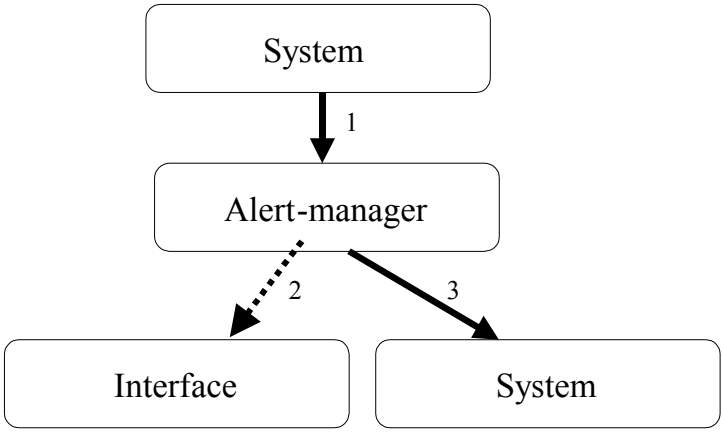


Figure C.19 Sequence diagram for the first application with Alerts mechanism

STEP 3. Abstraction of the design solution for Alerts

- **Solution:**

- Diagram:



- Participants:

  - System: represents the element of the system to be checked in order to identify anything of importance for this element. It is responsible for notifying the Alert-manager to check the state of the element to be checked within the system. (1). Depending on what is to be checked, it also sends the request to the part of the system responsible for running the check (3) and, when the check has been run, sends the respective results (if required) to the interface (2).
  - Alert-manager: represents a component of the system that is capable of receiving a checking order and forwarding this order to the part of the system that is capable of processing it. It receives the checking order from one part of the system (1) and forwards this request to the part of the system concerned (3). Finally, if applicable, it displays any alert information that is of interest to the user (2) to check that one or more system components are working correctly.

C.7 Status Indication First Iteration

STEP 1. Design solution without Status Indication

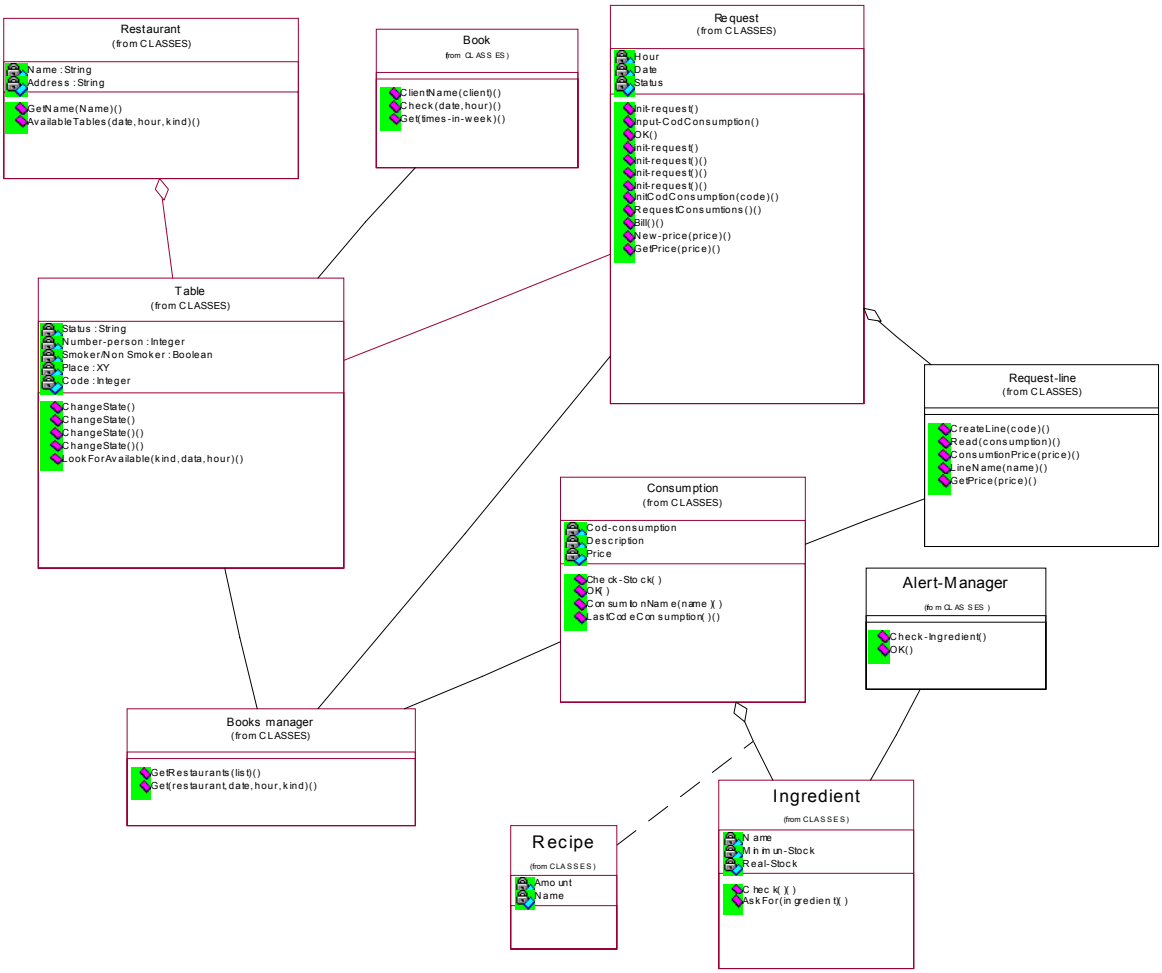


Figure C.20 Class diagram for the first application without Status Indication mechanism

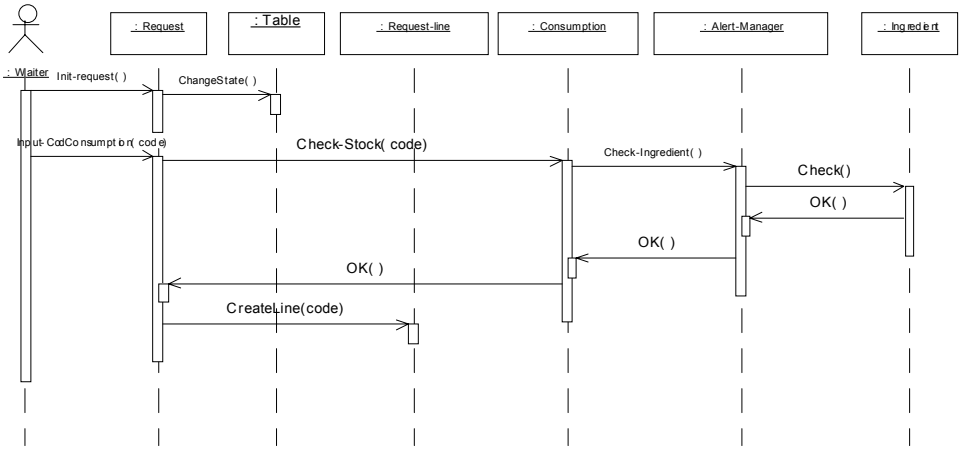


Figure C.21 Sequence diagram for the first application without Status Indication mechanism

As we can see the user is not receiving any information about what the system is doing, which can lead to confusion during system use.

STEP 2. Design solution with Status Indication

Requirement: the user must be informed about what is happening in the system.

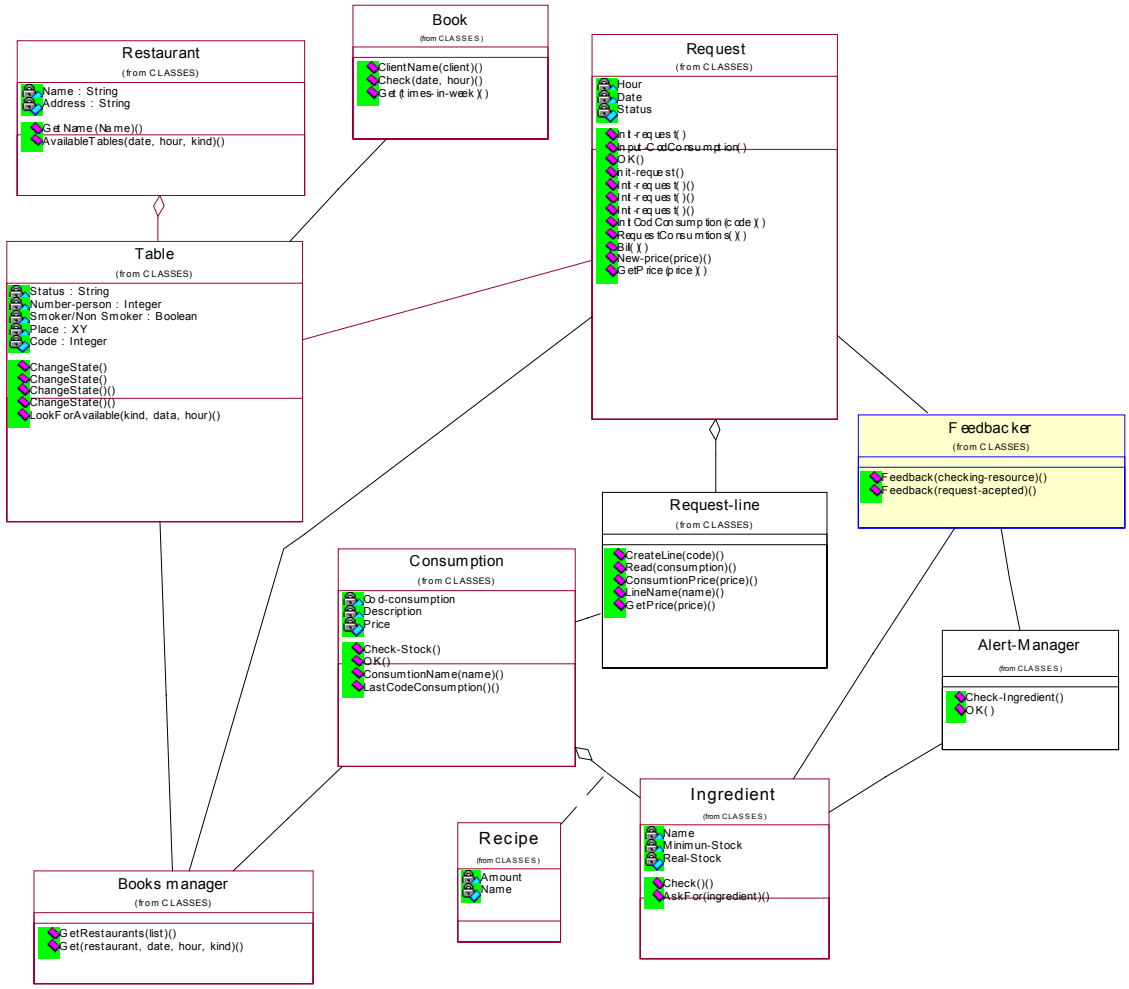


Figure C.22 Class diagram for the first application with Status Indication mechanism

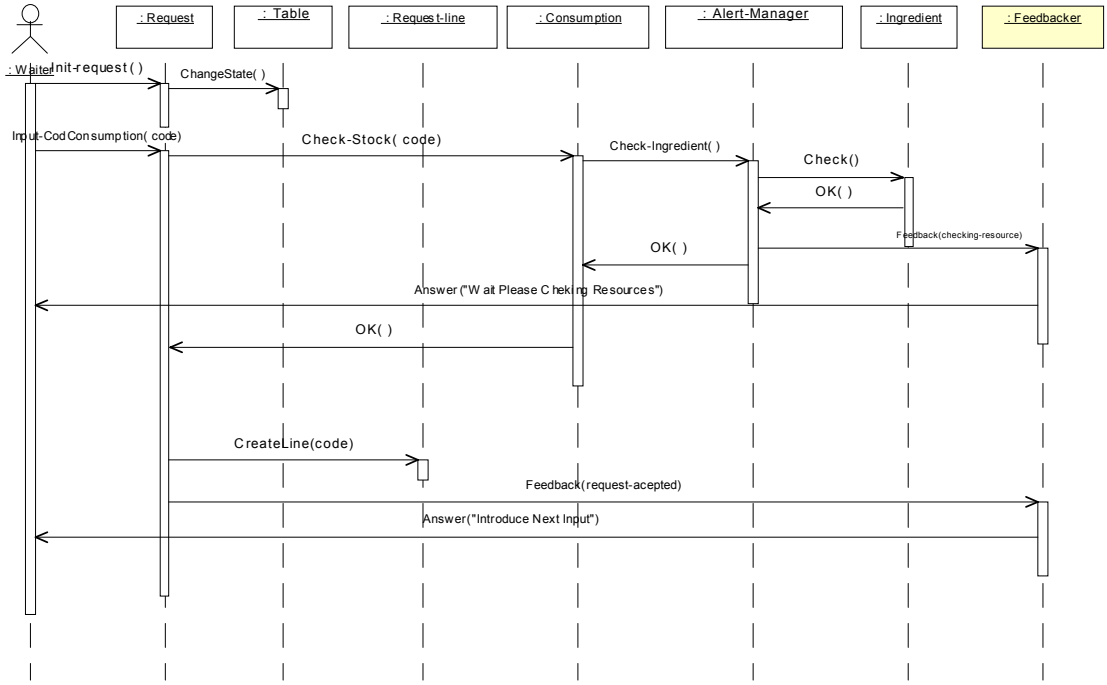
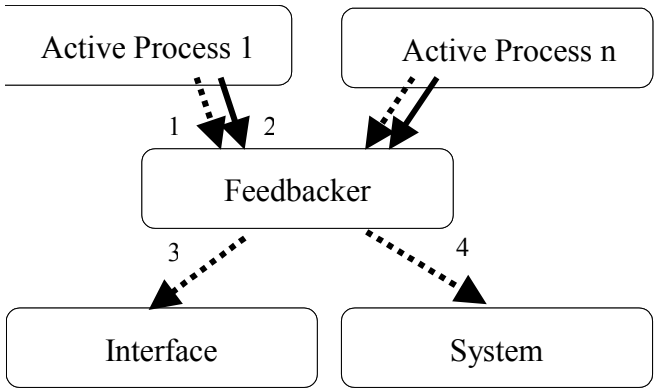


Figure C.23 Sequence diagram for the first application with Status Indication mechanism

STEP 3. Abstraction of the design solution for Status Indication

- **Solution:**
  - Diagram:



  - Participants:
    - Active-process i: this module has been represented more than once, because there may be several processes running simultaneously that request feedback (1) so that it will be each active process that sends the information that it wants to be fed back to Feedbacker (1).
    - Feedbacker: this module receives the request and data (1) (2), which indicates the desired type of feedback and the data to be fed back from each active process. Additionally, it needs to know the recipient of this feedback and will send this feedback either to another part of the system (4) and/or to the interface (3) to inform the user. For some guidelines on how to display this feedback on the interface, for example, how often it should be refreshed or where to place specific

information, see [Welie, 00]. These details should be taken into account in low-level design.

- Interface: it receives the feedback and displays it to user (3).
- System: this component is optional and represents other parts of the system that must be informed of the feedback (4).

C.8 History Logging First Iteration

STEP 1. Design solution without History Logging

Requirement: the user starts an order request

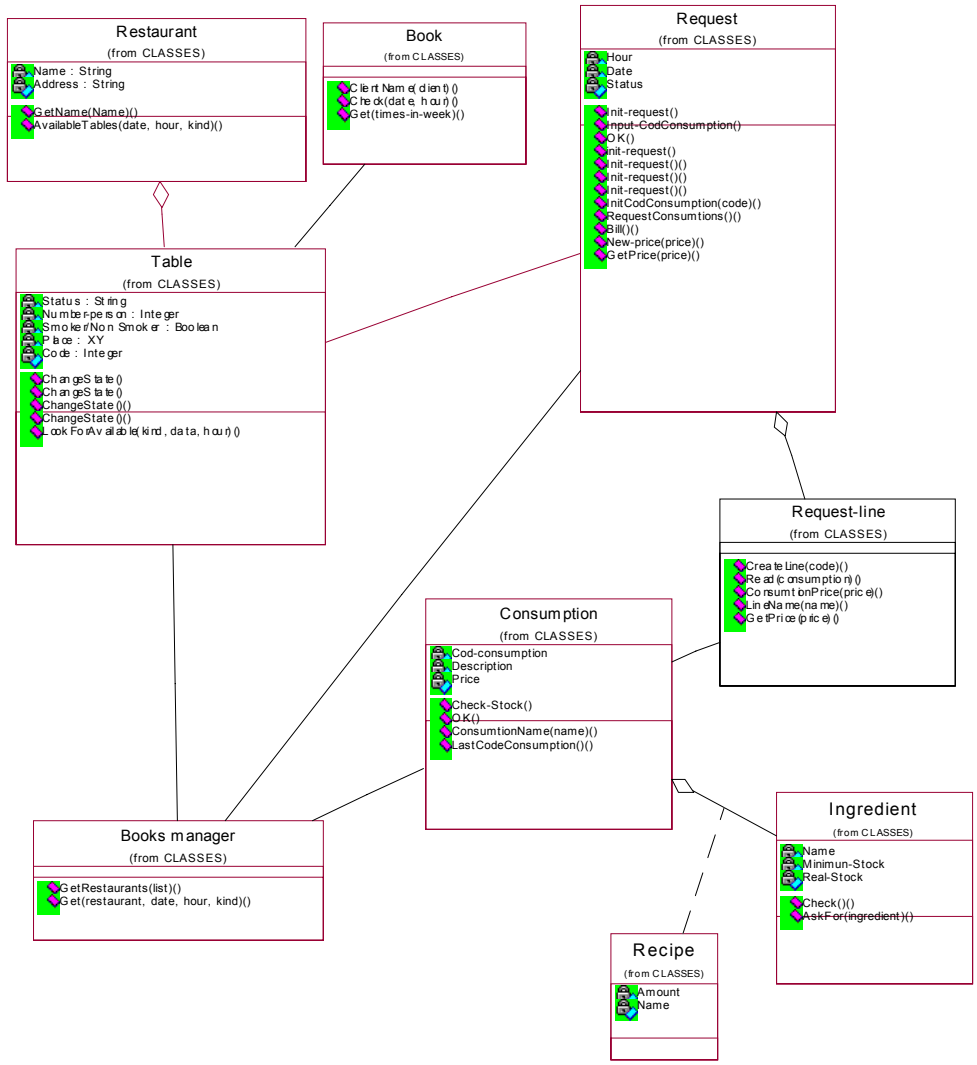


Figure C.24 Class diagram for the first application without History Logging mechanism

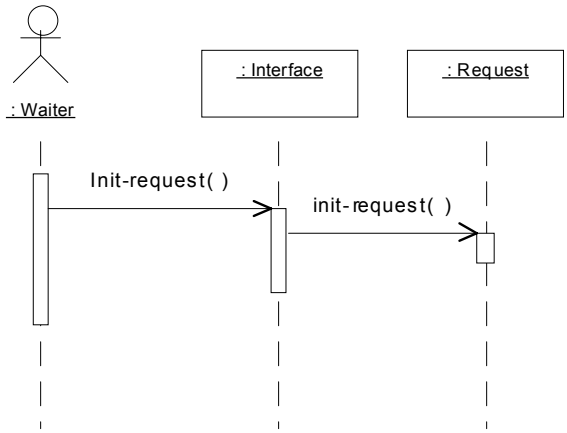


Figure C.25 Sequence diagram for the first application without History Logging mechanism

STEP 2. Design solution with History Logging

Requirement: When an order request is started, the system records that the user has opened an order.

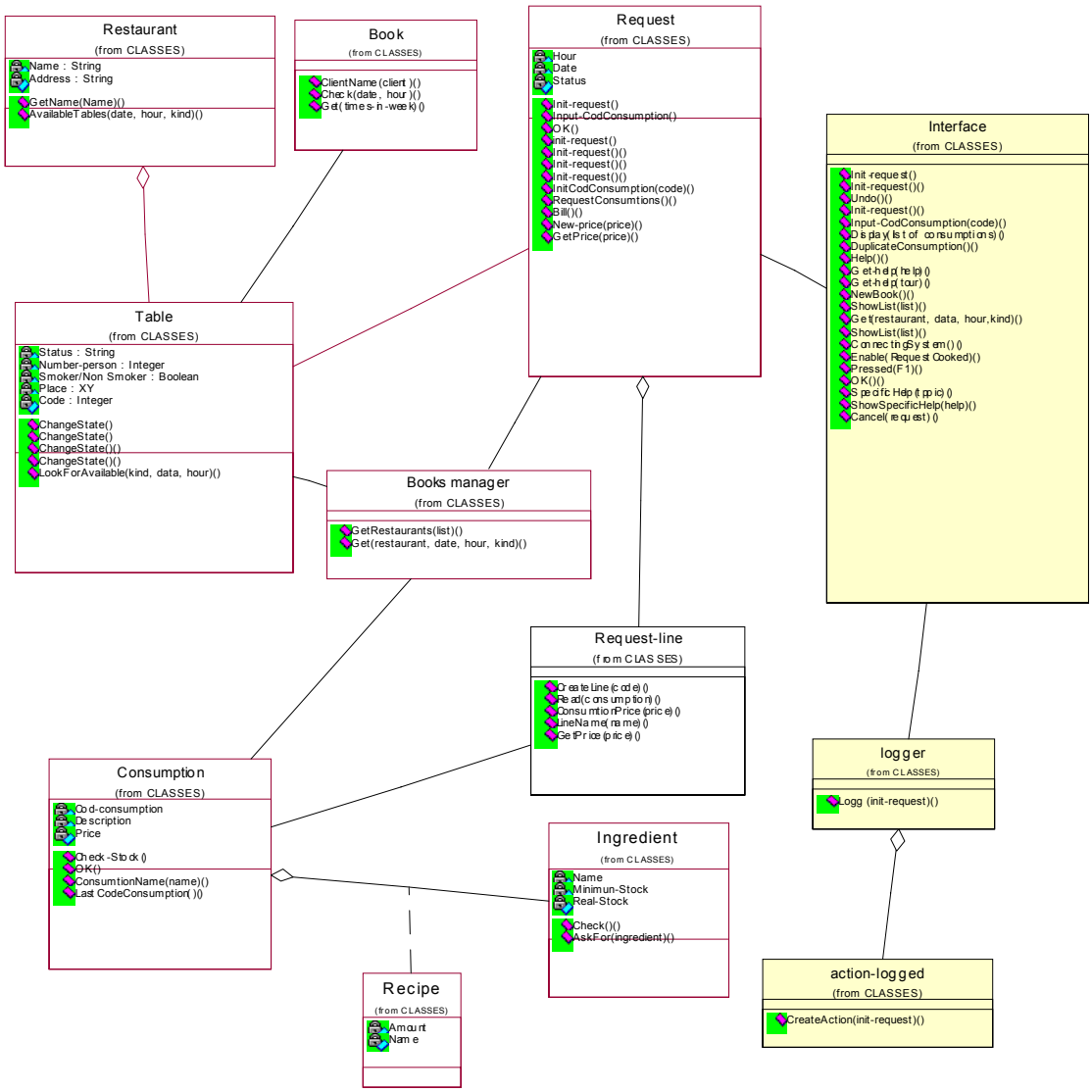


Figure C.26 Class diagram for the first application with History Logging mechanism

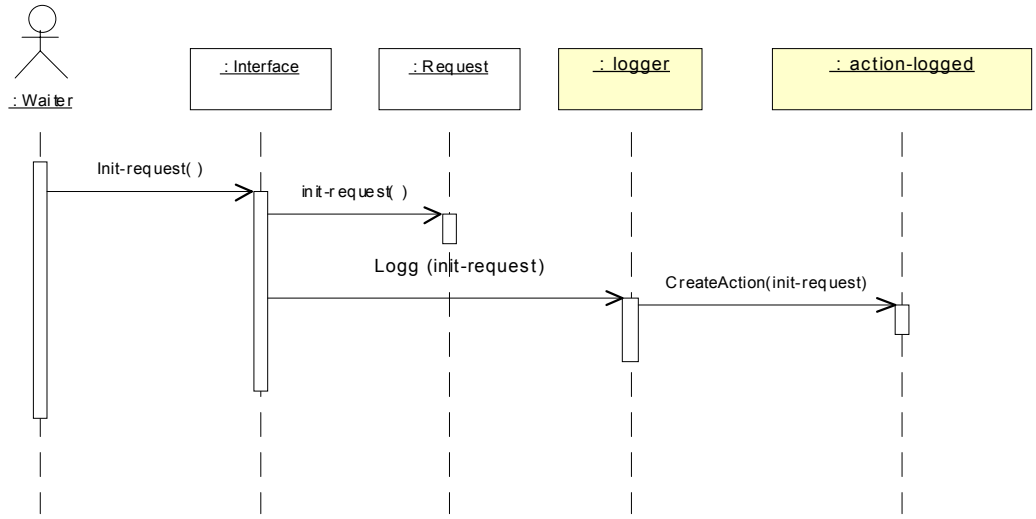
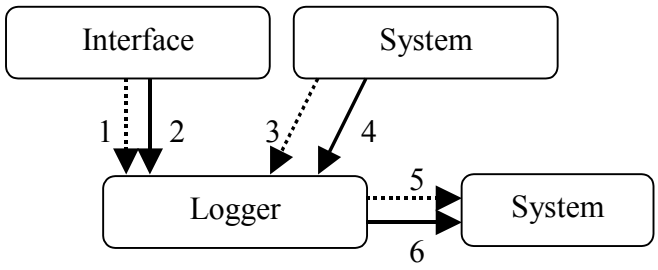


Figure C.27 Sequence diagram for the first application with History Logging mechanism

STEP 3. Abstraction of the design solution for History Logging

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it receives the request to execute an operation in the system, which may contain both the operation and data (1) (2). As we will see later, this execution request can also come from the actual system (3) (4).
    - Logger: this module receives the actions and the data requested by the user or from another part of the system (1) (2) (3) (4) and stores the logged action and data either internally or in another part of the system, in which case it will have to send this action and data to the system (5) (6) to be processed in the respective part of the system.
    - System: this module sends the functions and data that are executed in the system to the logger (3) (4), and also, optionally, if the logger does not store the logged actions internally, sends the information to the part of the system that manages these actions (5) (6).

C.9 Undo First Iteration

STEP 1. Design solution without Undo

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

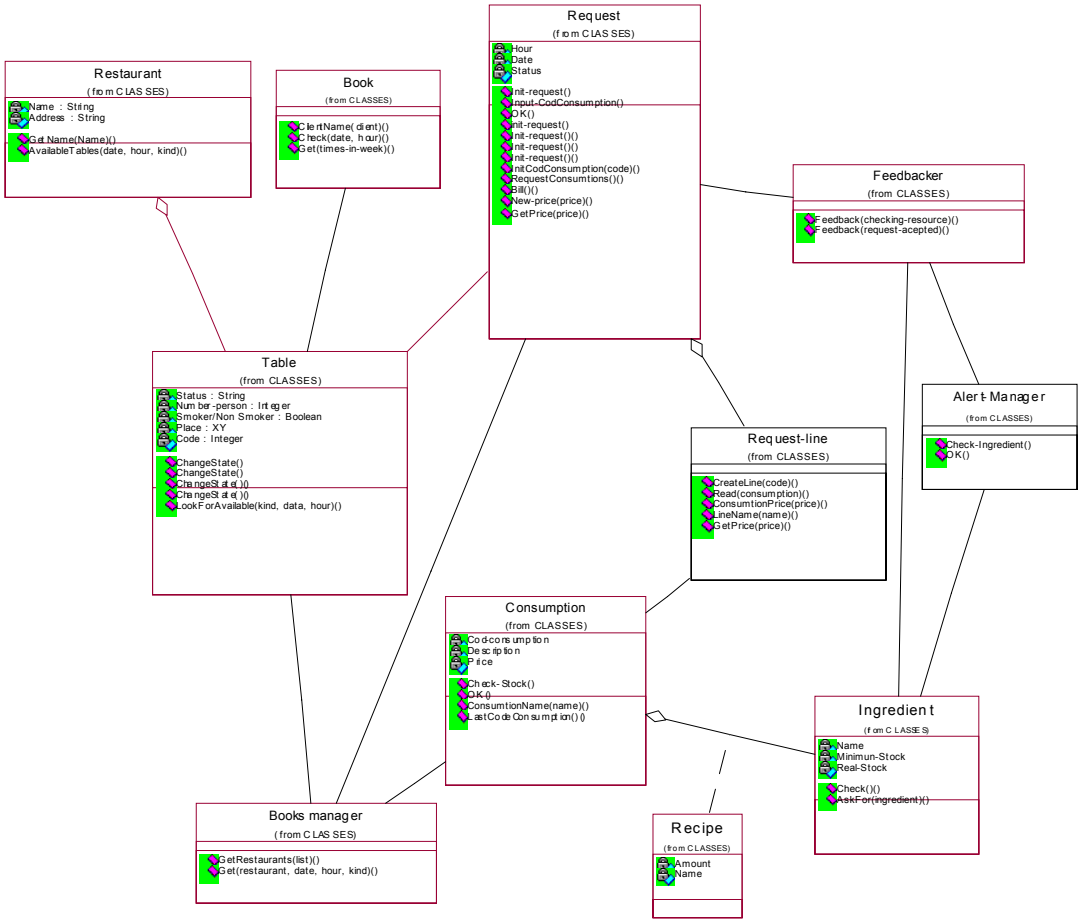


Figure C.28 Class diagram for the first application without Undo mechanism

STEP 2. Design solution with Undo

Requirement: the user can push the undo button

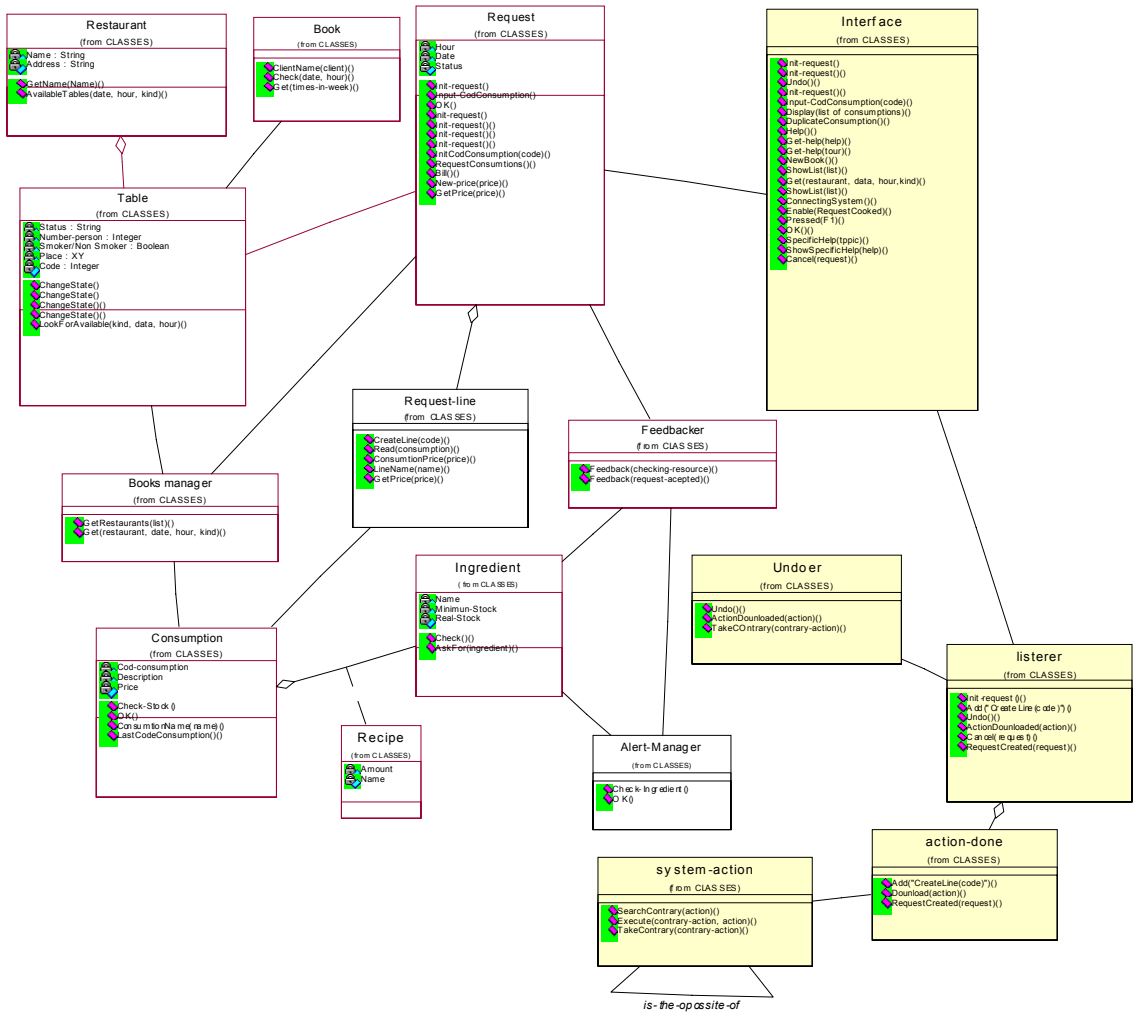


Figure C.29 Class diagram for the first application with Undo mechanism

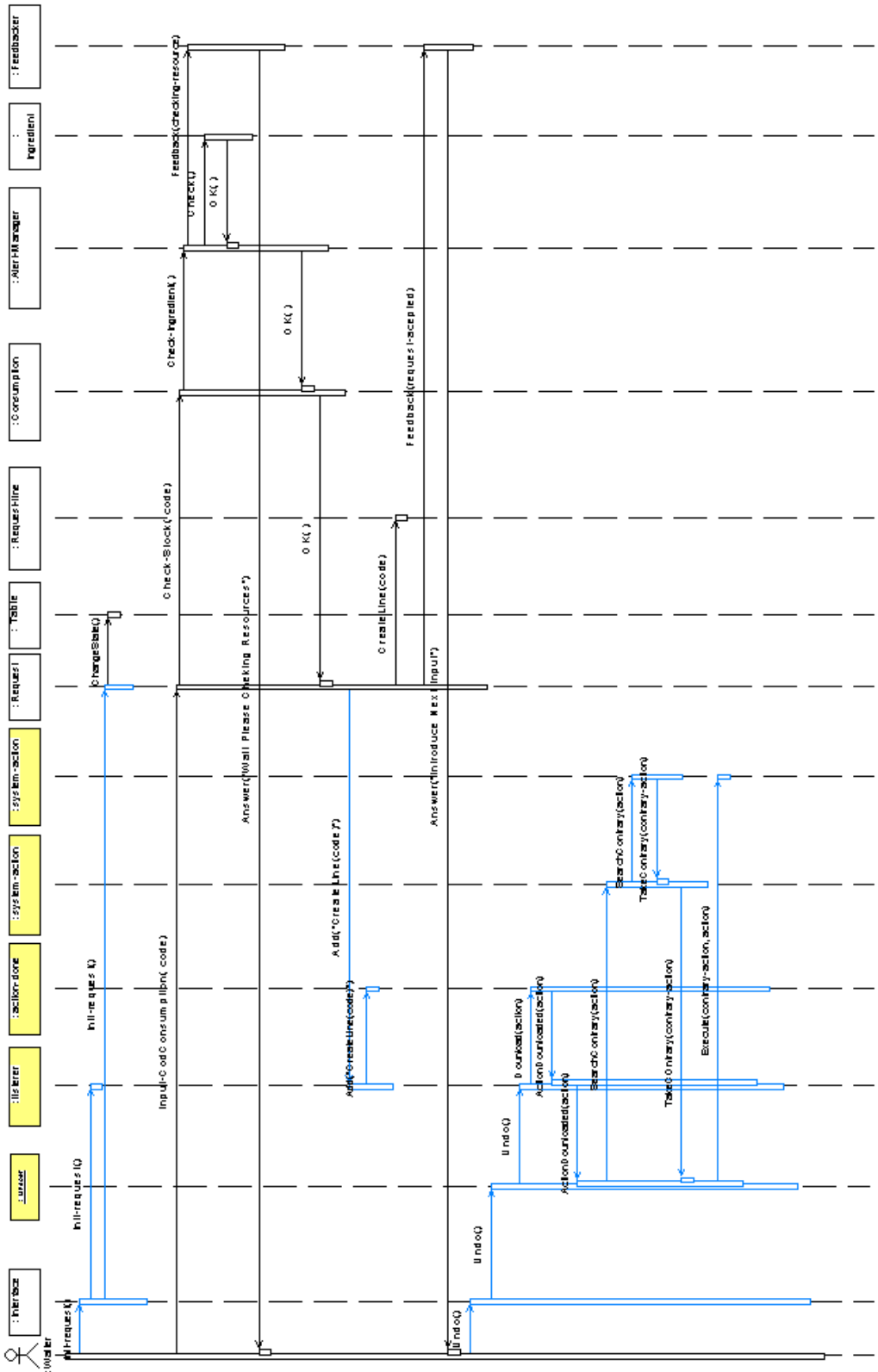
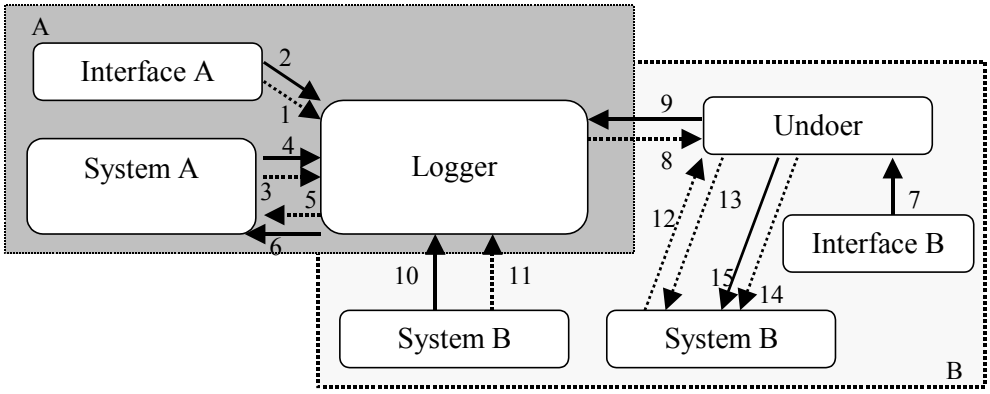


Figure C.30 Sequence diagram for the first application with Undo mechanism

STEP 3. Abstraction of the design solution for Undo

- **Solution:**
  - Diagram:



  - Participants: This pattern has two clearly separate parts. These parts have been labelled in the illustration as A and B, respectively. Part A collects the actions performed in the system (the number of actions to be stored will have to be specified when the system is developed) so that they can be later undone. Part B manages the respective undo.
    - InterfaceA: receives the request to execute an operation in the system, which may contain both the operation and data (1) (2). As we will see later, this execution request can also come from the actual system (3) (4).
    - SystemA: this module sends the functions and data executed in the system to the logger (3) (4) and also, optionally, if the logger does not store the actions internally, will send the information to the part of the system that manages these actions (5) (6).
    - Logger: this module receives the actions and the data requested by the user or from another part of the system (1) (2) (3) (4) and stores the logged action and data either internally or in another part of the system, in which case it will have to send this action and data to the system (5) (6) to be processed by the respective part of the system. Logger receives the undo request from Undoer (9) and, if the logged actions are stored in the logger, it then sends them one by one to Undoer (8). If they are not stored in the logger, it will receive both the data and the operation to be undone from another part of the system that we have named System B through (11) and (10), respectively.
    - Interface B: receives the undo request and sends it to Undoer through (7).
    - Undoer: sends the undo request to logger (9) and also sends each of the actions to be undone that it receives from logger to System B (13), as well as receiving the opposite operation to the one performed from System B (12). When it knows which opposite operation is to be performed, it sends the operation to System B along with the data associated with the operation in question through (14) and (15).
    - System B: it will search the system for both the action performed and the data associated with this operation (10) (11) if the data are not stored internally in the logger. It receives the actions to be undone (13) and provides the opposite operation (12) (for which purpose it will have to store what the opposite is for each action, see implementation section, for example). The opposite action and the respective data will be sent to the respective part of the system (15) and (14).

C.10 Form or Field Validation First Iteration

STEP 1. Design solution without Form or Field Validation

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

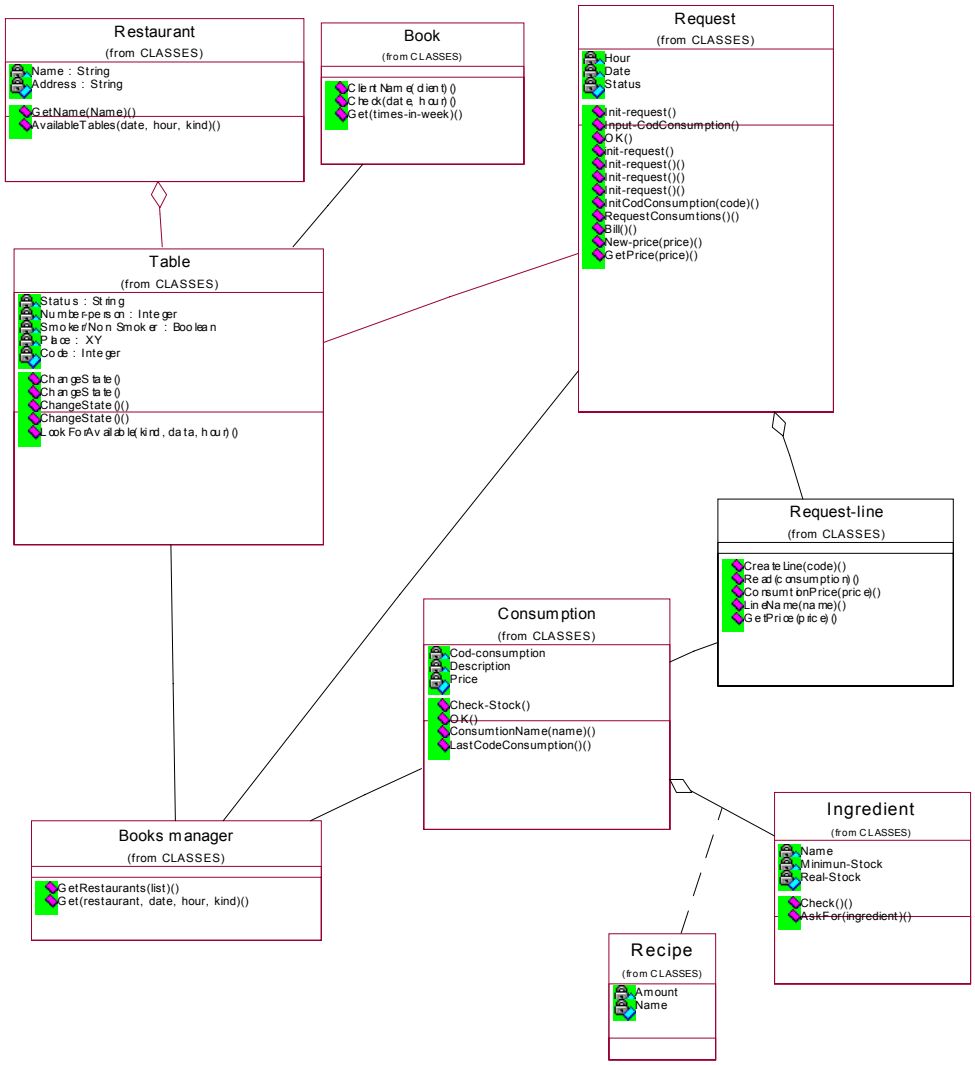


Figure C.31 Class diagram for the first application without Form or Field Validation mechanism

STEP 2. Design solution with Form or Field Validation

Requirement: The system should validate the foodstuff code when it has been entered by the waiter and before it is copied to the order.

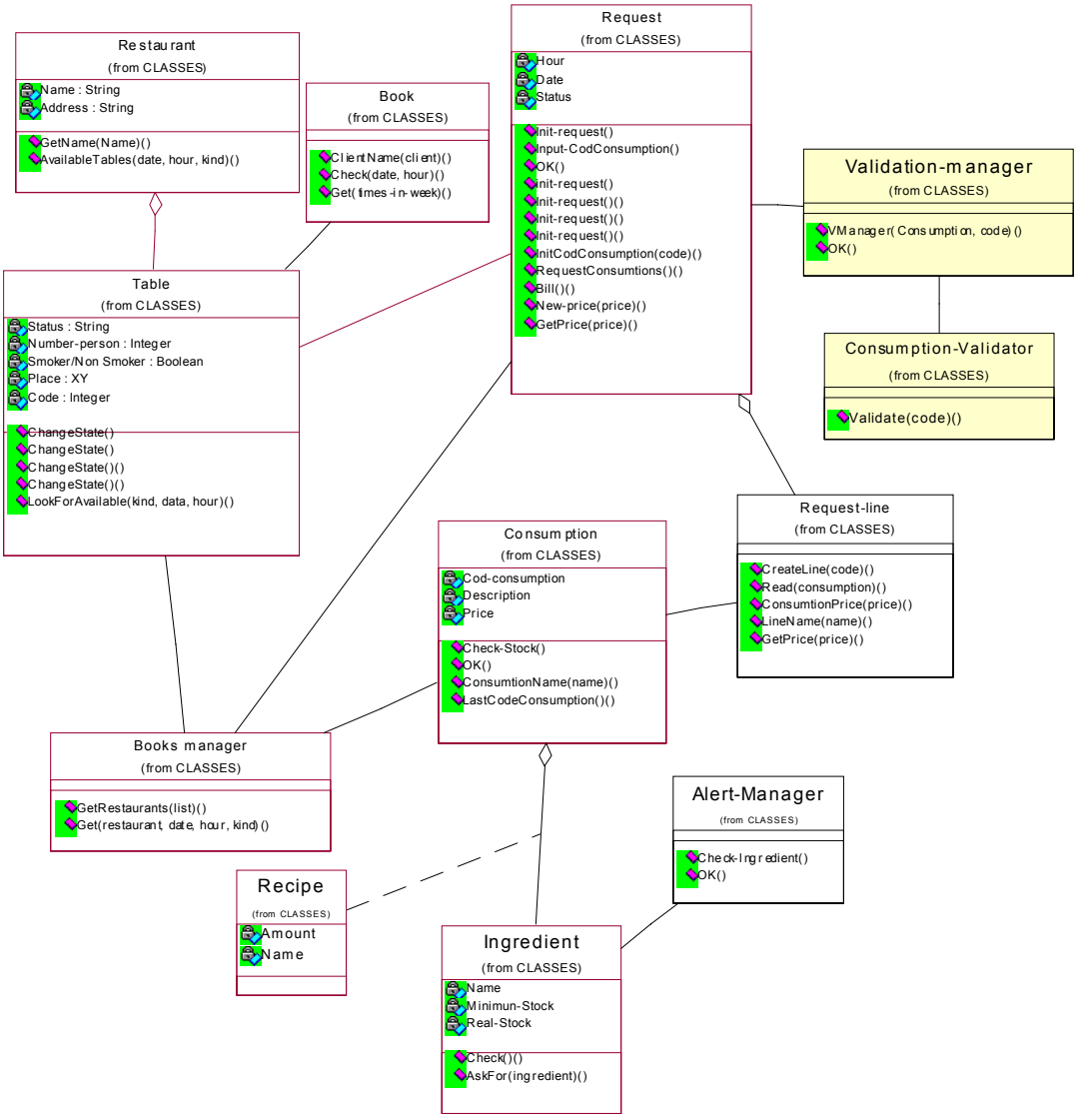


Figure C.32 Class diagram for the first application with Form or Field Validation mechanism

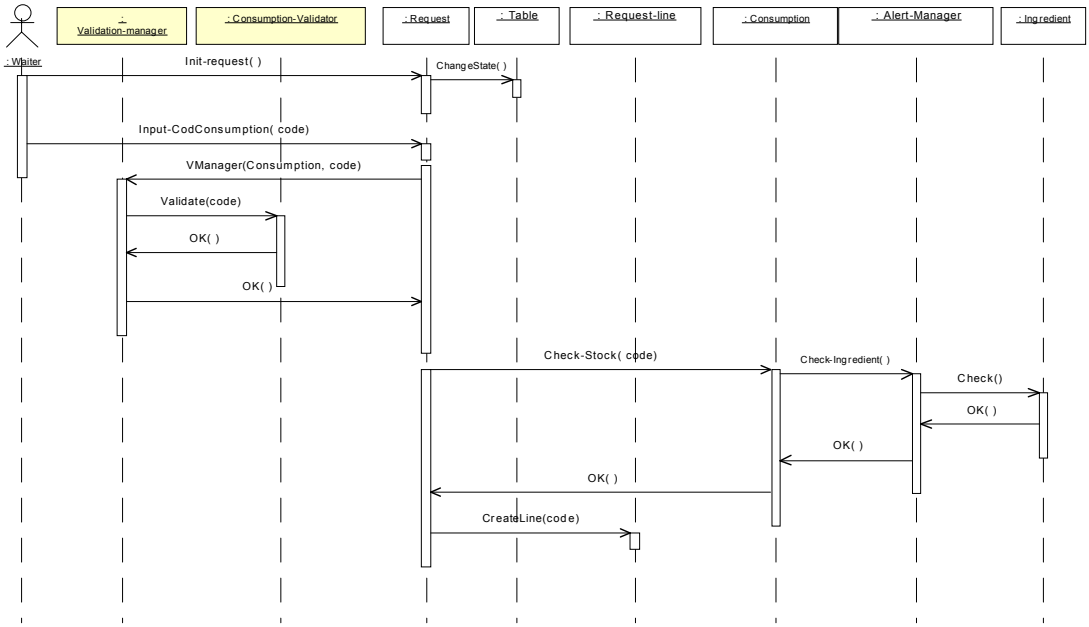
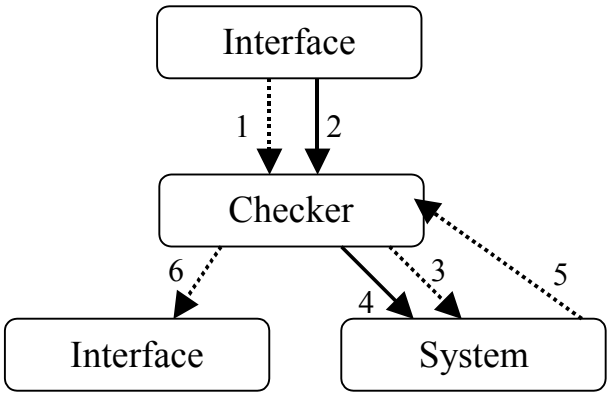


Figure C.33 Class diagram for the first application with Form or Field Validation mechanism

STEP 3. Abstraction of the design solution for Form or Field Validation

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends a data set (1) and the function requested by the user (2) to Checker for validation. Additionally, after data validation, it will receive error data or OK from the Checker to be displayed to the user if the system is to be designed this way (6).
    - Checker: it collects an operation requested by the user through the interface (2) as well as a data set (1). This module can be designed to validate the data or to send the data to another system component for validation (3) (4). In the latter case, it also receives the result of the validation (OK or error) (5) and, in any case, will send the result of the validation to the user if so required (6).
    - System: this component will be optional and will only exist if the Checker is not capable of validating the data. If necessary, it receives both the function and the associated data for validation from Checker (3) (4) and, after validation, returns the result of the validation to Checker.

C.11 Provision of Views First Iteration

STEP 1. Design solution without Provision of Views

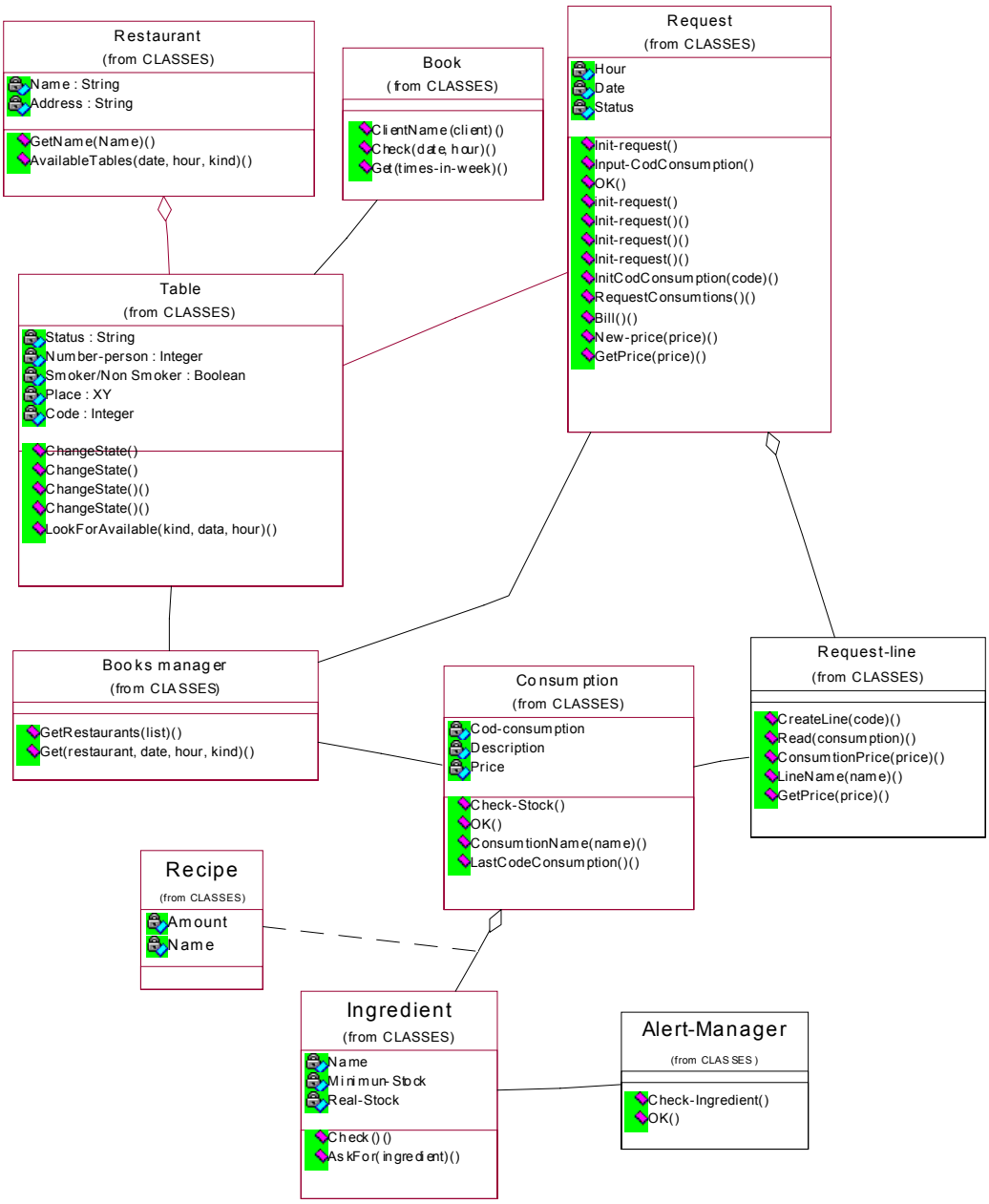


Figure C.34 Class diagram for the first application without Provision of Views mechanism

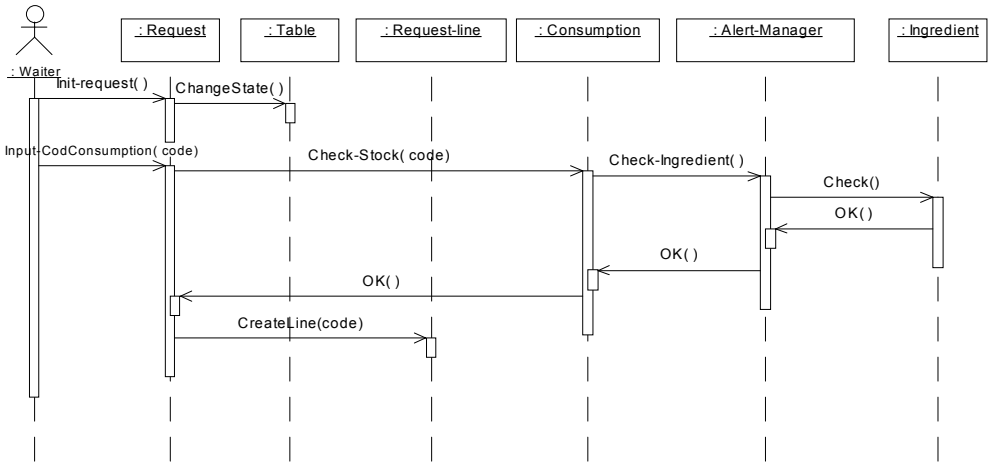


Figure C.35 Sequence diagram for the first application without Provision of Views mechanism

STEP 2. Design solution with Provision of Views

Requirement: The system should be able to provide the customer with the list of things ordered so far at any time while the order is being placed.

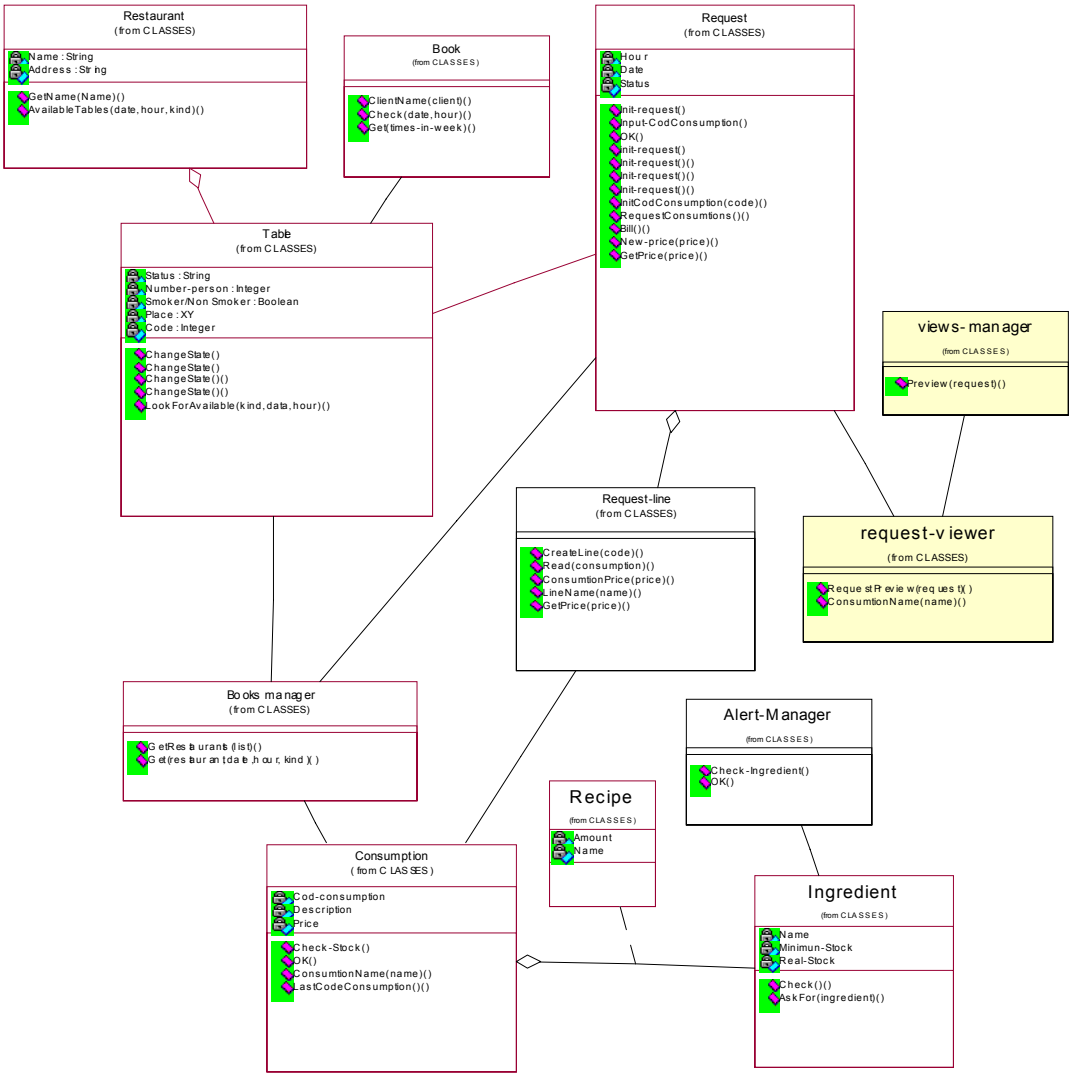


Figure C.36 Class diagram for the first application with Provision of Views mechanism

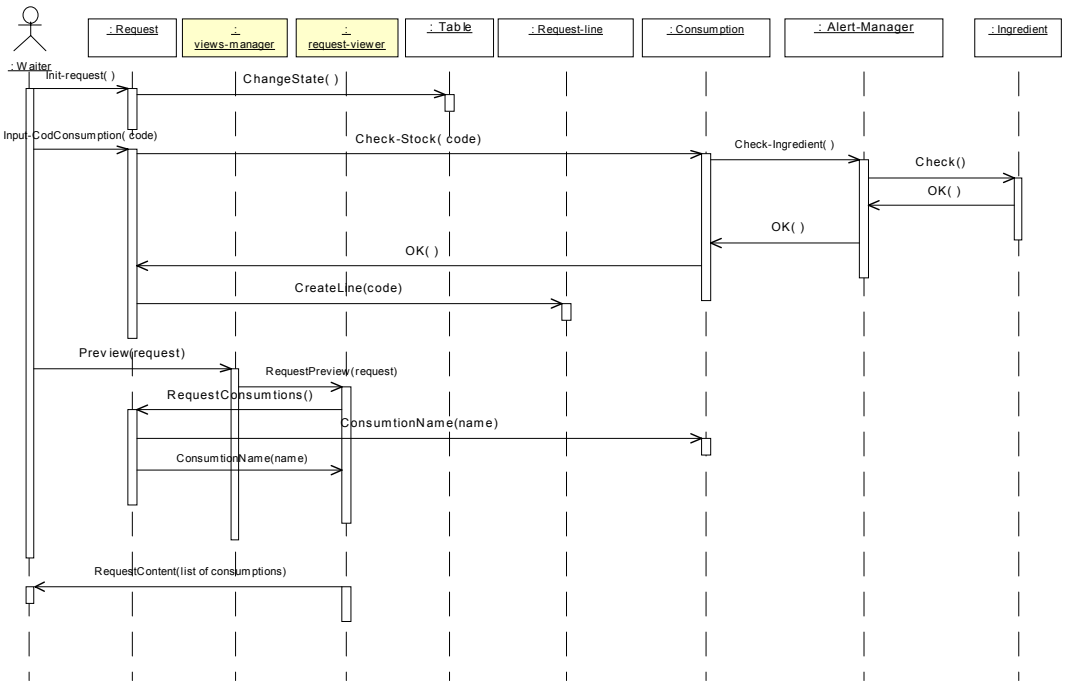
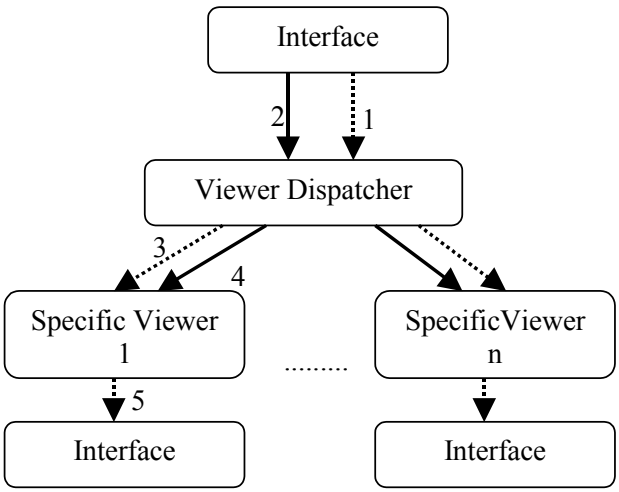


Figure C.37 Sequence diagram for the first application with Provision of Views mechanism

STEP 3. Abstraction of the design solution for Provision of Views

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends the data received (1) and the specific function requested by the user (2) to viewer-dispatcher. Additionally, when the data have been transferred to the specific viewer that knows how to interpret them, they are displayed by the interface (5). For information about how to present some views in the interface, see [Welie, 00]
    - Viewer Dispatcher: it receives the data (1) and the requested function (2) and, depending on this information, decides which viewer should interpret the operation and data. These (3) and (4) are sent to the respective Specific Viewer.

- Specific Viewer i: it receives a request (4) and data to be viewed (3), which it interprets as befits the viewer in question, sending them to the interface (5).

C.12 Workflow Model First Iteration

7.1.1.1.1 STEP 1.1. Design solution without Workflow Model First Iteration

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

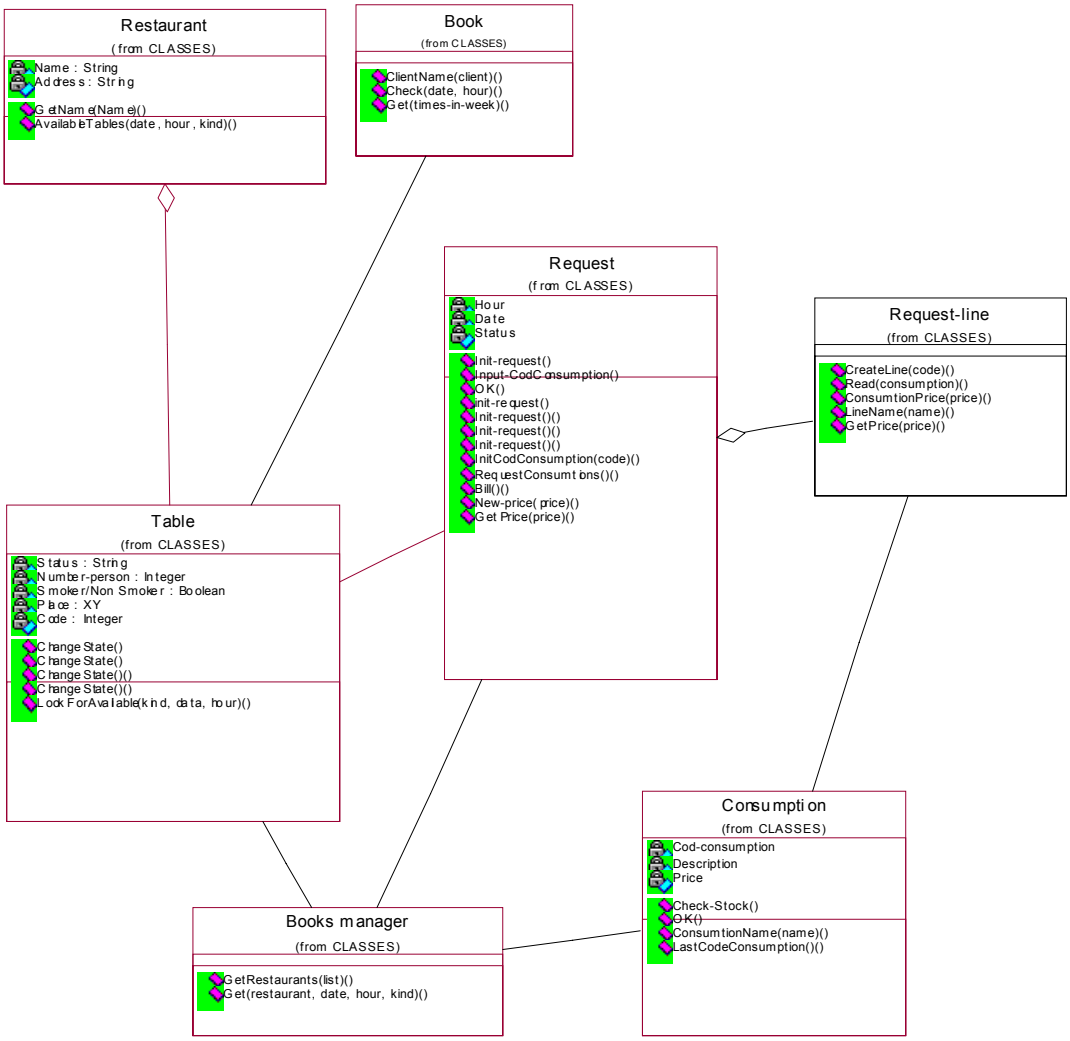


Figure C.38 Class diagram for the first application without Workflow Model mechanism

STEP 2. Design solution with Workflow Model

Requirement: when the cook connects to the system, the only enabled function will be enter as cooked when he has finished cooking an order.

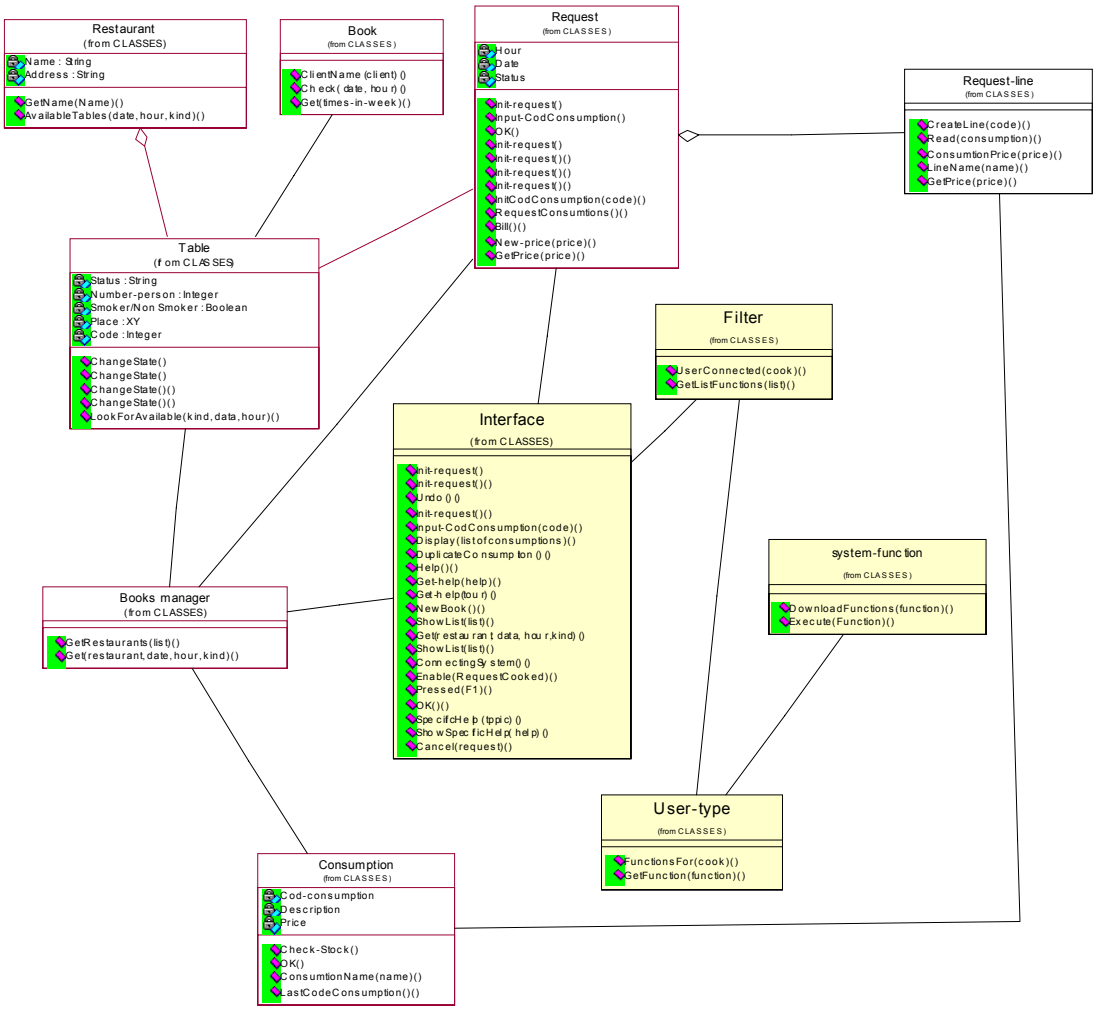


Figure C.39 Class diagram for the first application with Workflow Model mechanism

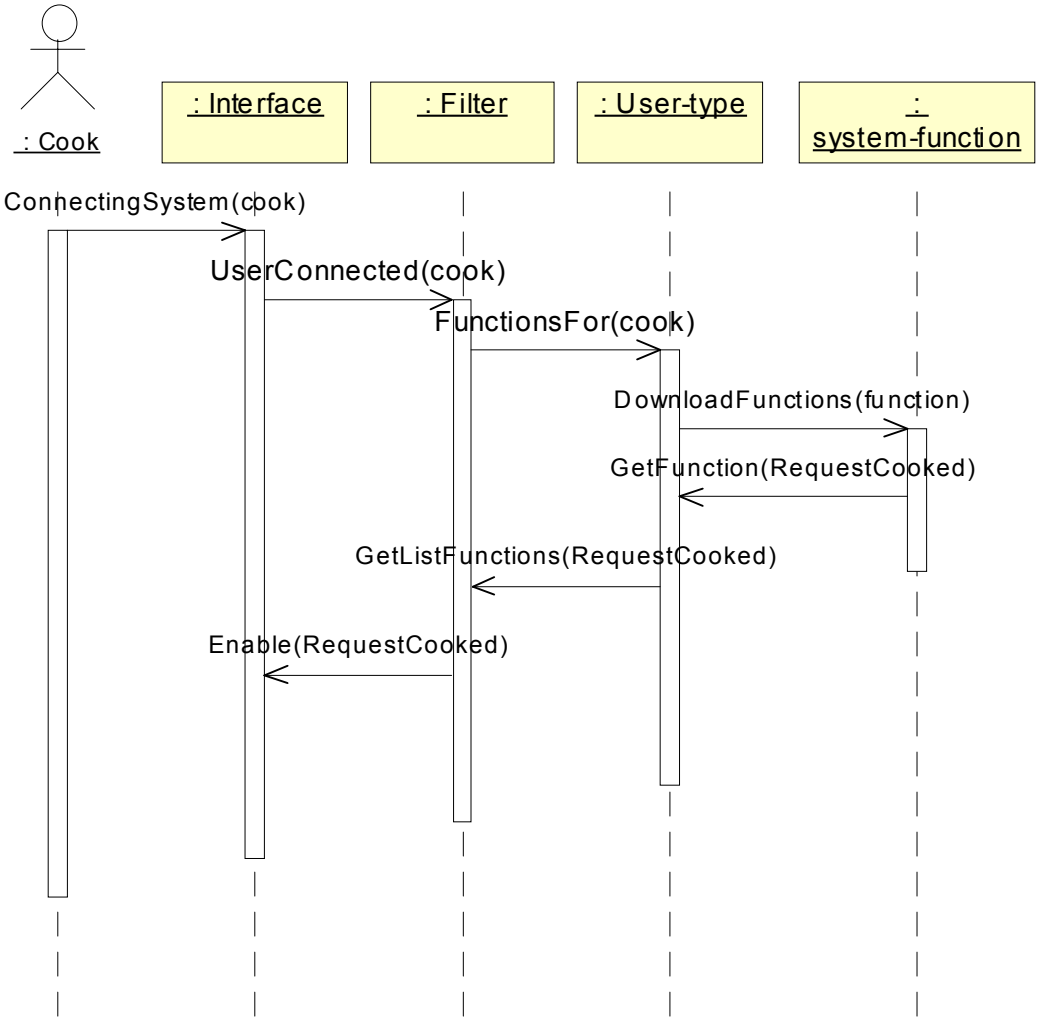
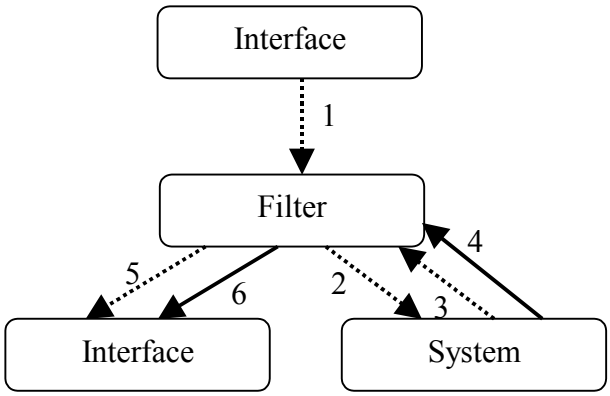


Figure C.40 Sequence diagram for the first application with Workflow Model mechanism

STEP 3. Abstraction of the design solution for Workflow Model

- **Solution:**
  - Diagram:



  - Participants:

- Interface: it sends the data related to the user who is trying to access the system (1) to the system. Additionally, the interface receives the data and operations (5) (6) that make up the interface for the user in question from Filter.
- Filter: it receives the type of user who wants to connect to the system (1) from the interface. Additionally, if it does not store all the functionality that should be associated with each user internally, it sends the data about the user in question to another system component (2) and receives both the data (3) and the operations (4) to which this user should have access from this component. When it has this information, it then passes it on to the interface for proper display (5) (6).
- System: this component is optional and will only exist if the Filter is not capable of storing the functionalities associated with each system user internally. Accordingly, this component receives the data on the user type who has connected from Filter (2) and returns both the data and operations that this user type can access from the interface (5) (6) to Filter.

C.13 User Profiler First Iteration

STEP 1. Design solution without User Profiler

Requirement: the client can ask for the bill.

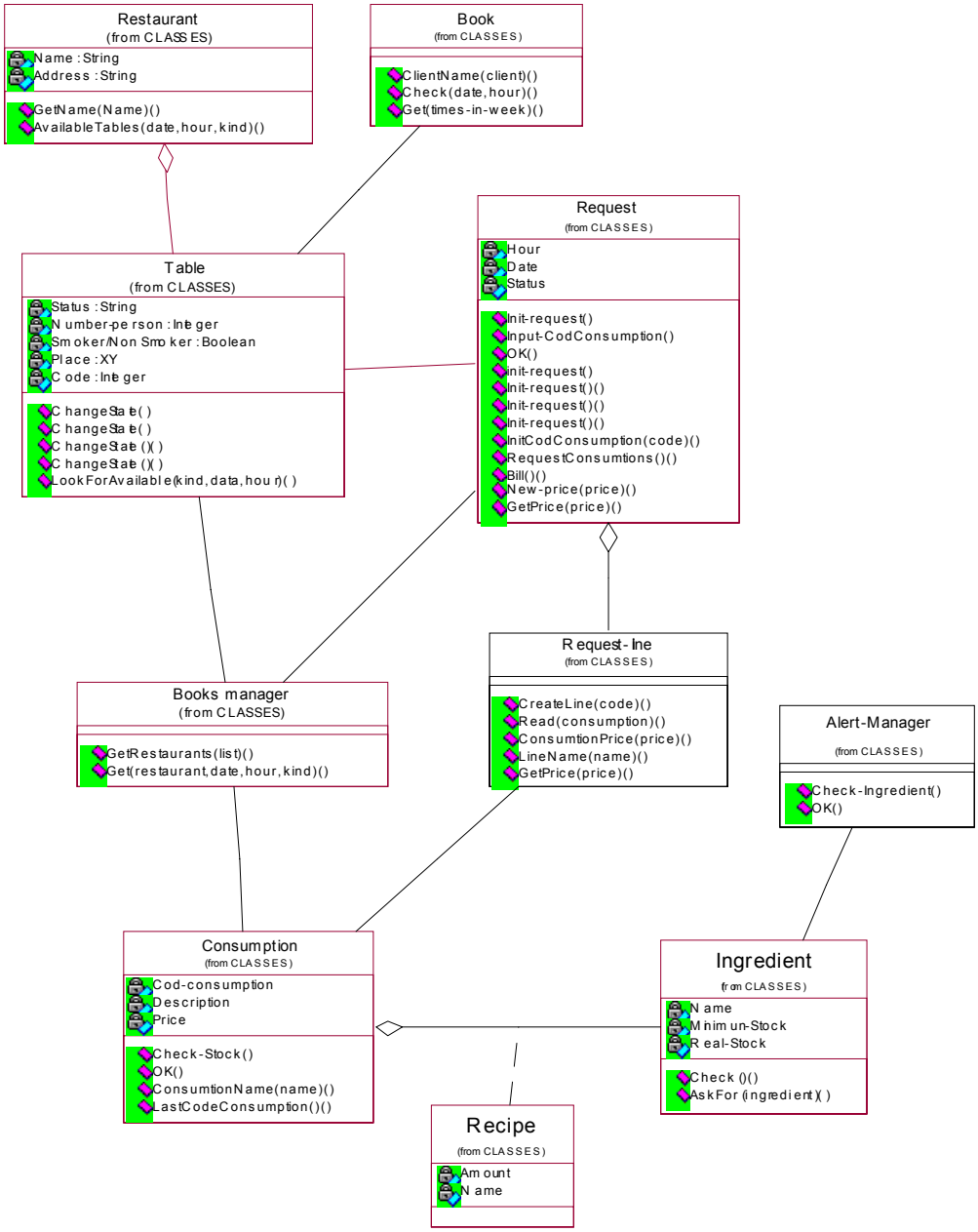


Figure C.41 Class diagram for the first application without User Profiler mechanism

STEP 2. Design solution with User Profiler

Requirement: The system should be able to identify the customer who is making the order at a given table so that he can be given personalised treatment depending what type of customer it is.

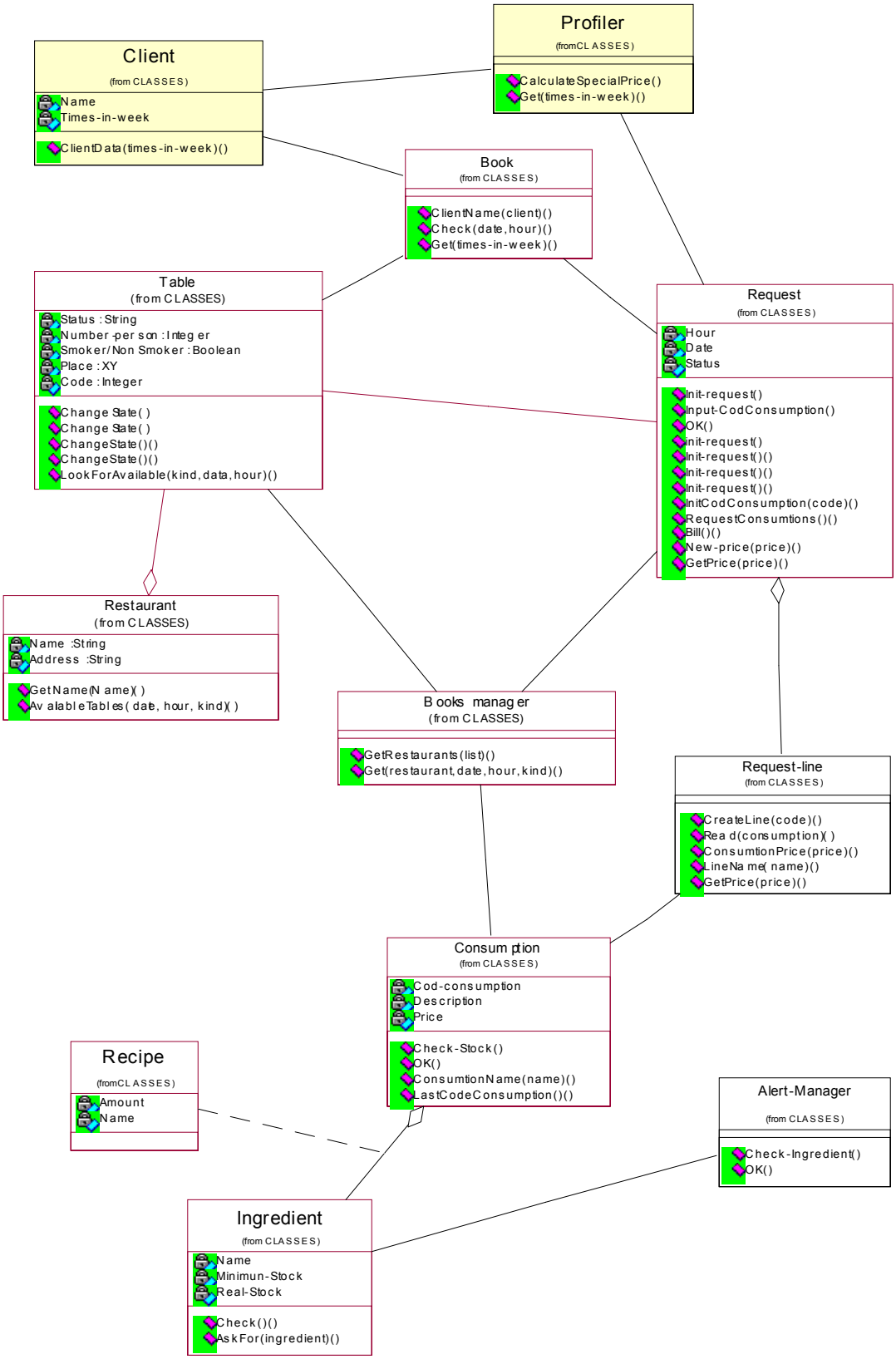


Figure C.42 Class diagram for the first application with User Profiler mechanism

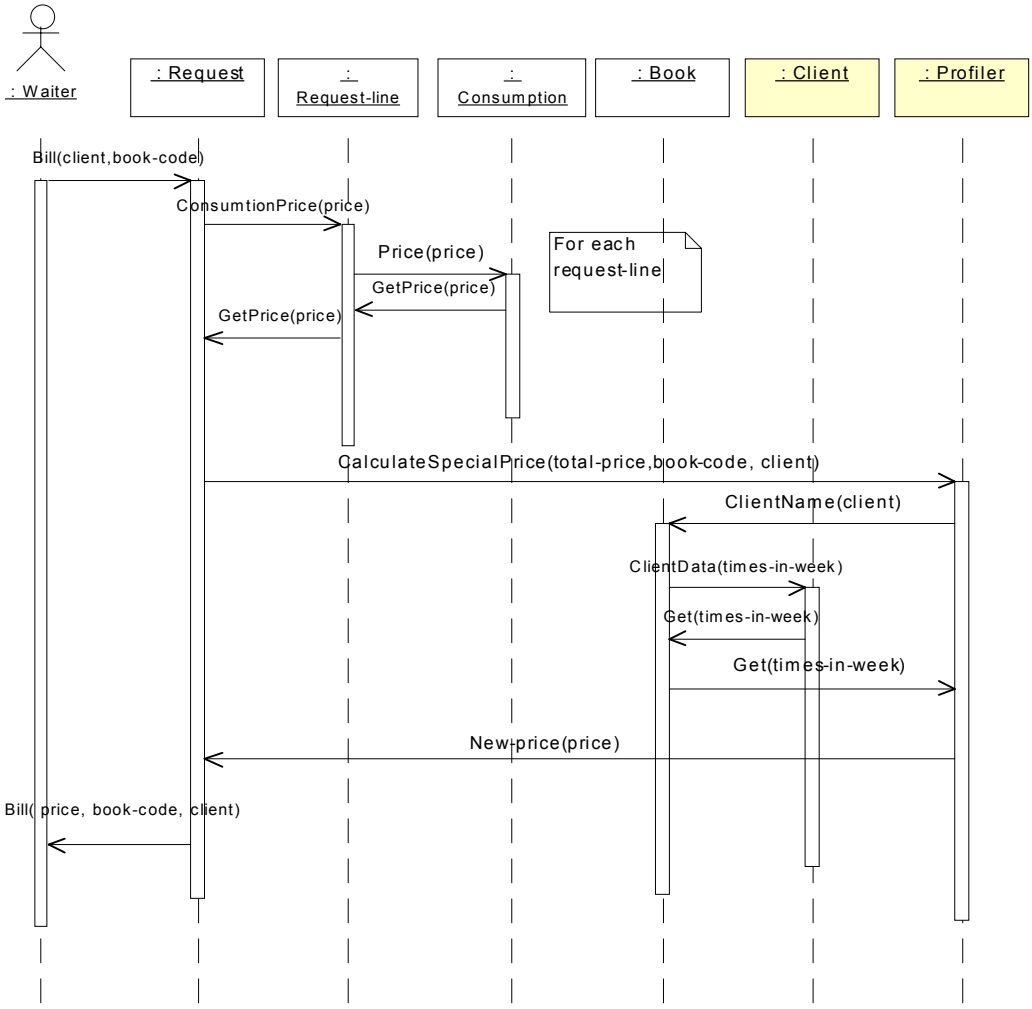
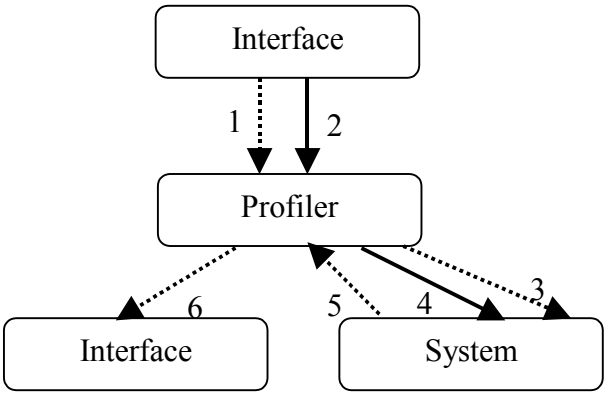


Figure C.43 Sequence diagram for the first application with User Profiler mechanism

STEP 3. Abstraction of the design solution for User Profiler

- **Solution:**
  - Diagram:



  - Participants:

- Interface:
  - For profile information creation, it sends both the data (1) and the operation (2) that the user defines for his system to the profiler.
  - For profile retrieval, the interface sends the profile data (1) to the profiler. Additionally, profiler sends the data associated with this profile to the interface.
- Profiler: .
  - For profile information creation, it receives the data (1) and the operation (2) that the user defines for his system from the interface. If it is not capable of storing this profile information internally, it will send it to another system component through (3) and (4).
  - For profile retrieval, it receives the data of the profile to be retrieved (1) from the interface. If it does not store the profile information internally, it will ask another system component to process the requested information and/or operation (3) (4) and will receive the information associated with the required profile (5) from this system component. Then, if this information is to be displayed by the interface, it will send it to the interface through (6).
- System: this component is optional and will only exist if profiler is not capable of storing the information associated with each system profile internally. It receives the data and/or operations of the required profile type (3) (4) from profiler and sends the data associated with this profile (5) to profiler.

C.14 Shortcuts First Iteration

STEP 1. Design solution without Shortcuts

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

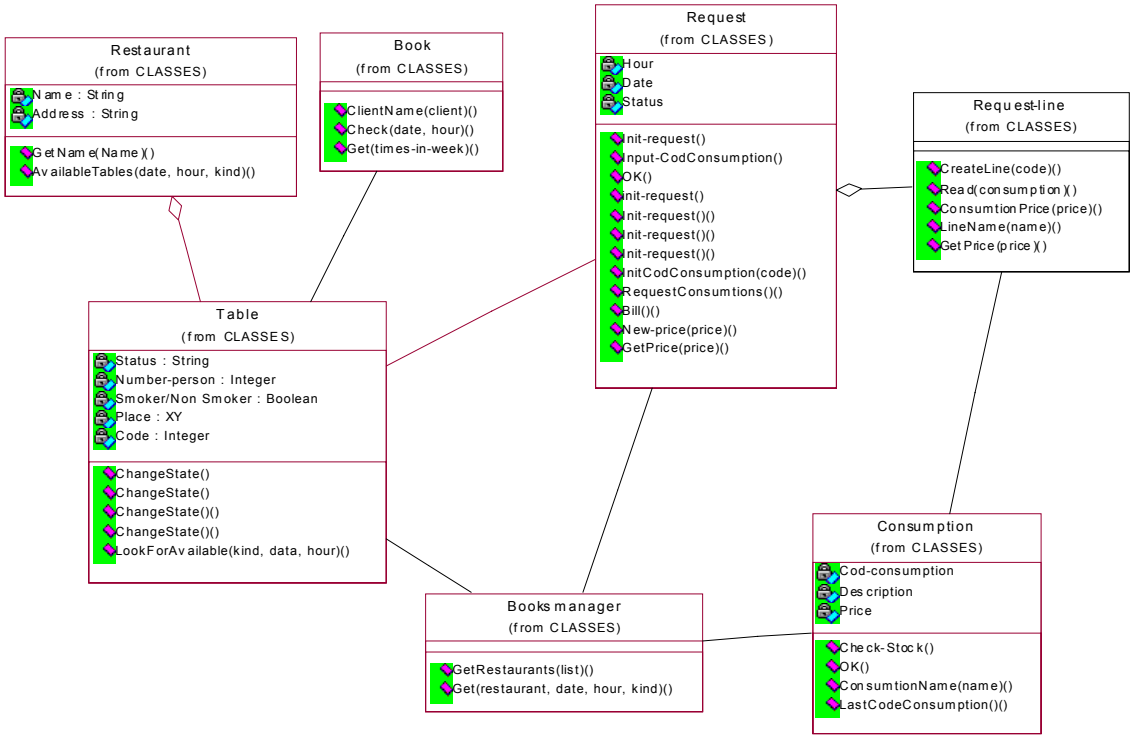


Figure C.44 Class diagram for the first application without Shortcuts mechanism

STEP 2. Design solution with Shortcuts

Requirement: the waiter presses F1, which corresponds to the function that tells the waiter to go and collect a given order that has now been cooked.

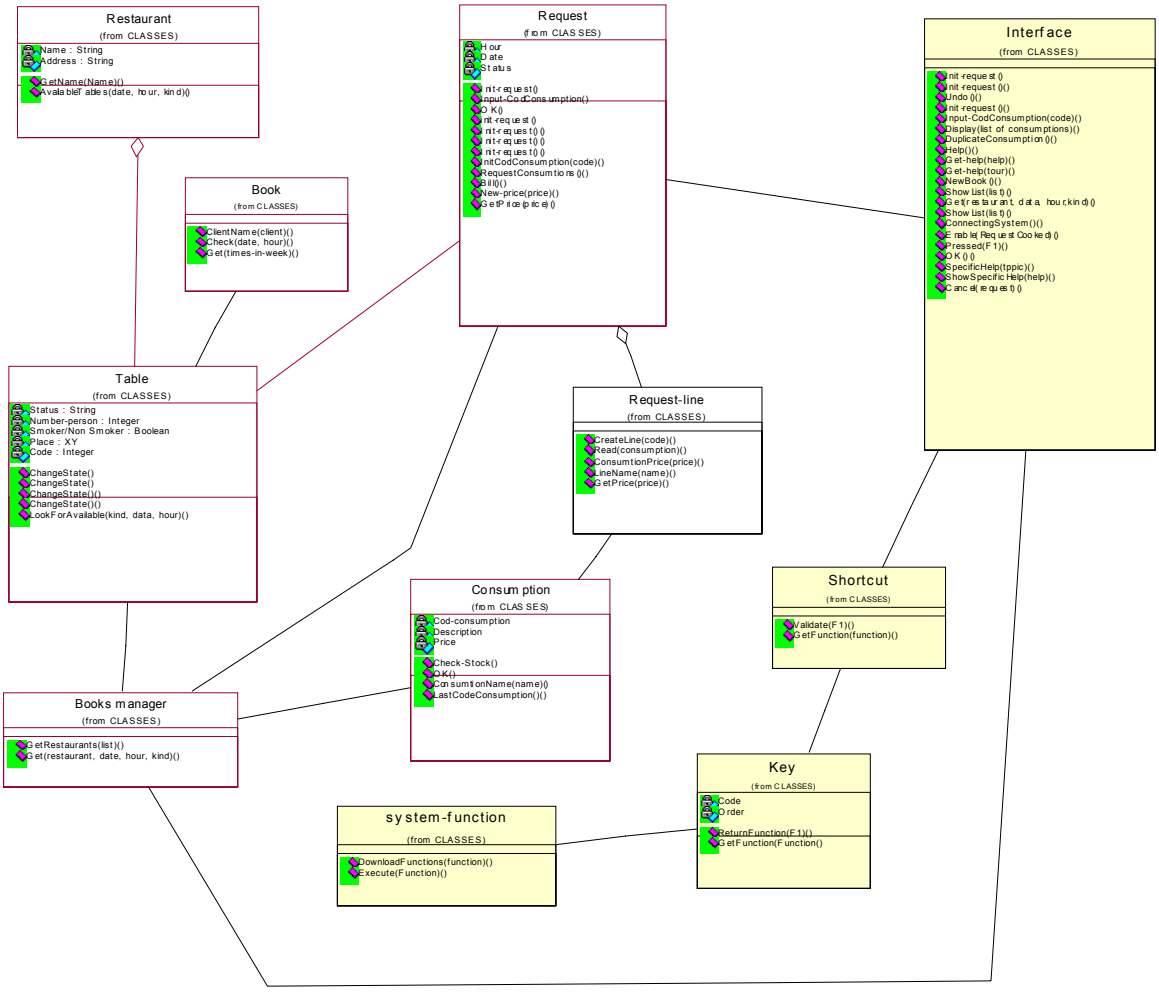


Figure C.45 Class diagram for the first application with Shortcuts mechanism

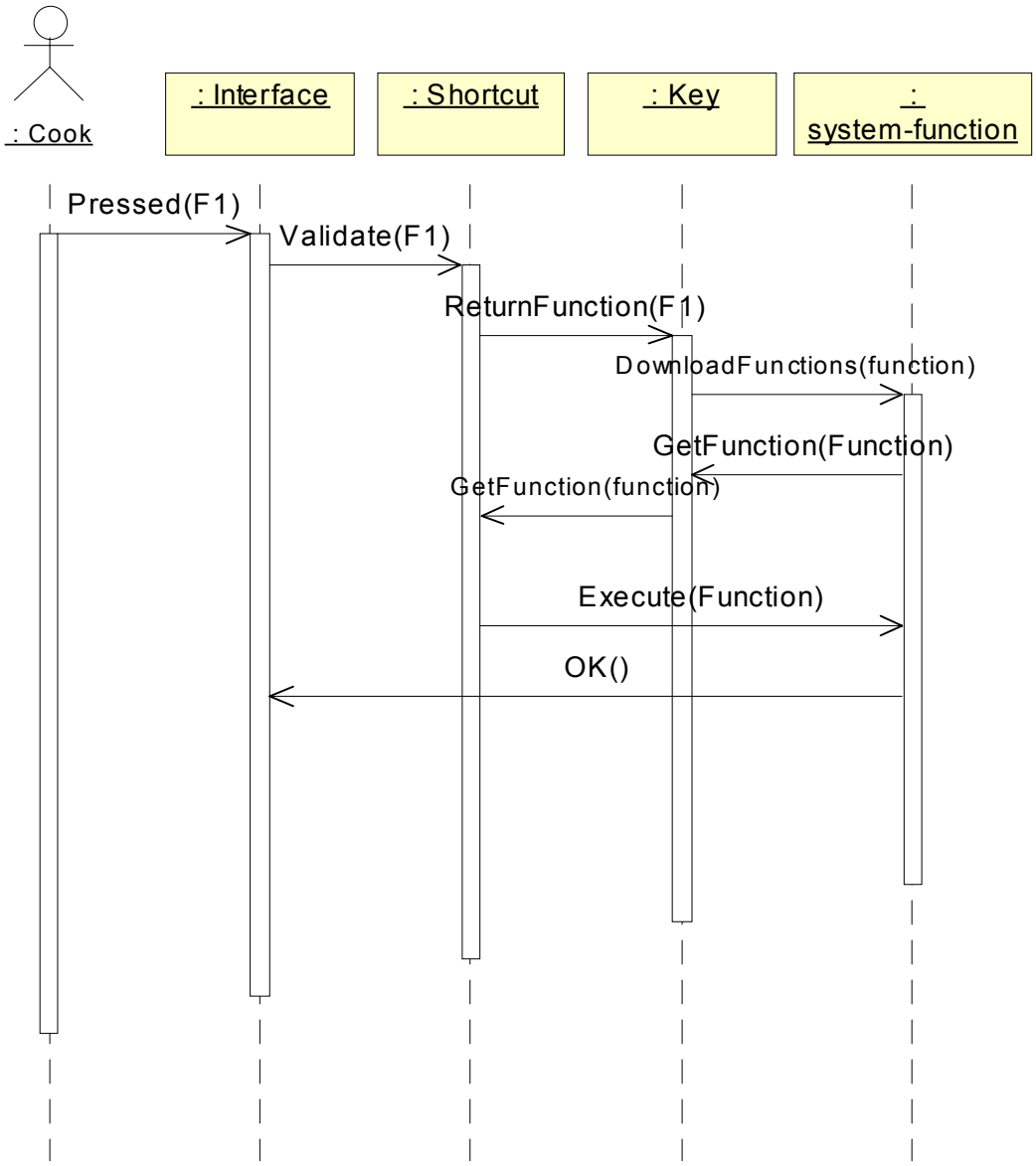
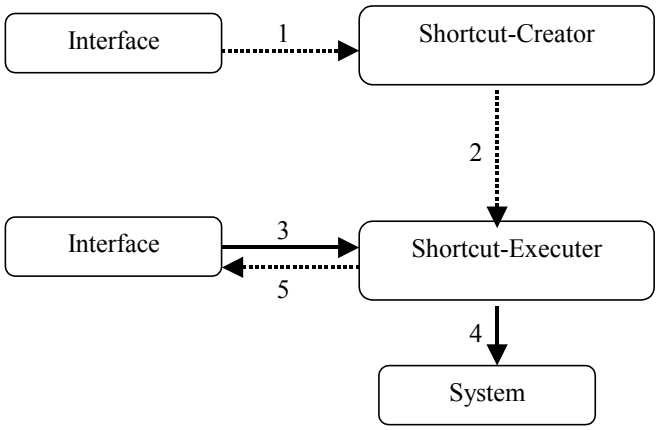


Figure C.46 Sequence diagram for the first application with Shortcuts mechanism

STEP 3. Abstraction of the design solution for Shortcuts

- **Solution:**
  - Diagram:



- Participants:
  - Interface: it sends a data set (1) corresponding to a given system function, as well as the key combination that activates this function, to Shortcut-creator. Additionally, if a shortcut is to be executed, it sends a key combination (3) to the Shortcut-executor. When the shortcut has been executed, it will receive the result of the requested functionality or error if it cannot be executed (5) from Shortcut-executor.
  - Shortcut creator: it fills in a sort of array in which the name of the shortcut, the commands that activate it and the system function to be activated with these quick commands are stored. For this purpose, it receives a data set and the function to be executed when these keys are combined (1) from the interface, which it sends to Shortcut-executor for storage (2).
  - Shortcut executor: it receives a set of commands (3) from the interface and checks whether they match a set of commands associated with a given function. If the command set matches a system functionality, it requests the system to execute the function associated with this shortcut (4). In any case, whether they match a function or not, it sends the result of executing this function to the interface through (5).
  - System: it receives the order to execute the function associated with this key combination (4) from shortcut-executor.

C.15 Context Sensitive Help First Iteration

STEP 1. Design solution without Context Sensitive Help

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

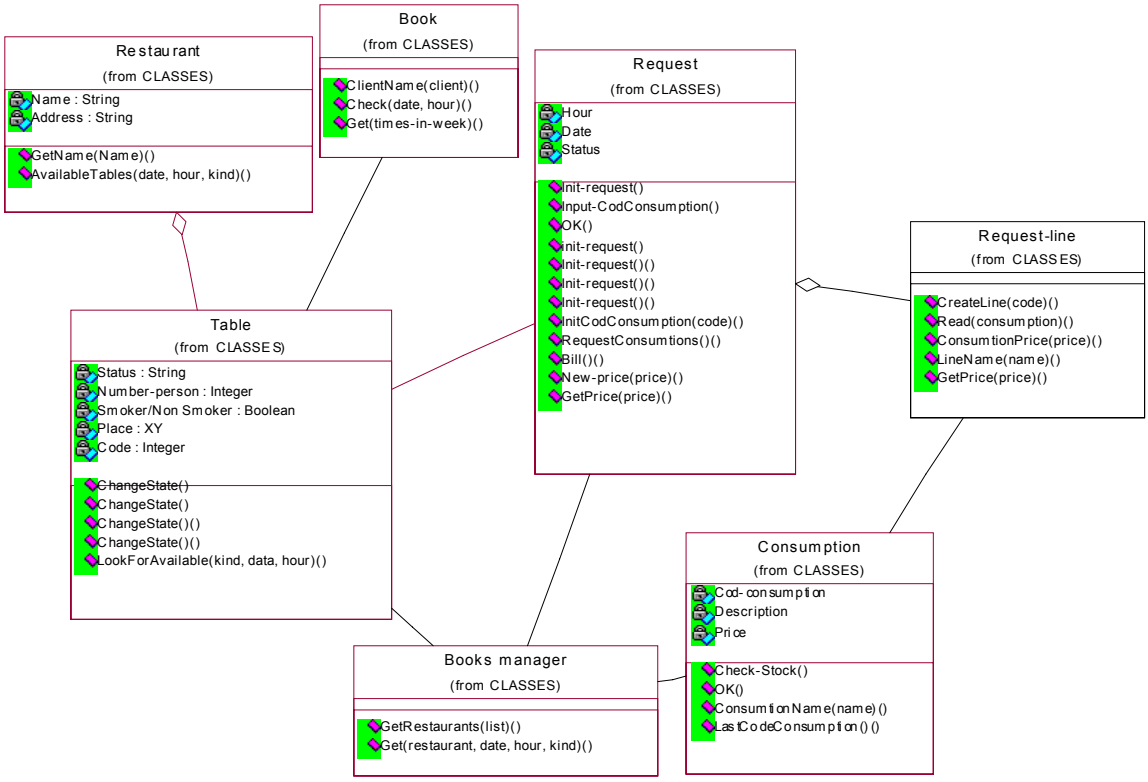


Figure C.47 Class diagram for the first application without Context Sensitive Help mechanism

STEP 2. Design solution with Context Sensitive Help

Requirement: the user can push the Sensitive Help button

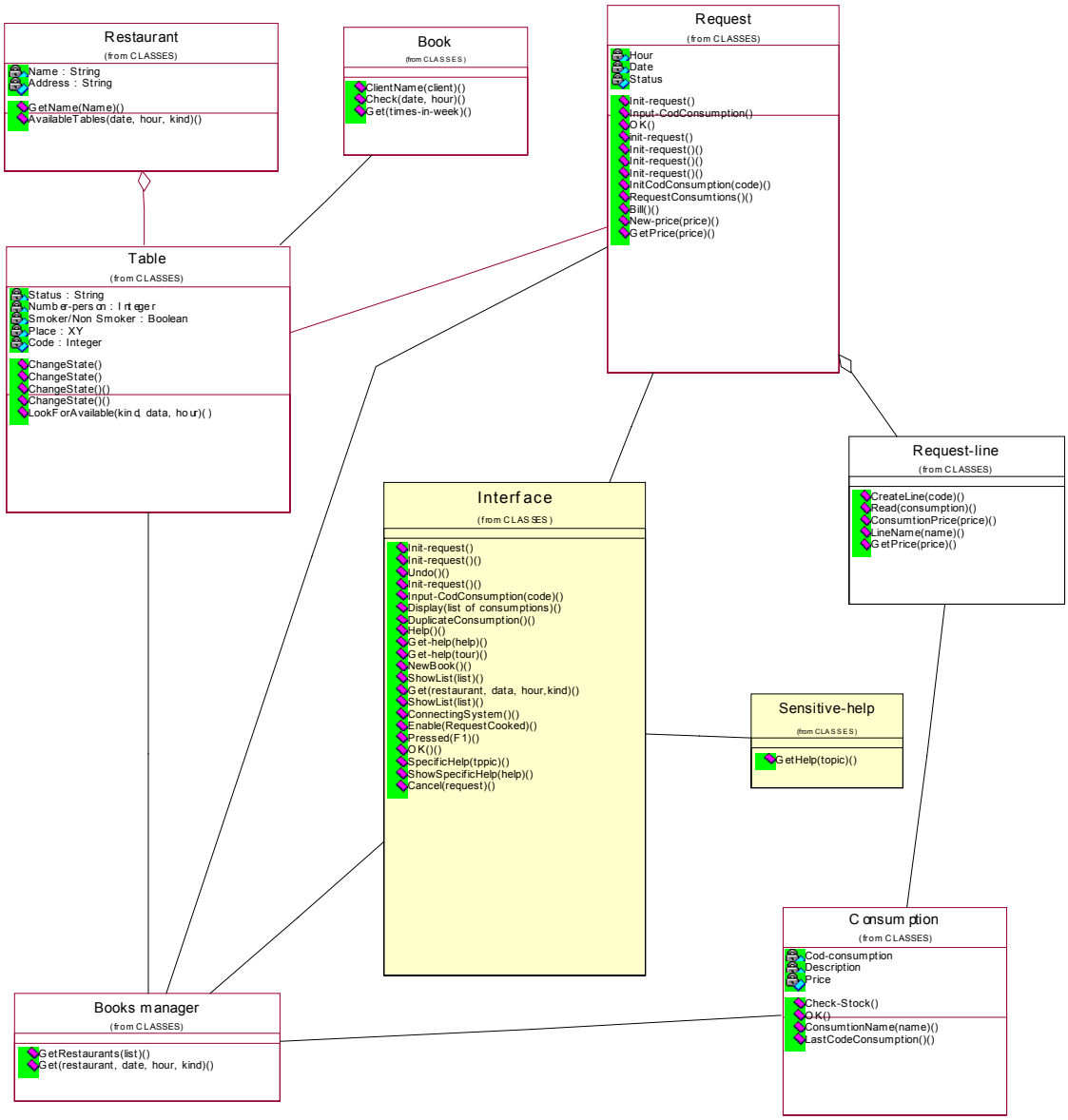


Figure C.48 Class diagram for the first application with Context Sensitive Help mechanism

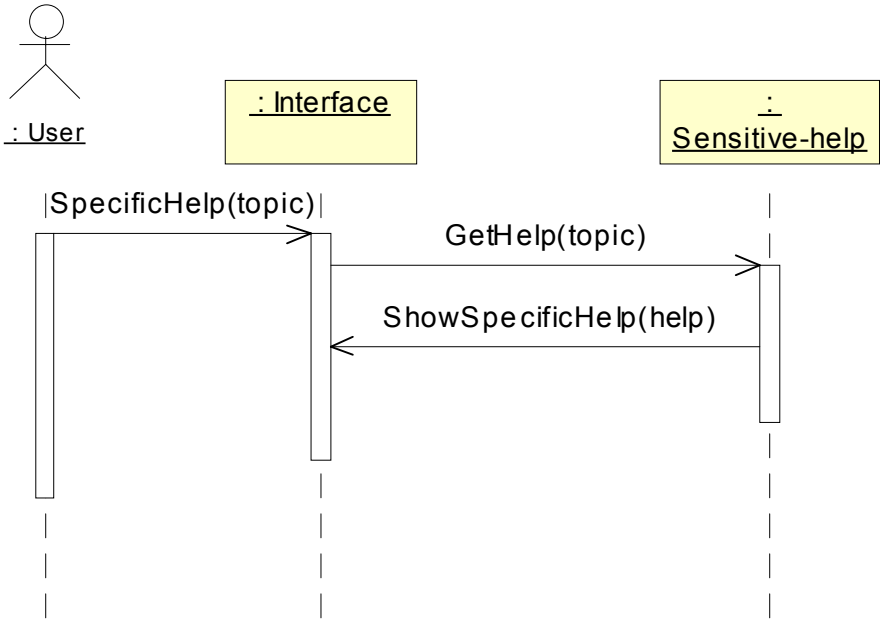
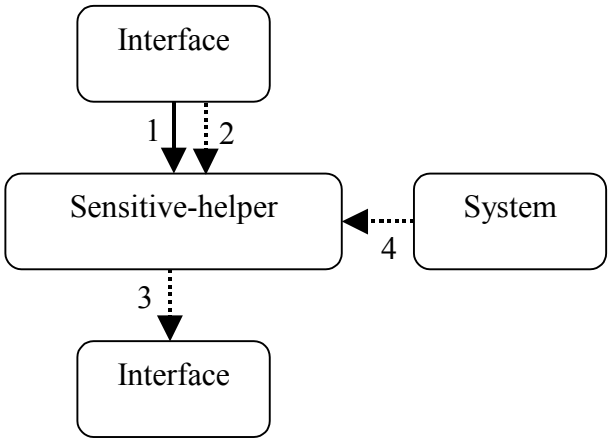


Figure C.49 Sequence diagram for the first application with Context Sensitive Help mechanism

STEP 3. Abstraction of the design solution for Context Sensitive Help

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it warns the sensitive-helper through (1) that the cursor is on top of the element specified in (2). Additionally, it will display the help information it receives from Sensitive-helper (3).
    - Sensitive-helper: it identifies the help associated with a given element. This component receives the signal (1), which alerts it to the need to show help about the specified element through (2), from the interface. If the help is not stored internally in this component, this help will be provided by another part of the system through the information flow from System (4). When it has the help data, it informs the interface through (4).

- System: this component is optional and represents a part of the system in which the help will be stored if the Sensitive-helper is not capable of storing it internally. In this case, System will provide the help to the Sensitive-helper through (4).

C.16 Wizard First Iteration

STEP 1. Design solution without Wizard

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

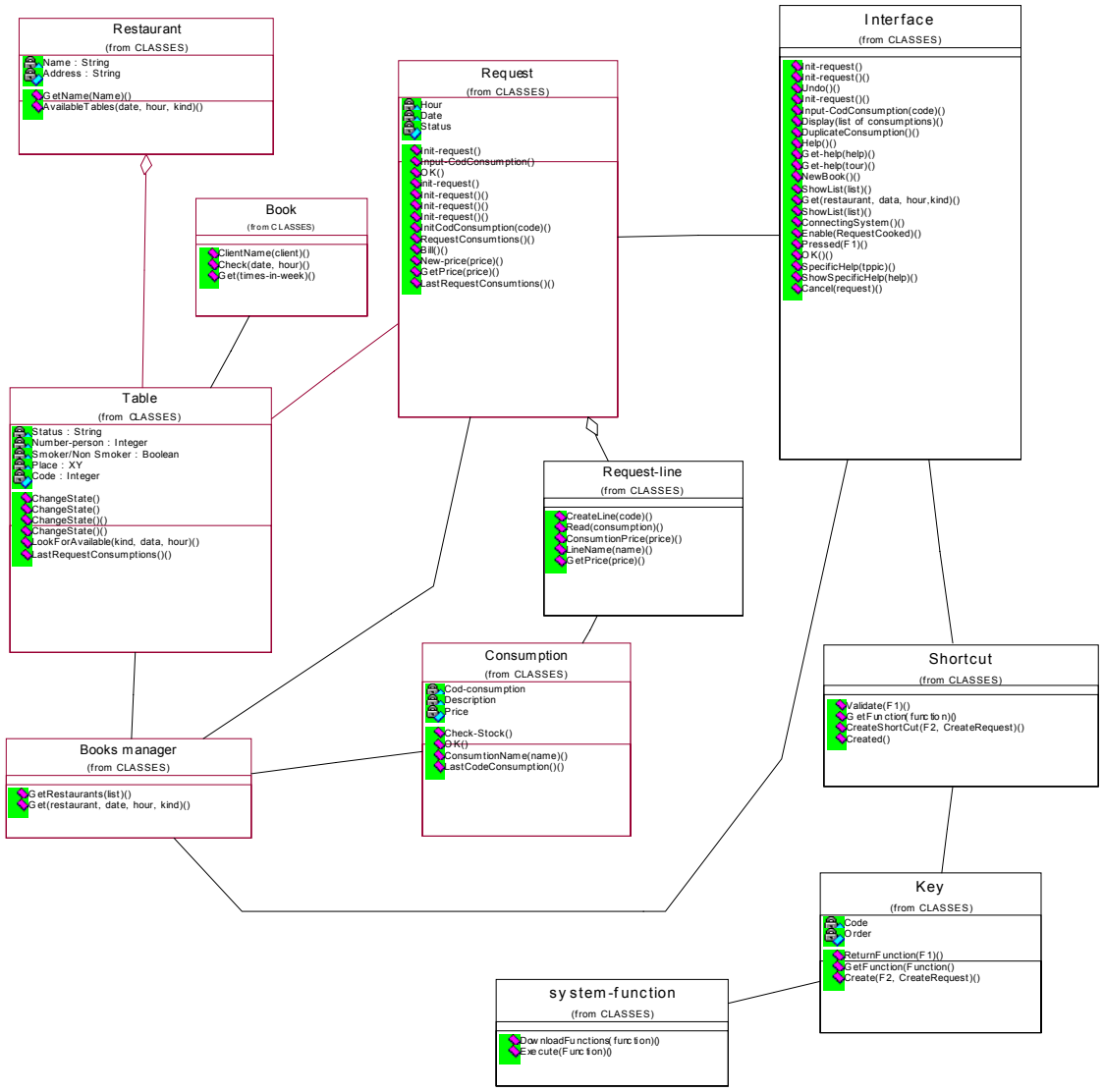


Figure C.50 Class diagram for the first application without Wizard mechanism

STEP 2. Design solution with Wizard

Requirement: the waiter creates a rapid access for the functionality “Create new order” by pressing F2.

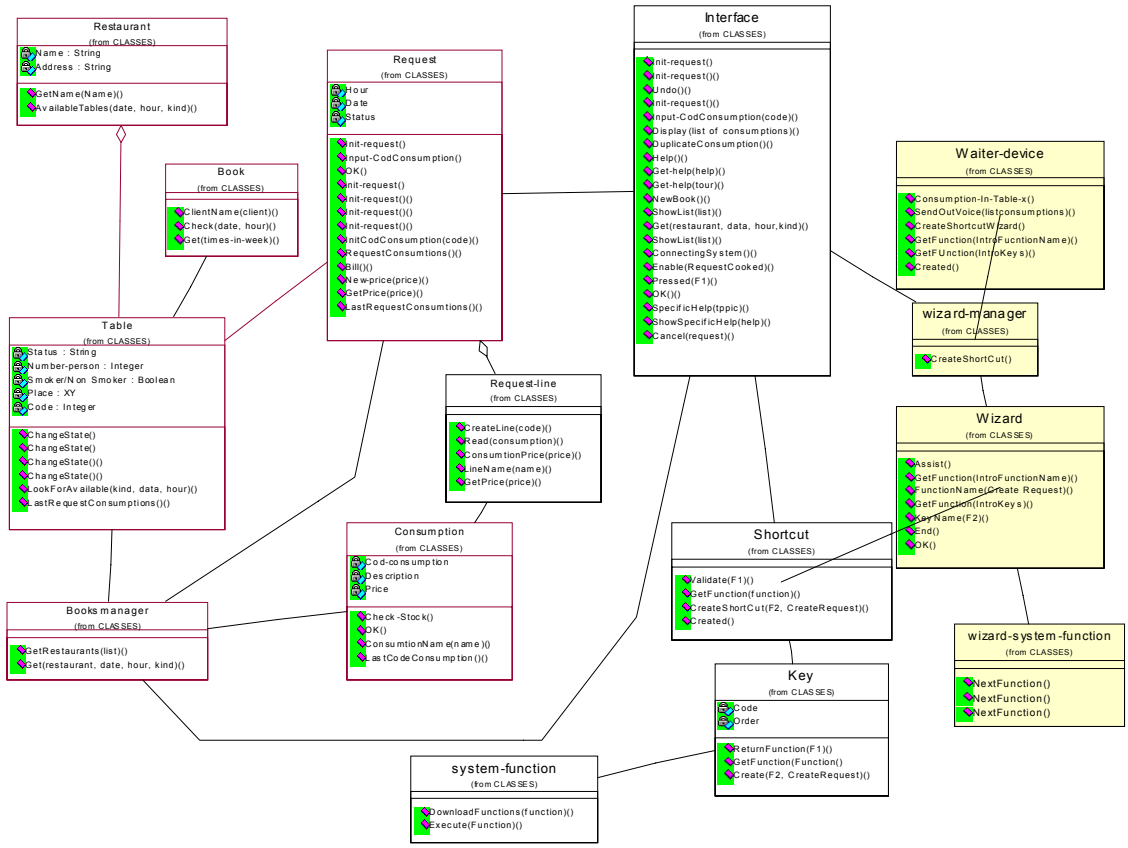


Figure C.51 Class diagram for the first application with Wizard mechanism

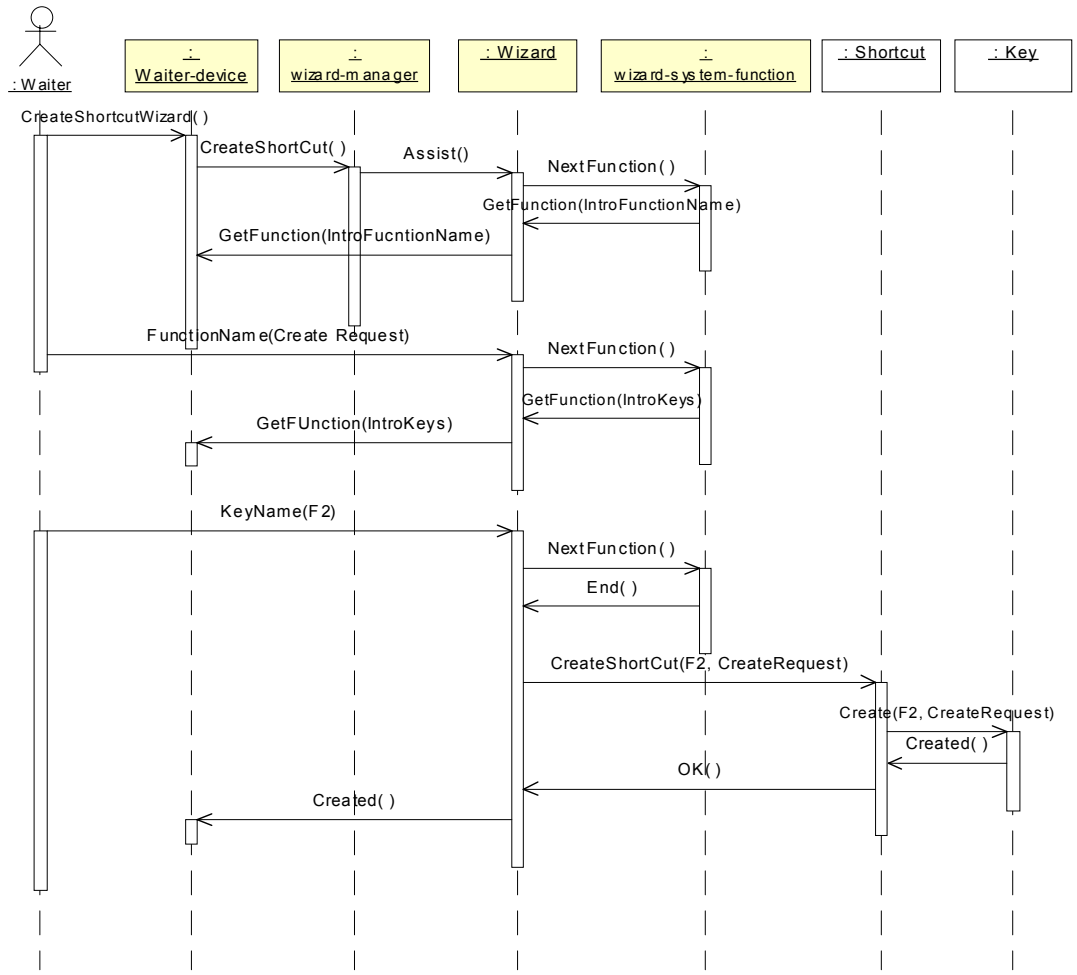
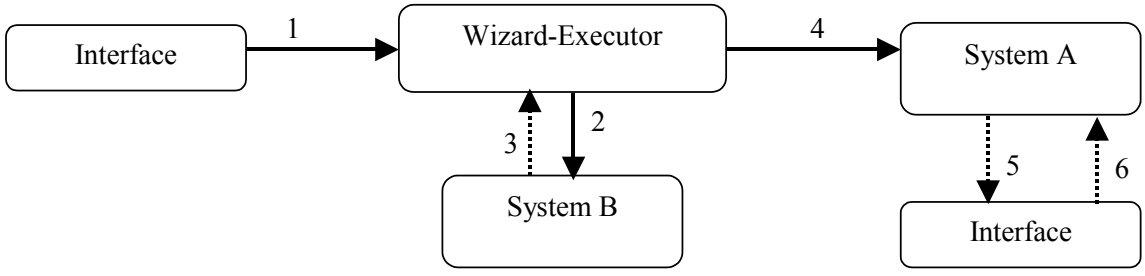


Figure C.52 Sequence diagram for the first application with Wizard mechanism

STEP 3. Abstraction of the design solution for Wizard

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends the functionality to be assisted (1) to Wizard-executor. Additionally, for every step in wizard execution for which the user needs to enter information or make a decision, System A sends this notification to the interface through (5). Once the interface has the required information, it sends it to System A through (6).

- Wizard-executor: it receives the request to execute a given wizard (1) from the interface. The information related to the wizard can be stored in the Wizard-executor or another system component. If Wizard-executor does not store the different steps of the wizard internally, it consults System B through (2), and, receives the information on the function to be executed to perform the different steps of the wizard from System B through (3). For each step to be taken, Wizard-executor asks the System to execute the functionality associated with each step through (4).
- SystemA: it represents the part of the system that executes each step of the wizard. It receives the different functions to be executed from the Wizard-executor through (4) and, if user intervention is required, System A will inform the interface through (5) and will receive the information entered by the user through the interface by means of (6).
- SystemB: This module is optional and will only be necessary if the Wizard-executor does not store the steps for each wizard that can be executed in the system internally. It receives the request for the name of the next step in wizard execution from the wizard-executor (2) and returns the information on the name of the function to be executed through (3).

C.17 Cancel First Iteration

STEP 1. Design solution without Cancel

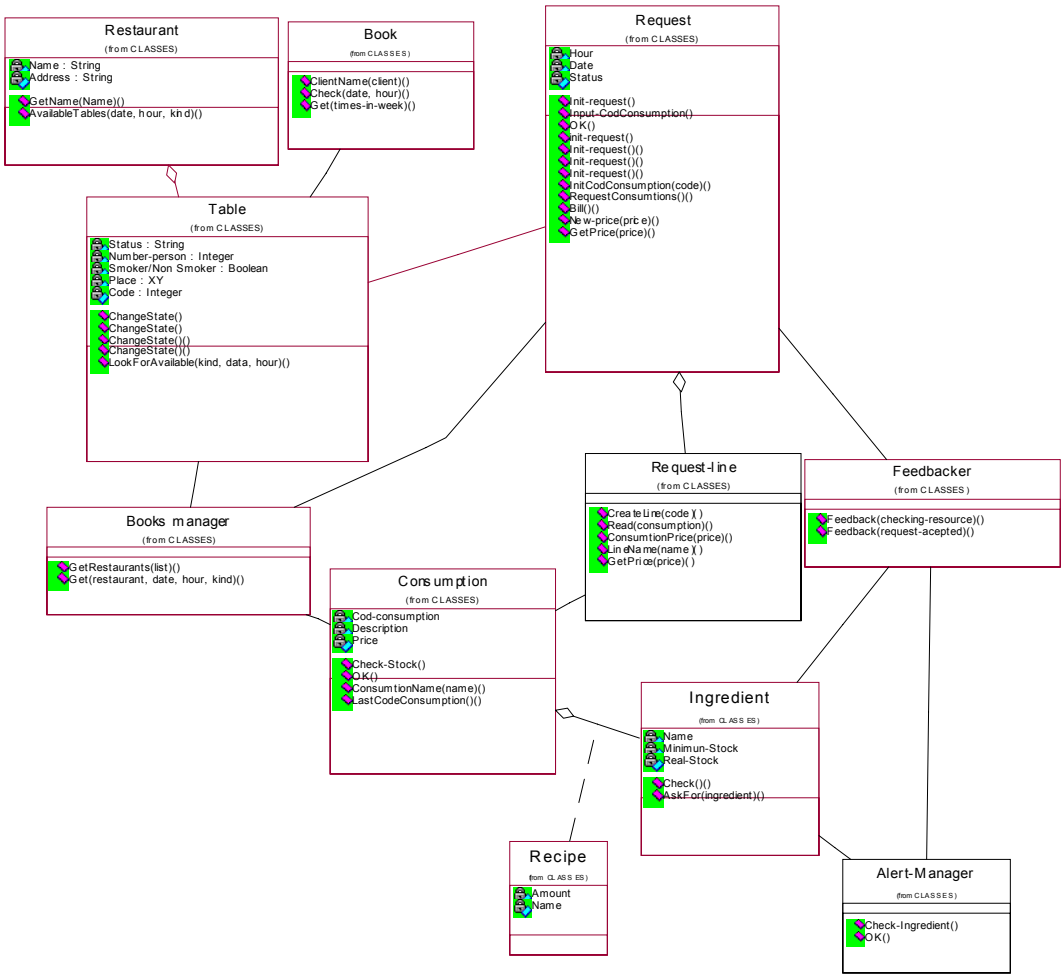


Figure C.53 Class diagram for the first application without Cancel mechanism

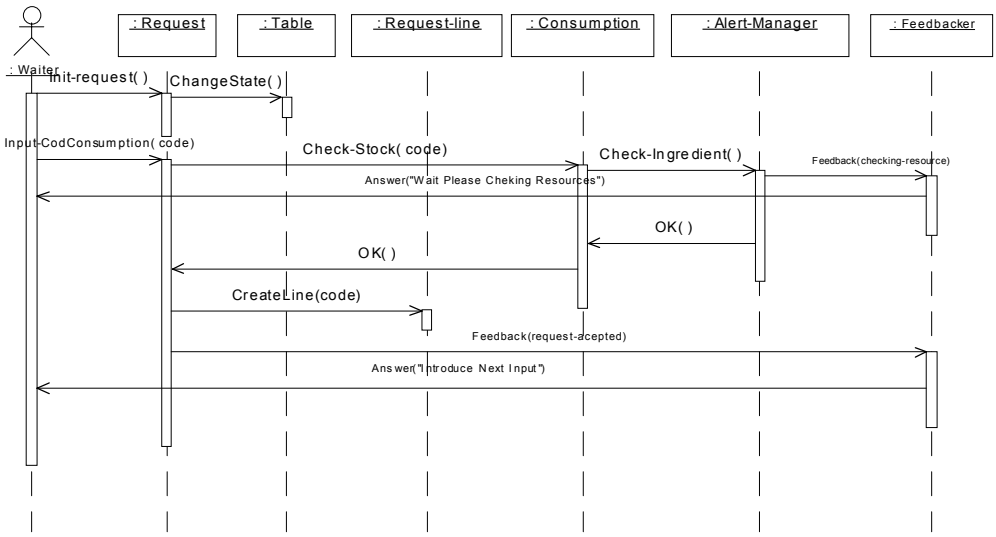


Figure C.54 Sequence diagram for the first application without Cancel mechanism

STEP 2. Design solution with Cancel

Requirement: The waiter can cancel an order even if it has not be sent to the kitchen.

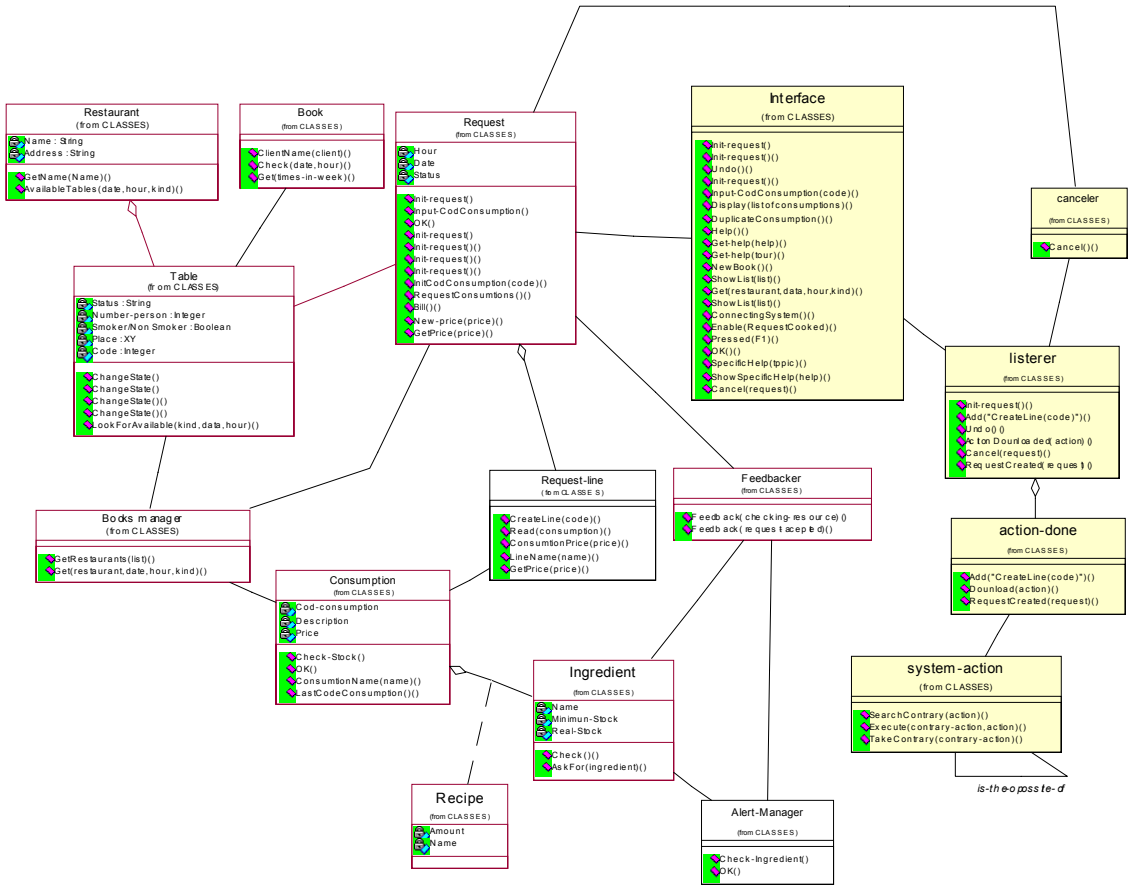


Figure C.55 Class diagram for the first application with Cancel mechanism

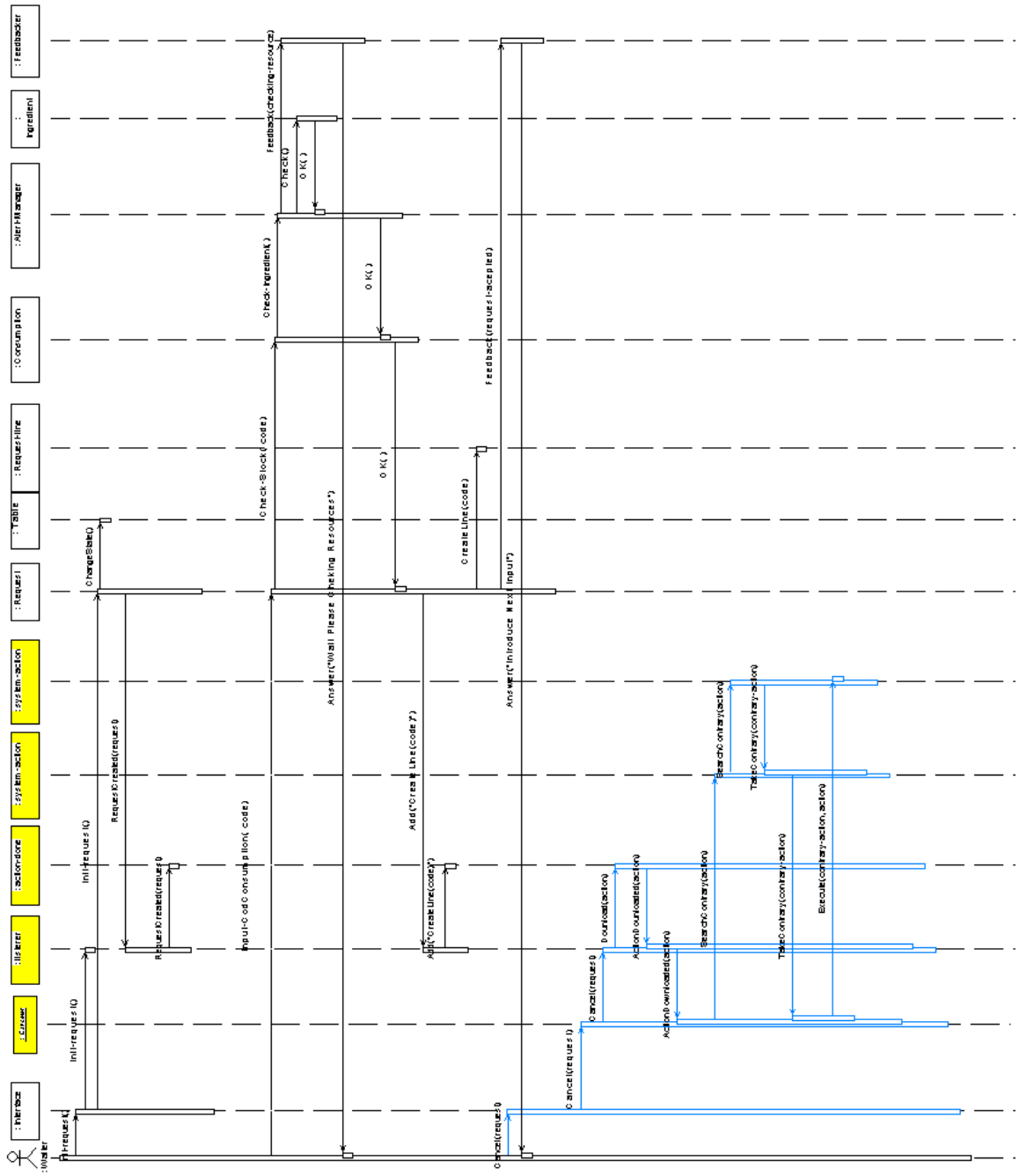
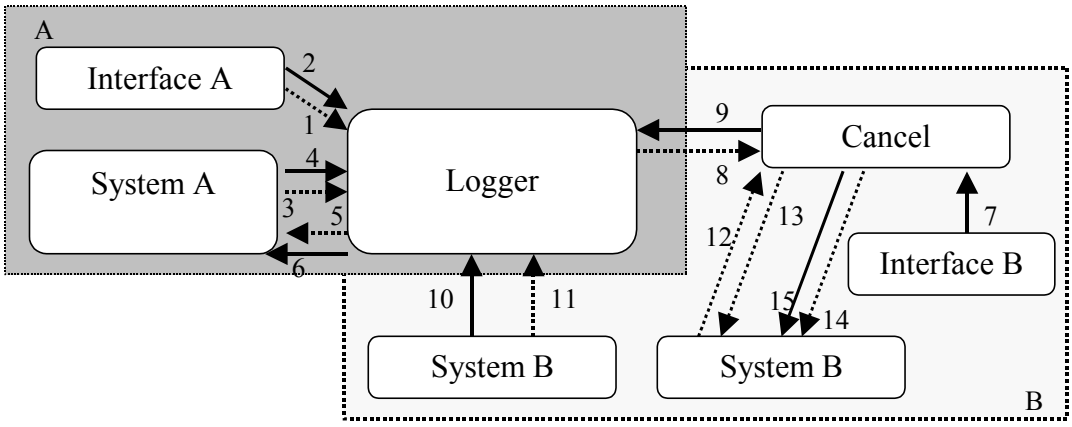


Figure C.56 Sequence diagram for the first application with Cancel mechanism

STEP 3. Abstraction of the design solution for Cancel

- **Solution:**
  - Diagram:



- Participants:
  - InterfaceA: it receives the request to execute an operation in the system, which may contain both the operation and the data (1) (2). As we will see later, this execution request can also come from the system (3) (4).
  - SystemA: this module sends the functions and data to be executed in the system (3) (4) to the logger, and also, optionally, if the logger does not store the logged actions internally, sends the information to the part of the system that manages these actions (5) (6).
  - Logger: this module receives the actions and the data requested by the user or from another part of the system (1) (2) (3) (4) and stores the logged actions and data either internally or in another part of the system, in which case it will have to send this action and the data to be processed by the respective part of the system to the system (5) (6). Logger receives the cancel request (9) from Canceler, then, if the logged actions are stored internally, it sends them one by one to Canceler in (8), provided that the all the operations stored by the logger have been performed. If the operations have only be stored but not executed, then nothing is sent in (8) and if they are stored externally, all the logger will have to do is receive them in (10) and (11) and delete them. If they are not stored internally, it will receive both the data and the operation to be cancelled from another part of the system, which we have called System B, by means of (11) and (10), respectively.
  - Interface B: it receives the cancel request and sends it to Canceler in (7). Additionally, it will search the system for both the action performed and the data associated with this operation (10) (11), provided that the logger does not store the data internally.
  - System B: it searches the system for both the action performed and the data associated with this operation (10) (11), unless the logger stores the data internally. It receives the actions to be undone. It receives the actions to be undone (13) and provides the opposite action (12) (for which purpose, it will have to store the opposite for each action, see implementation section for example). The opposite action and the respective data will be sent to the respective part of the system (15) and (14) for execution.
  - Canceler: it sends the cancel request (9) to logger and also sends each of the actions to be undone that it receives from logger to System B (13) and receives the opposite operation to the one performed (13) from system B. When it knows what opposite operation to be performed is, it sends it to System B along with the data associated with this operation through (14) and (15). Alternatively, if all the operations are stored in the system and performed together when the user presses accept, then Canceler will simply read through (10) and (11) and delete the accumulated operations, in which case (14) and (15) will not be used at all.

C.18 Multi-tasking First Iteration

STEP 1. Design solution without Multi-tasking

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

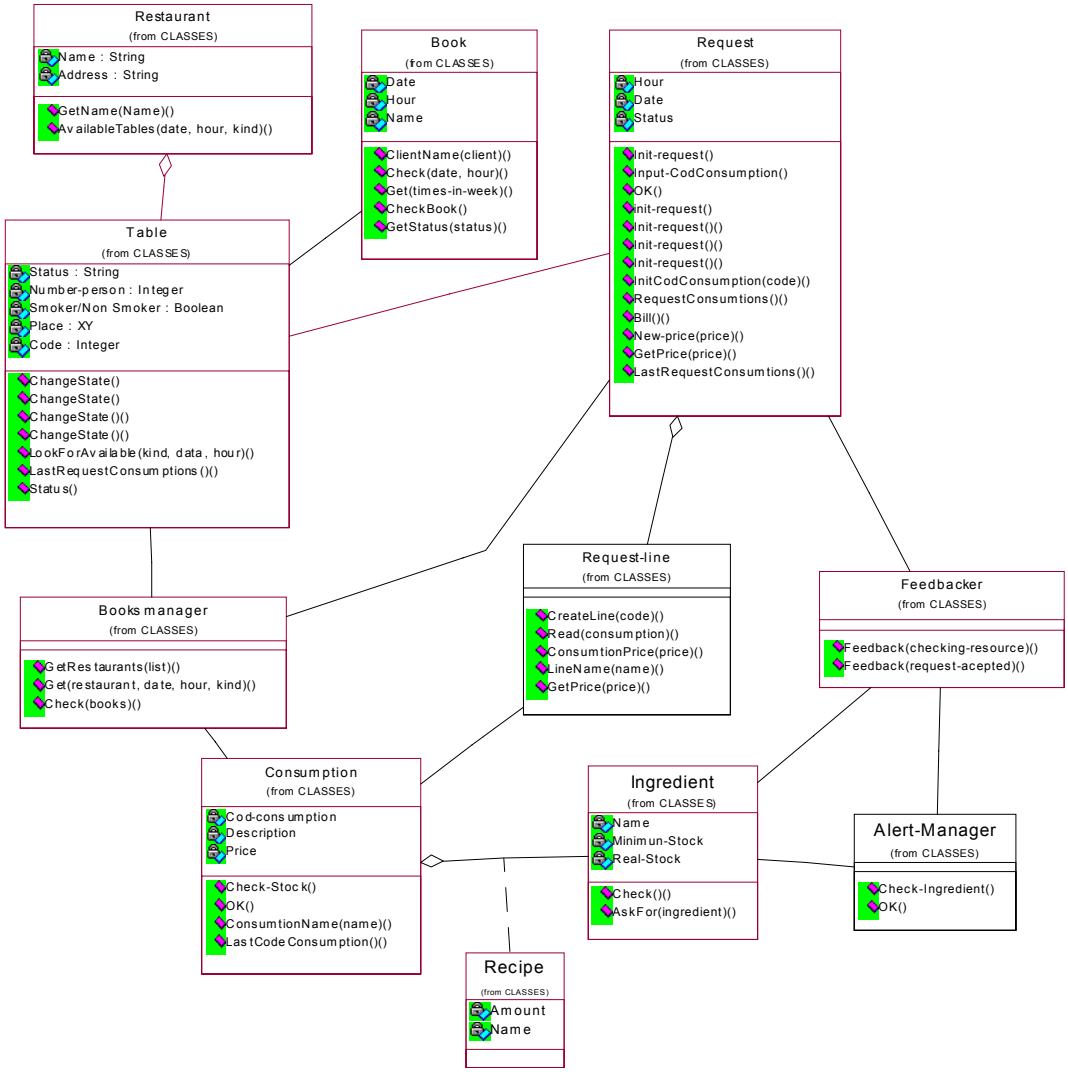


Figure C.57 Class diagram for the first application without Multi-tasking mechanism

STEP 2. Design solution with Multi-tasking

Requirement: the clock advises that any reservations made should be cancelled 20 minutes after reservation time if the diners have not arrived.

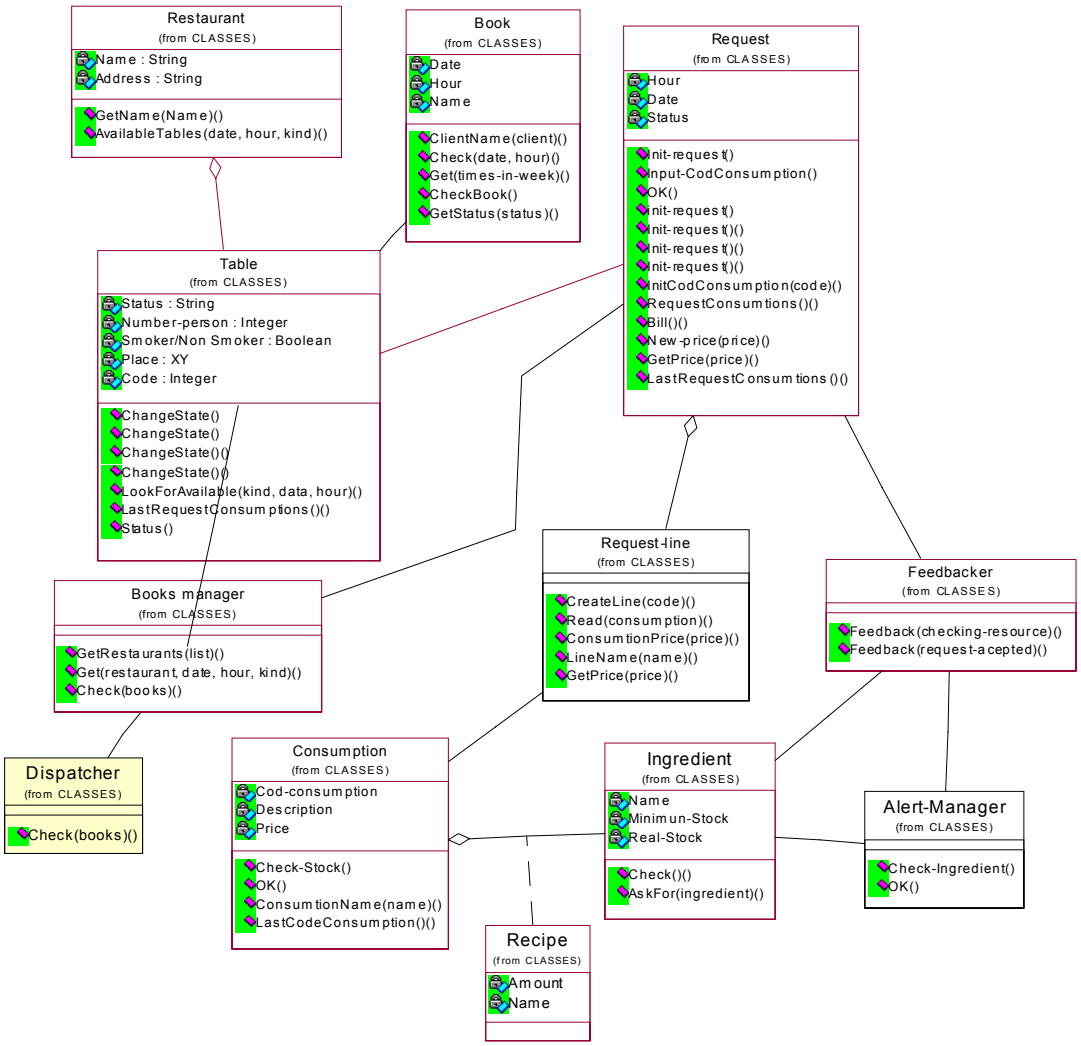


Figure C.58 Class diagram for the first application with Multi-tasking mechanism

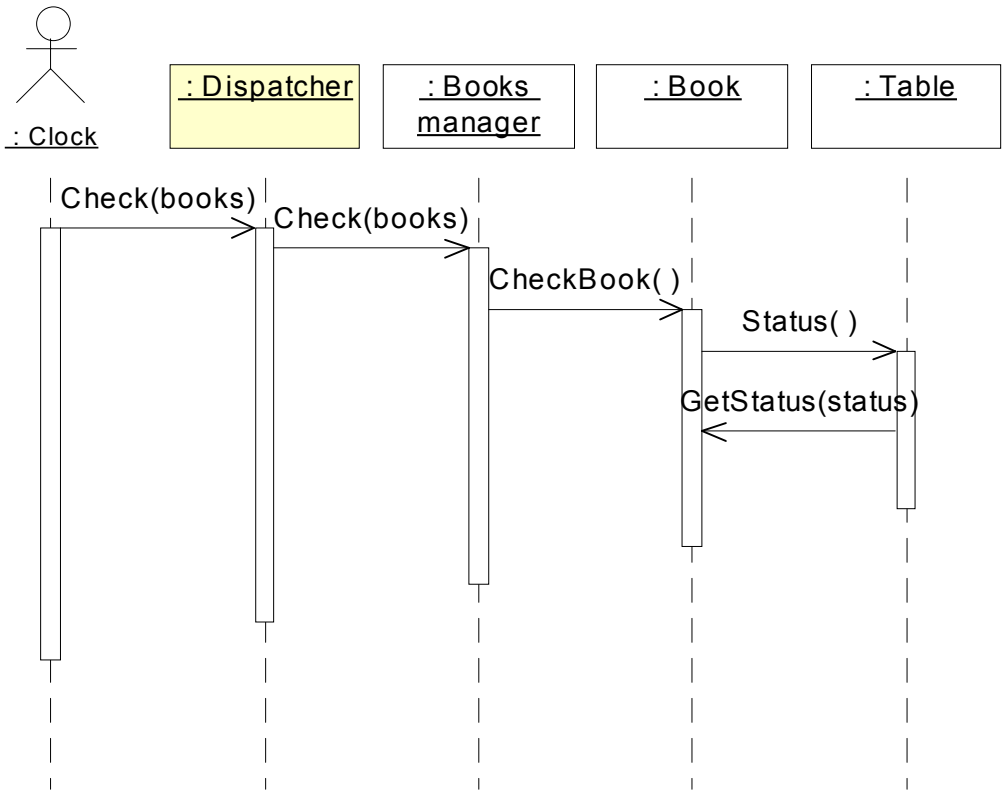
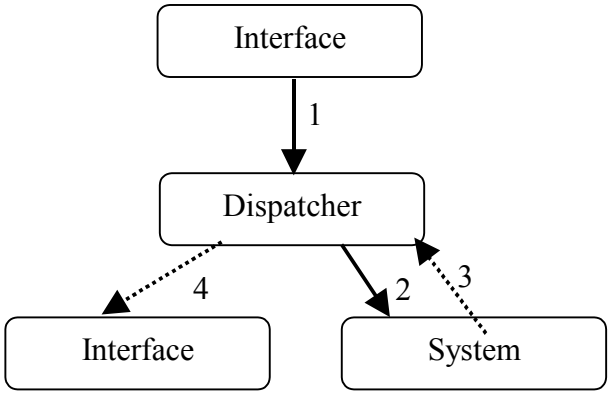


Figure C.59 Sequence diagram for the first application with Multi-tasking mechanism

STEP 3. Abstraction of the design solution for Multi-tasking

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends the function to be executed to dispatcher in (1). Additionally, if the user is to be informed of anything that is happening, he receives information from the dispatcher in (5).
    - Dispatcher: this component knows what resources are needed for each function that it has to execute in the system. It receives the function to be executed from the interface in (1). It sends the function to be executed to the system component in question in (2) after having checked that all the resources required to execute

this function exist. Additionally, it receives the result of performing this operation from the system in (3). This result may specify either error or OK if everything went according to plan. If user has to be informed of the result of the operation performed, it sends this information to the interface (4).

- System: this component refers to the part of the system responsible for executing the function specified by dispatcher in (3).

C.19 Command Aggregation First Iteration

STEP 1. Design solution without Command Aggregation

In this case, the interaction diagram does not appear because the "macros" functionality is new.

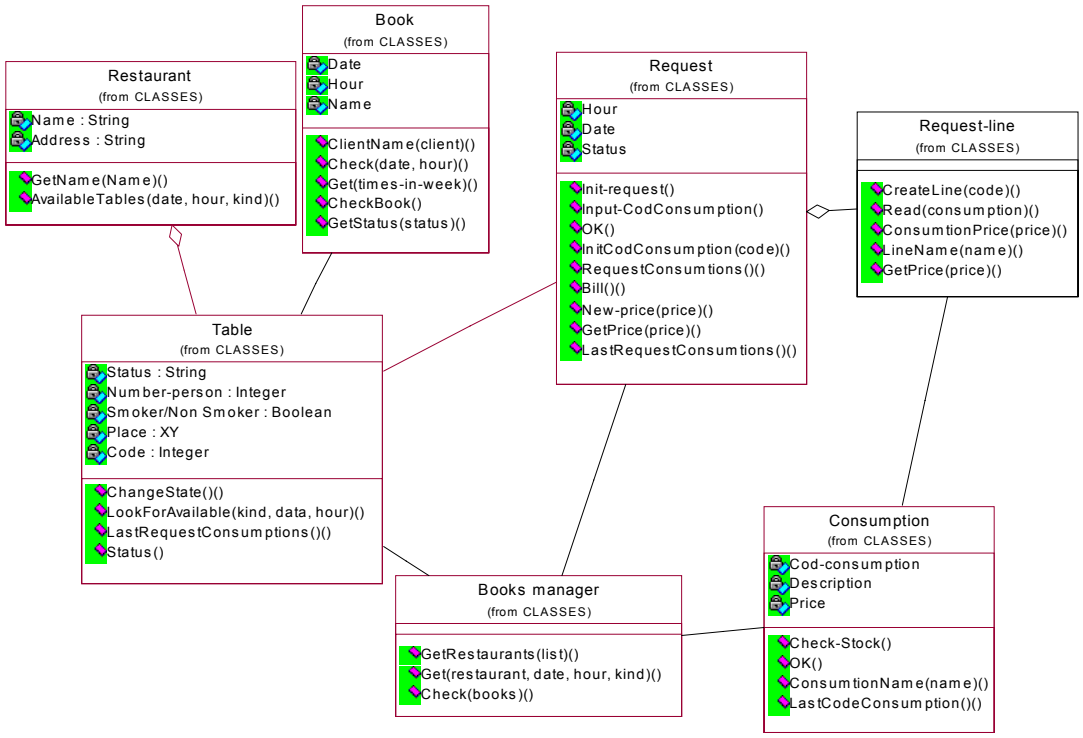


Figure C.60 Class diagram for the first application without Command Aggregation mechanism

STEP 2. Design solution with Command Aggregation

Requirement: the system must have the capability to create macros, for instance, to create a macro that would permit a maître d'hôtel to change the permitted period for arrival at the restaurant before the booking time, taking into account the number of bookings for each day.

In the next example is represented the creation of a macro.

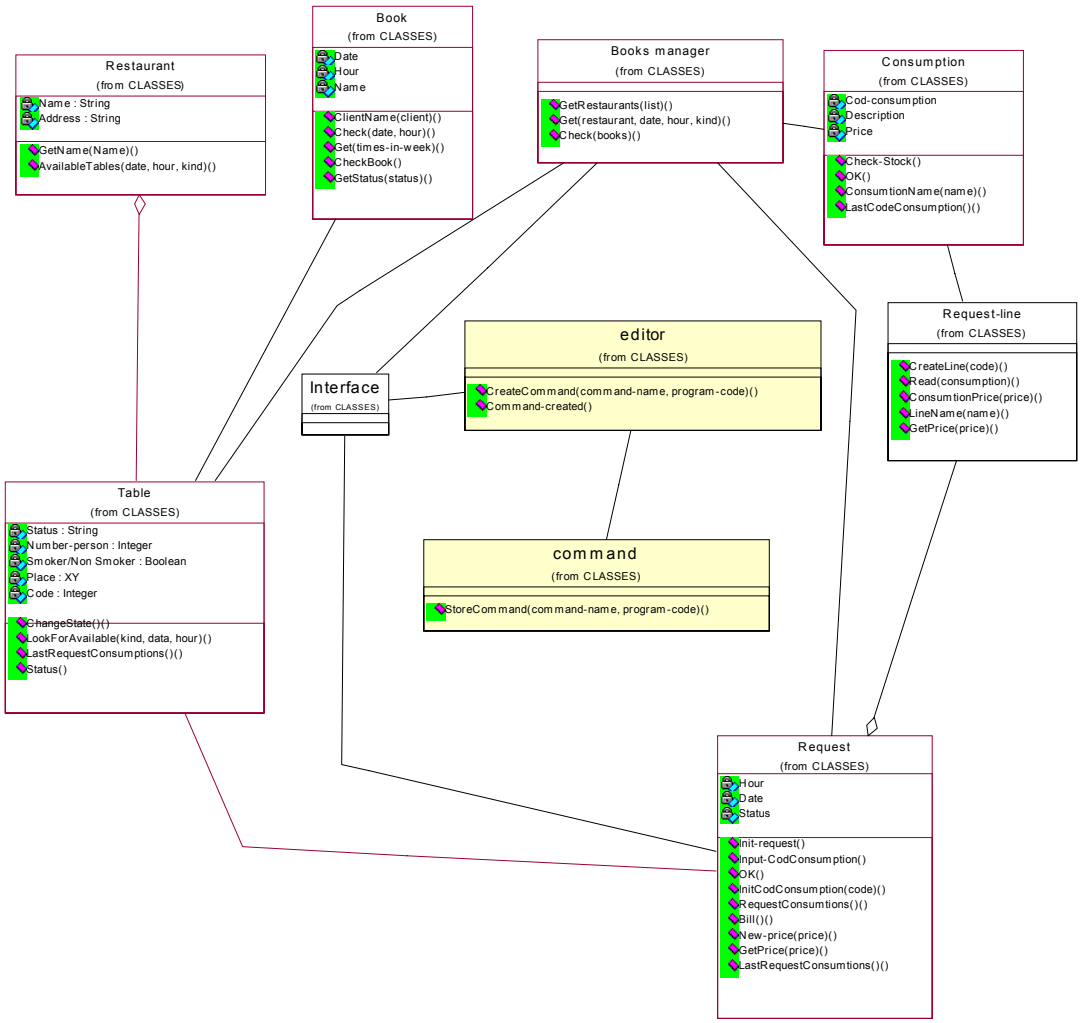


Figure C.61 Class diagram for the first application with Command Aggregation mechanism

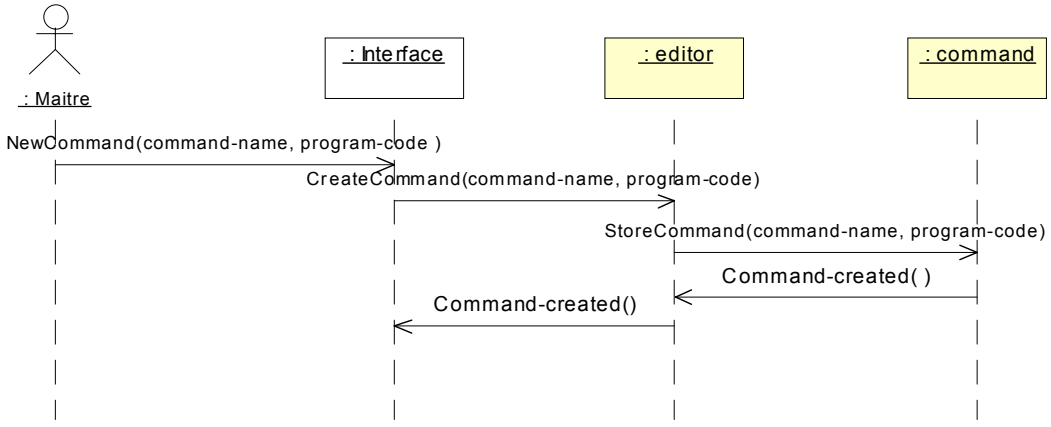
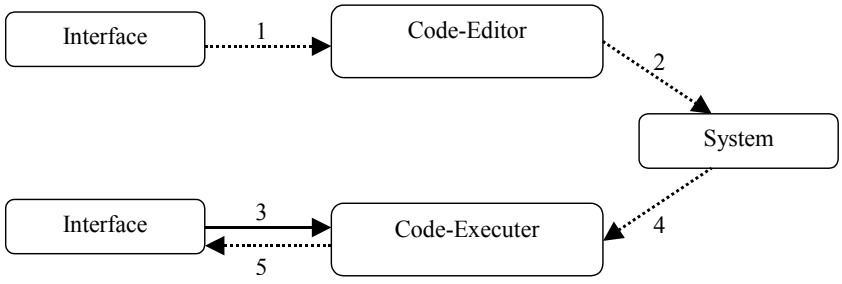


Figure C.62 Sequence diagram for the first application with Command Aggregation mechanism

STEP 3. Abstraction of the design solution for Command Aggregation

- Solution:
  - Diagram:



  - Participants:
    - Interface: it sends a data set (1) corresponding to a given command, as well as the program code associated with the command to be created, to Code-Editor. Additionally, if a command is to be executed, it sends the name of the previously created command (3) to the Code-executer. When the command has been executed, it will receive the result of the command or error, if it cannot be executed, (5) from Code-Executer.
    - Code-Editor: it receives the name of the command to be created and the program code to be associated with the command (1) from the interface, which it sends to system for storage (2).
    - Code-Executer: it receives a previously created command (3) from the interface. It asks the system for the program code to be executed (4) and executes this code. Also it sends the result of executing this command to the interface through (5).
    - System: it receives the name of the command as well as the associated program code (3). It also sends the program code associated with a command to the Code-executer when it is requested for execution (4).

C.20 Actions for Multiple Objects First Iteration

STEP 1. Design solution without Actions for Multiple Objects

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

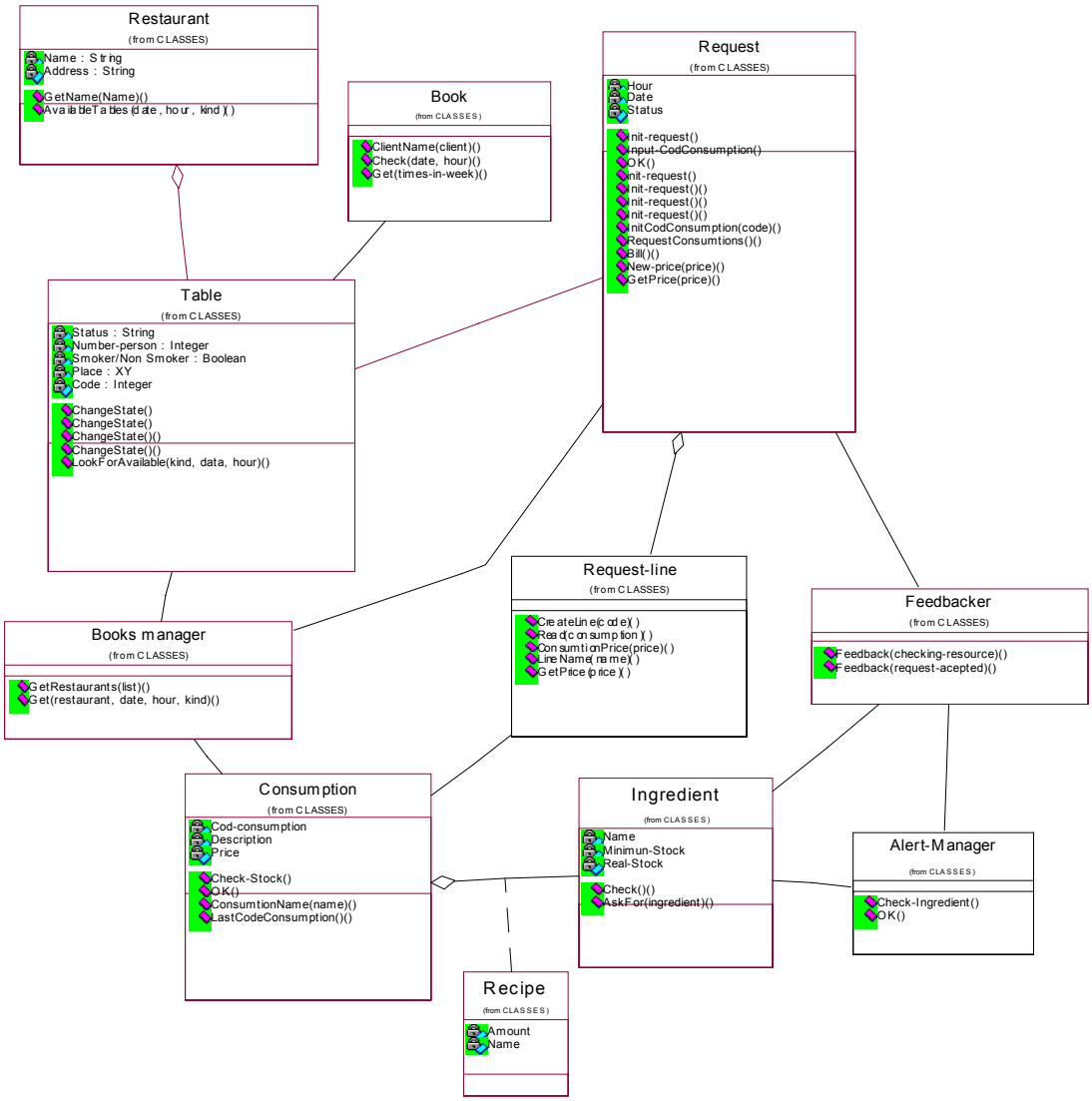


Figure C.63 Class diagram for the first application without Actions for Multiple Objects mechanism

STEP 2. Design solution with Actions for Multiple Objects

Requirement: the cook selects several ingredients and requests restocking.

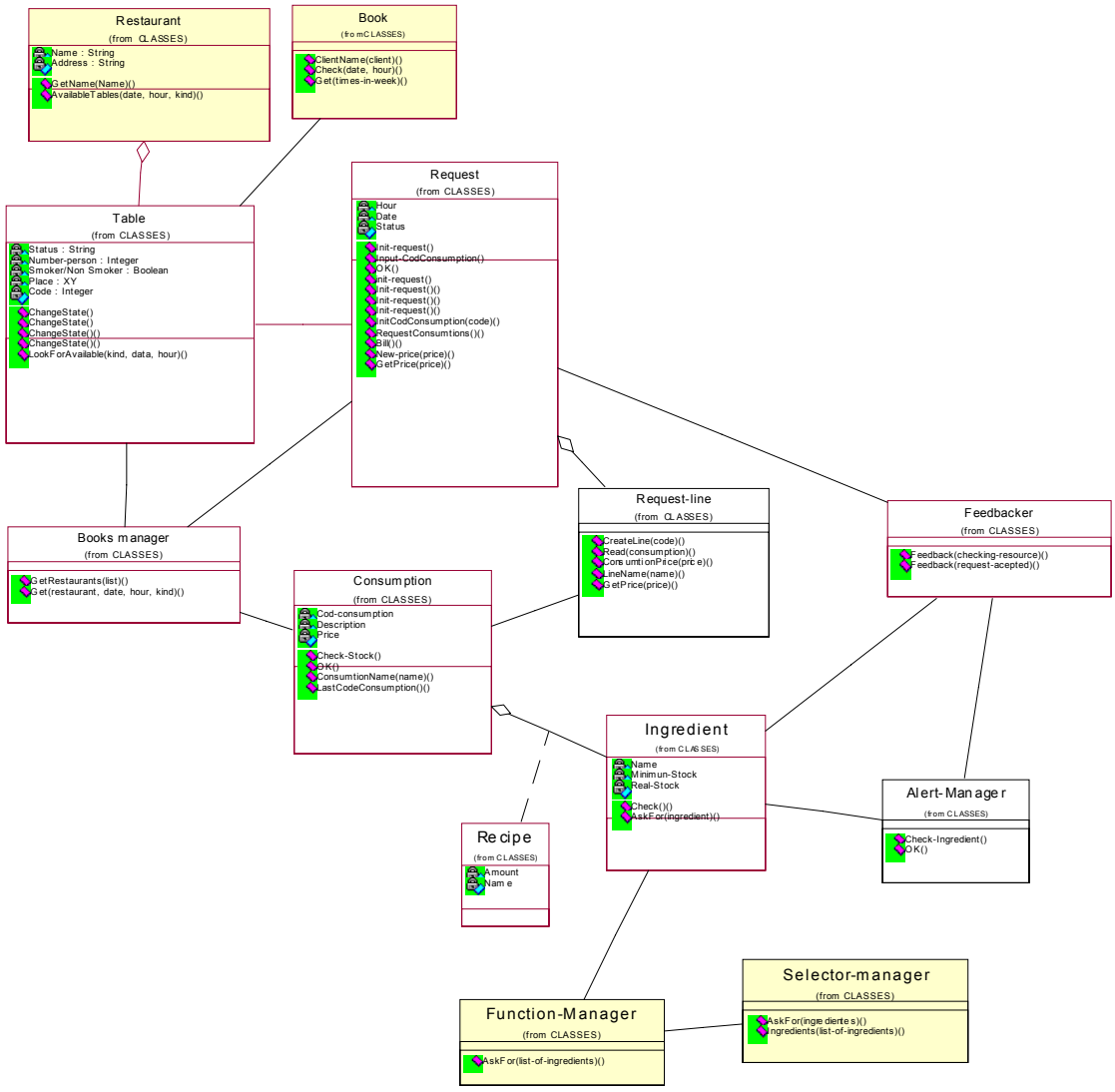


Figure C.64 Class diagram for the first application with Actions for Multiple Objects mechanism

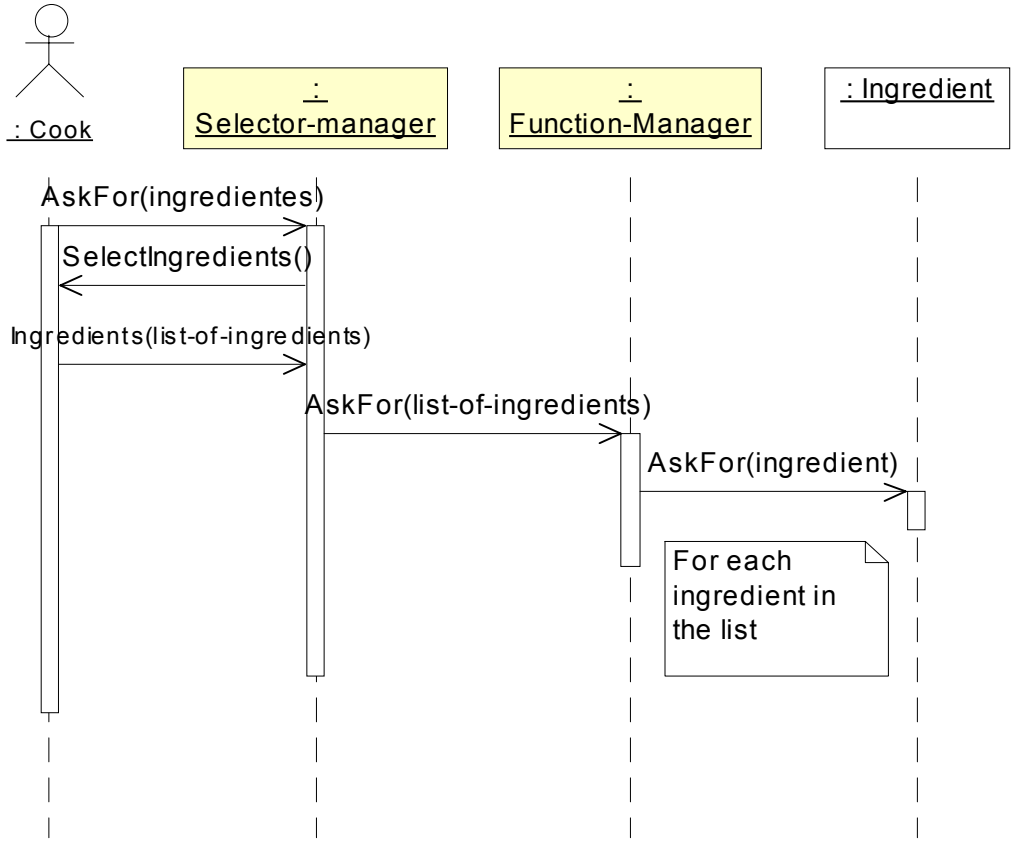
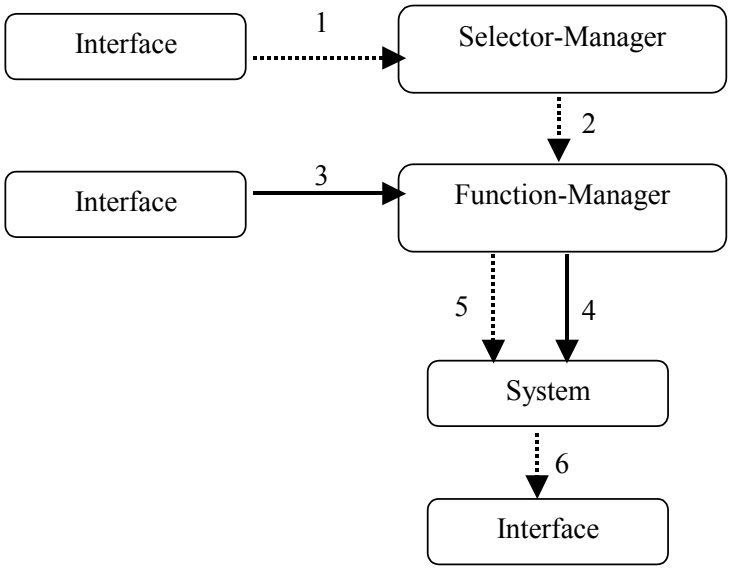


Figure C.65 Sequence diagram for the first application with Actions for Multiple Objects mechanism

STEP 3. Abstraction of the design solution for Actions for Multiple Objects

- **Solution:**
  - Diagram:



  - Participants:

- Interface: it sends the set of objects selected by the user from the interface to Selector-manager in (1). Additionally, it sends the function to be executed in (3). If, after the requested operation has been executed, the user is to be informed of the result of the operation, the respective data are sent to the interface in (6).
- Selector-manager: this component receives the set of elements on which to operate in (1). Additionally, it sends the set of objects on which the system is to operate to function-manager in (2).
- Function-manager: it receives the operation to be executed in (3) and receives the set of objects on which the system is to operate in (2).
- System: it receives the function to be executed (4) and the list of objects on which the specified function is to be executed in (5). Additionally, it sends the result of the executed function to the interface in (6).

ANNEX D: PHASE 1 SECOND ITERATION: THE AMUSEMENT PARK SYSTEM CONTROL CASE

D.1 Reusing Information Second Iteration

STEP 1. Design models without Reusing Information

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

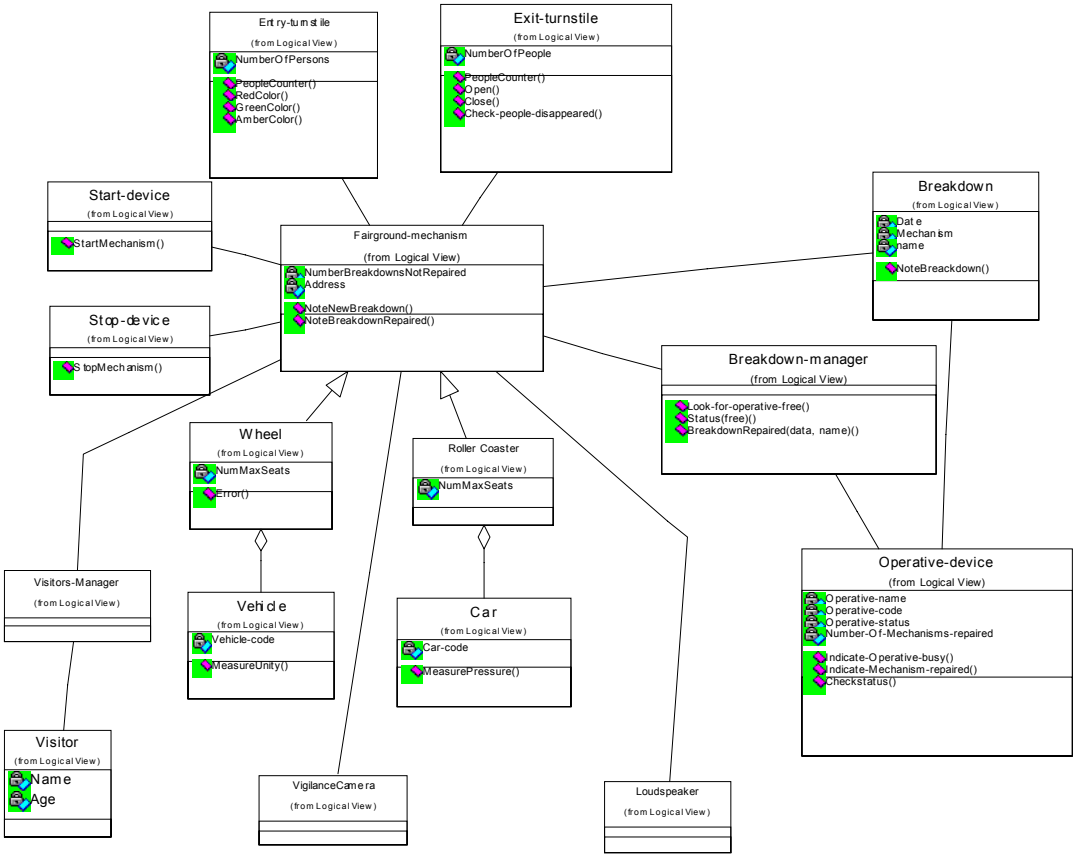


Figure D.1 Class diagram for the second application without Reusing Information mechanism

STEP 2. Design models with Reusing Information

Requirement: a father is entering the description of one of his sons as a park visitor and wants to copy the son's description because he has twins.

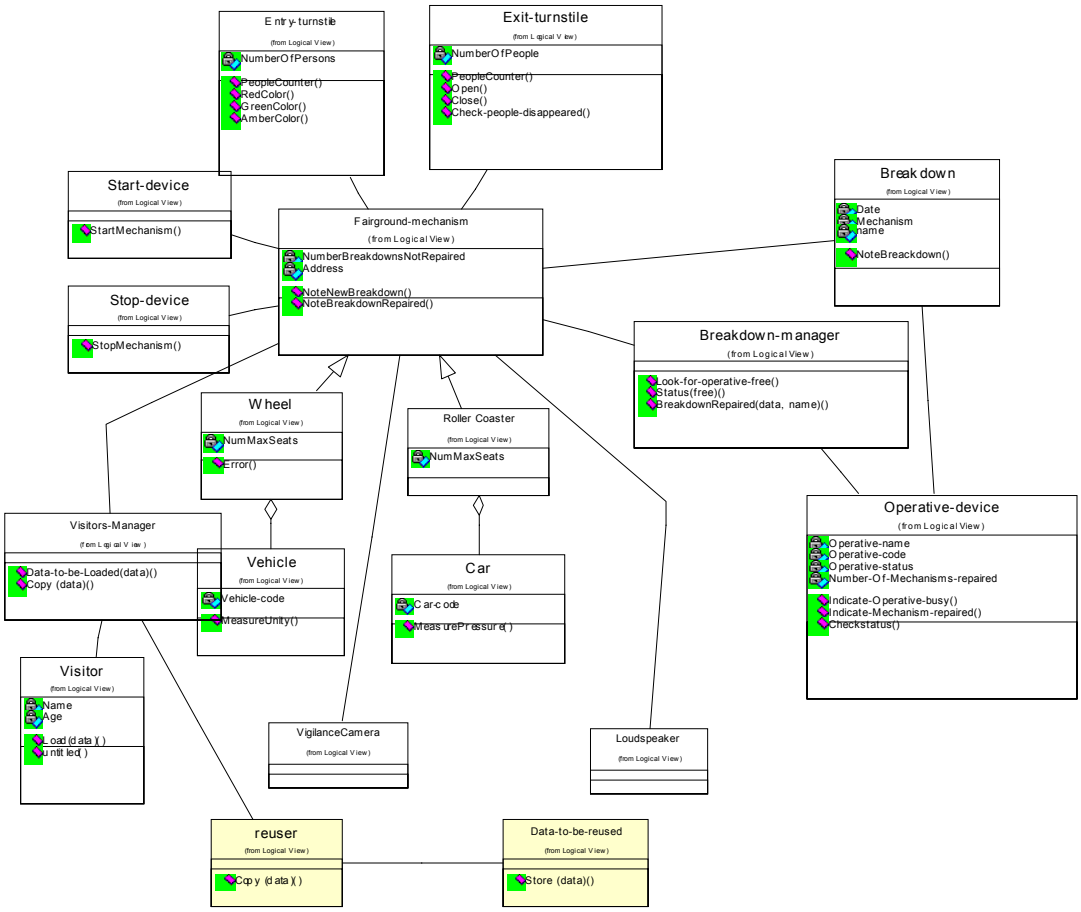


Figure D.2 Class diagram for the second application with Reusing Information mechanism

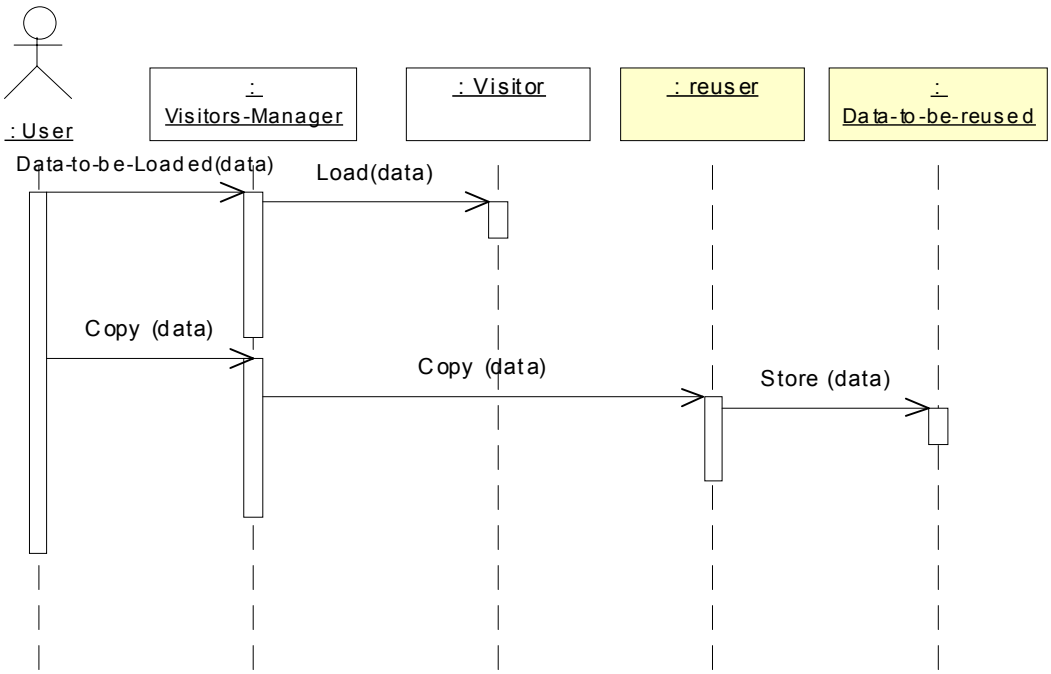


Figure D.3 Sequence diagram for the second application with Reusing Information mechanism

STEP 3. Abstraction of the design solution for Reusing Information

There are no modifications to the above generalisation presented in the first iteration.

D.2 Standard Help Second Iteration

STEP 2. Design solution without Standard Help

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

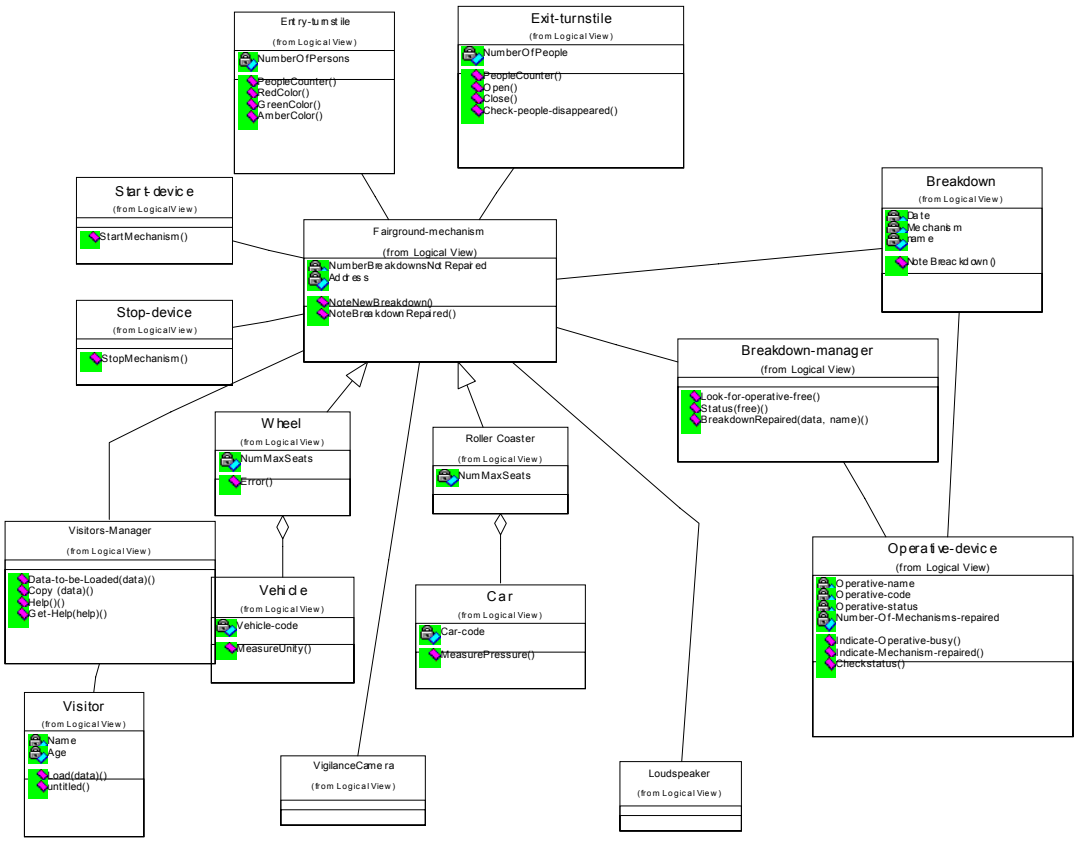


Figure D.4 Class diagram for the second application without Standard Help mechanism

STEP 2. Design solution with Standard Help

Requirement: The user wants to enter his/her particulars in the visitor terminal and asks the system for help on terminal use.

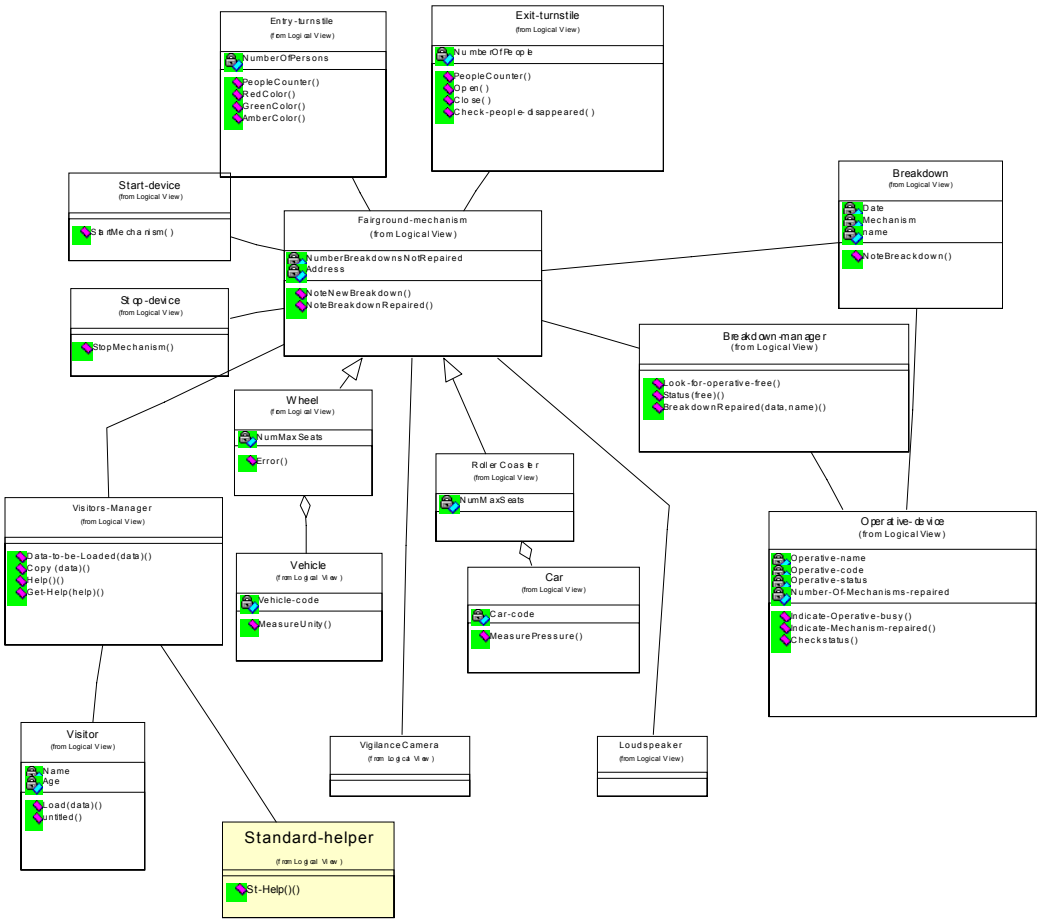


Figure D.5 Class diagram for the second application with Standard Help mechanism

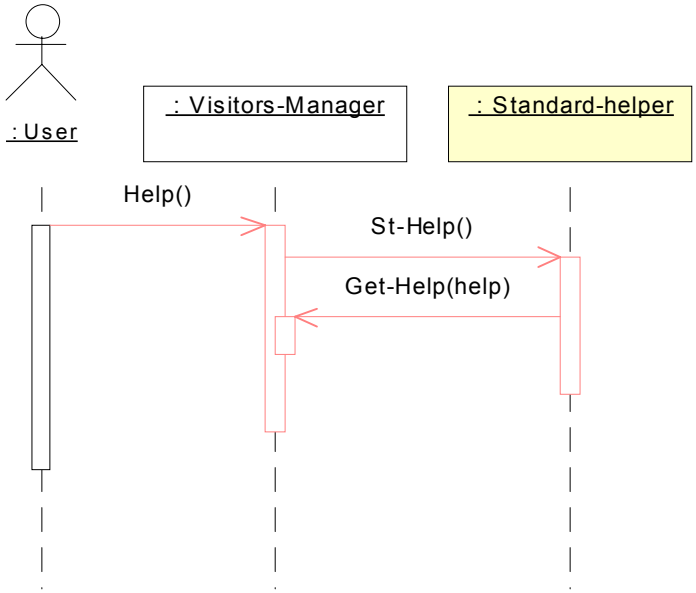


Figure D.6 Sequence diagram for the second application with Standard Help mechanism

STEP 3. Abstraction of the design solution for Standard Help

There are no modifications to the above generalisation presented in the first iteration.

D.3 Tour Second Iteration

STEP 1. Design models without Tour

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

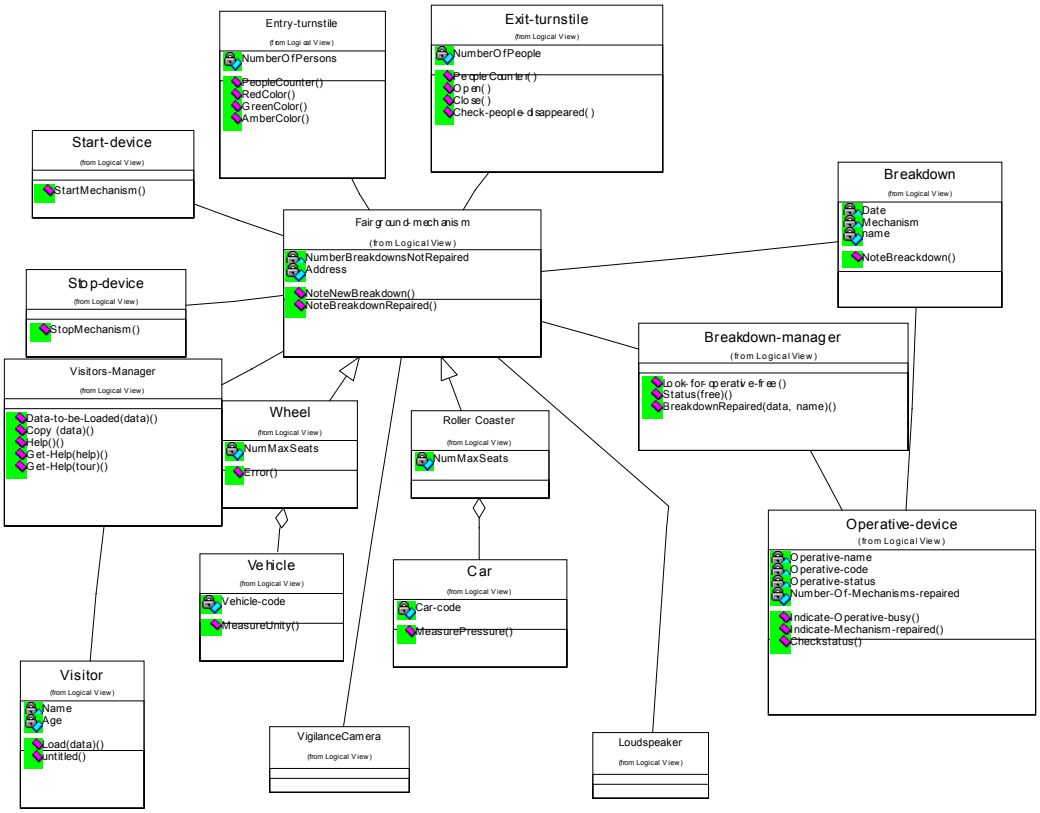


Figure D.7 Class diagram for the second iteration without Tour mechanism

STEP 2. Design models with Tour

Requirement: the user pushes Guided Help button.

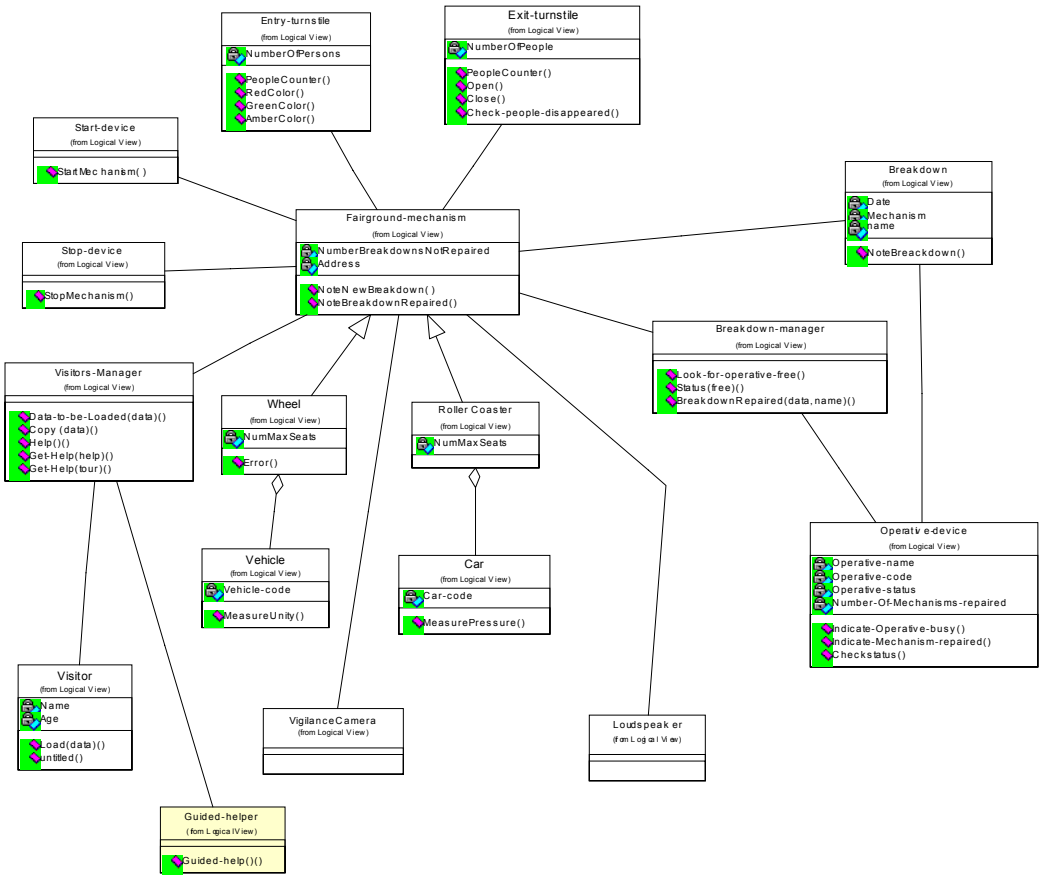


Figure D.8 Class diagram for the second iteration with Tour mechanism

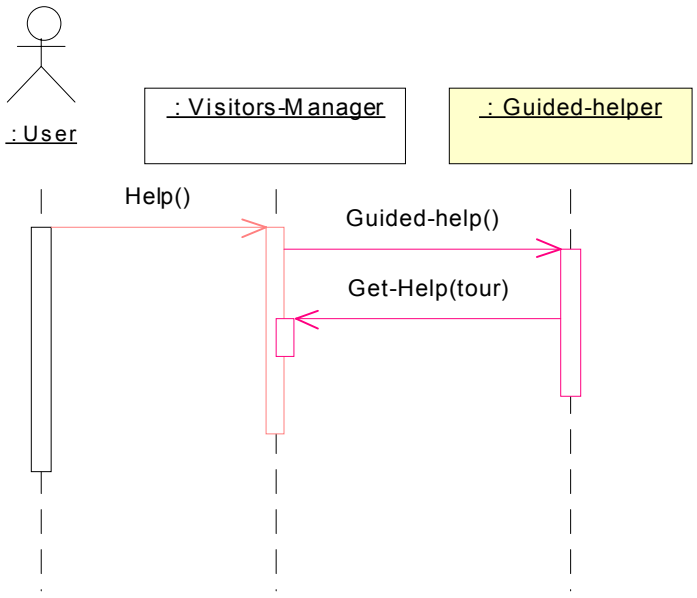


Figure D.9 Sequence diagram for the second iteration with Tour mechanism

STEP 3. Abstraction of the design solution for Tour

There are no modifications to the above generalisation presented in the first iteration.

D.4 Different Languages Second Iteration

STEP 1. Design models without Different Languages

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

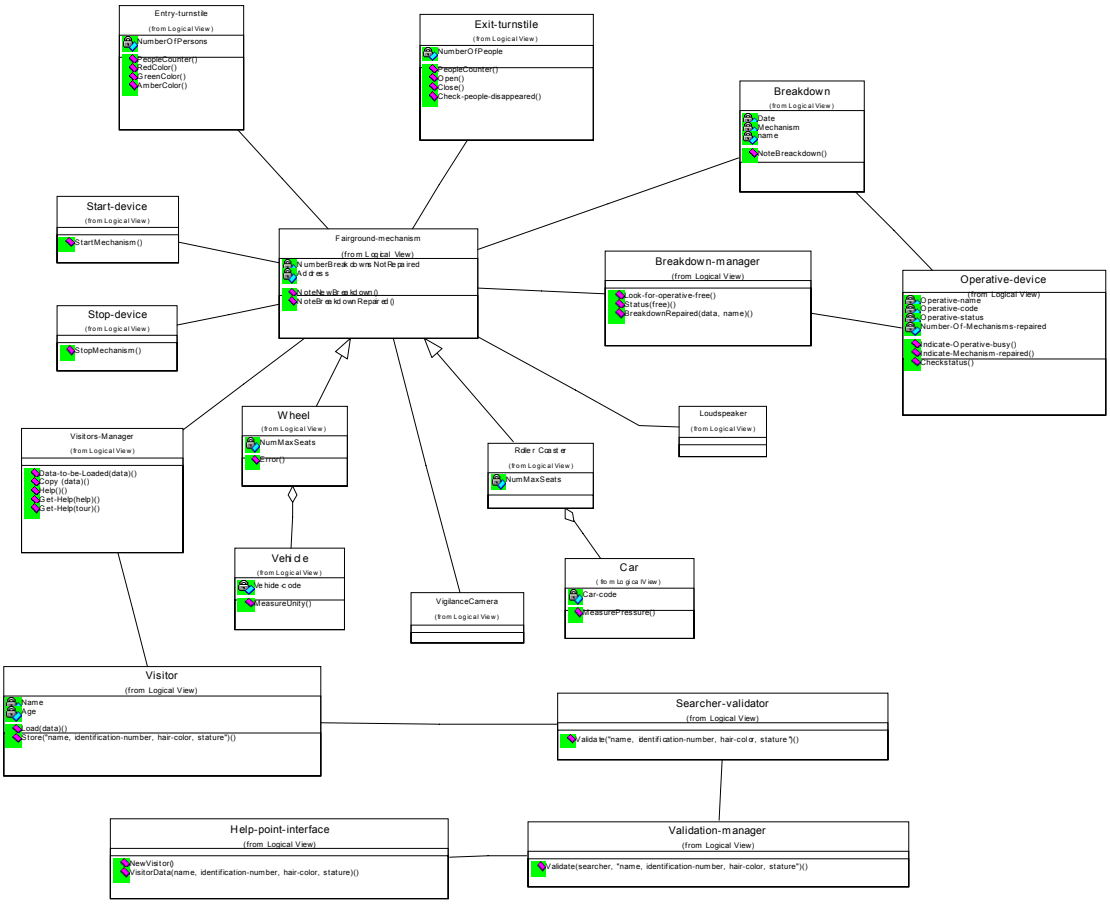


Figure D.10 Class diagram for the second application with Different Languages mechanism

STEP 2. Design models with Different Languages

Requirement: the park visitor enters the details of the person he wants to register in any language, and the system is capable of translating it to an internal exchange language so that the surveillance system later operates identically when searching for a given subject, irrespective of the language in which the subject details were entered.

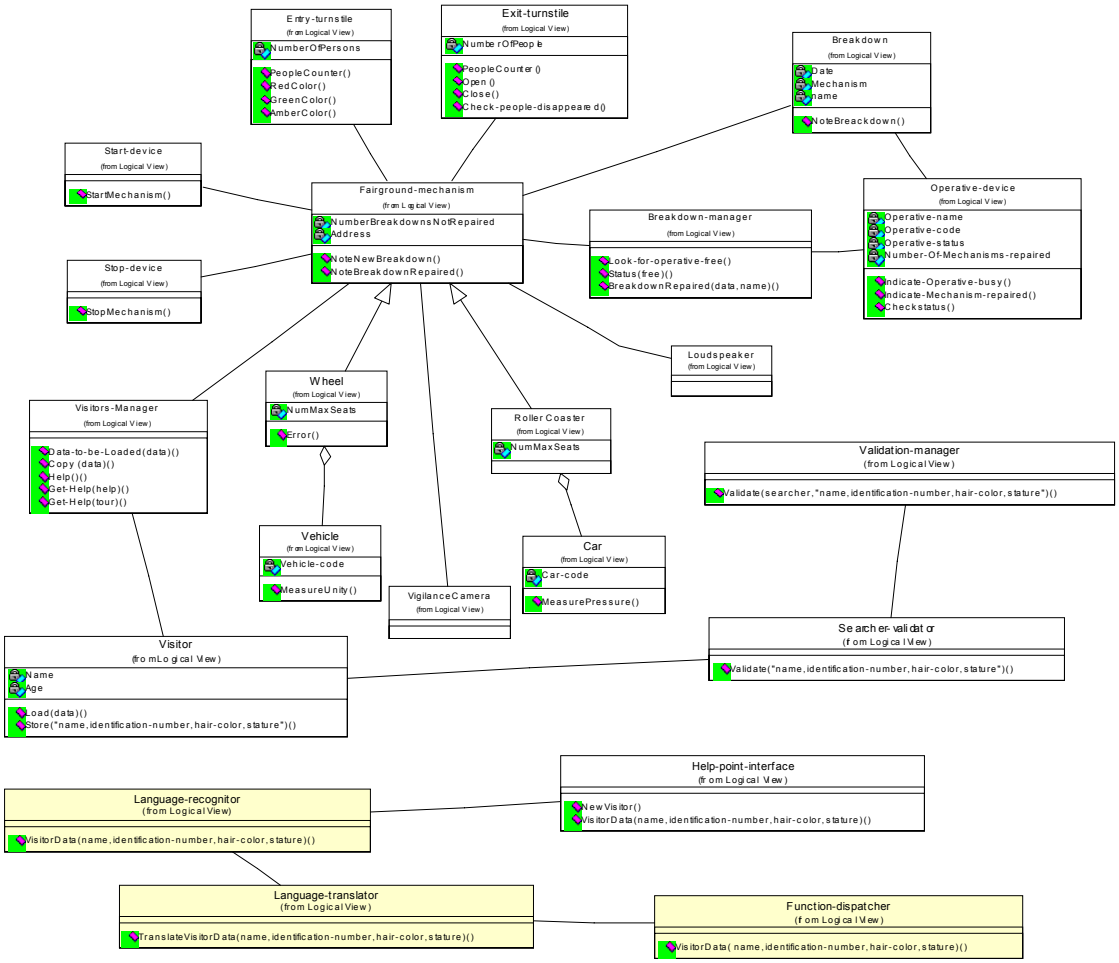


Figure D.11 Class diagram for the second application with Different Languages mechanism

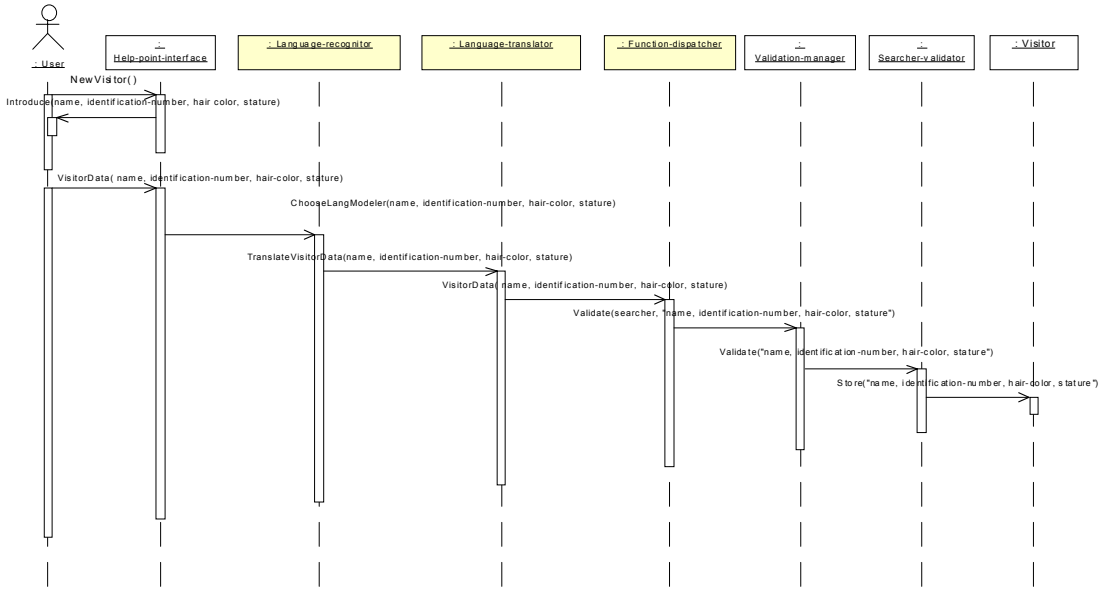


Figure D.12 Sequence diagram for the second application with Different Languages mechanism

STEP 3. Abstraction of the design solution for Different Languages

There are no modifications to the above generalisation presented in the first iteration.

D.5 Different Access Methods Second Iteration

STEP 1. Design models without Different Access Methods

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

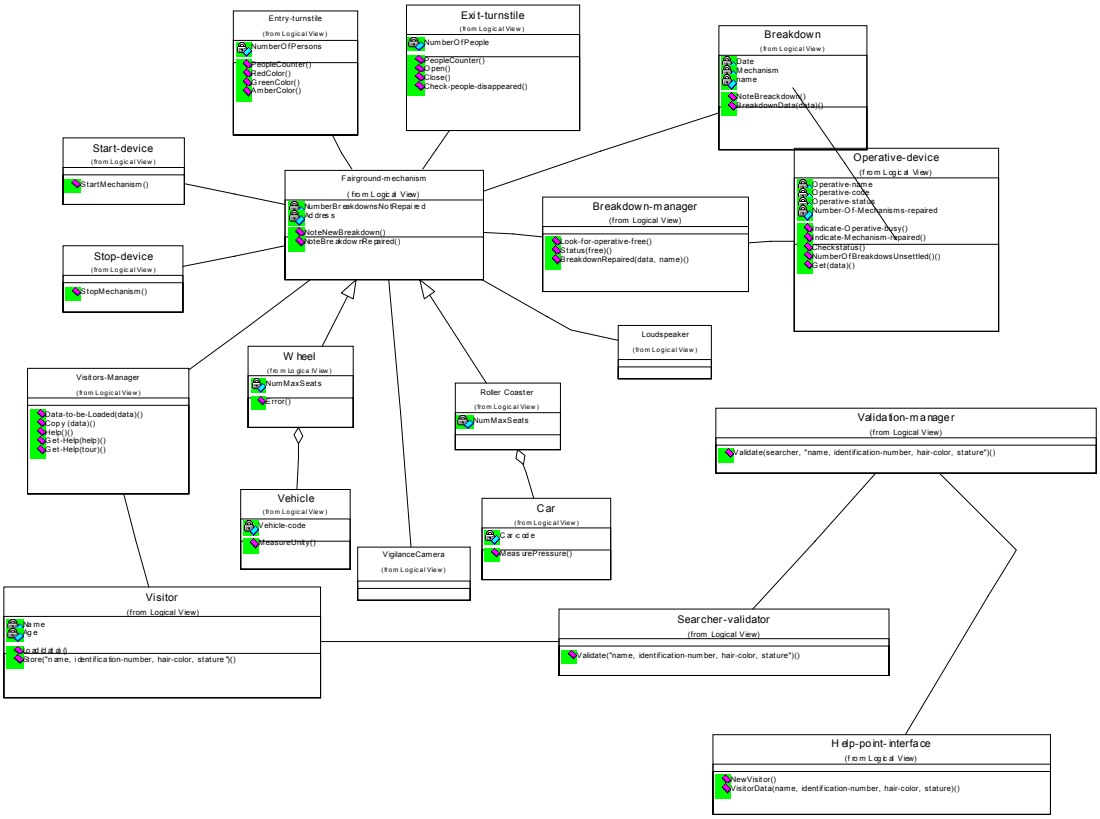


Figure D.13 Class diagram for the second application without Different Access Methods mechanism

STEP 2. Design models with Different Access Methods

Requirement: the operator can ask the operator device how many faults are pending repair, which he can do by simply speaking, that is, by voice, and the device also answers by voice. Additionally, the system must be capable of recognising what the operator says and who the operator is.

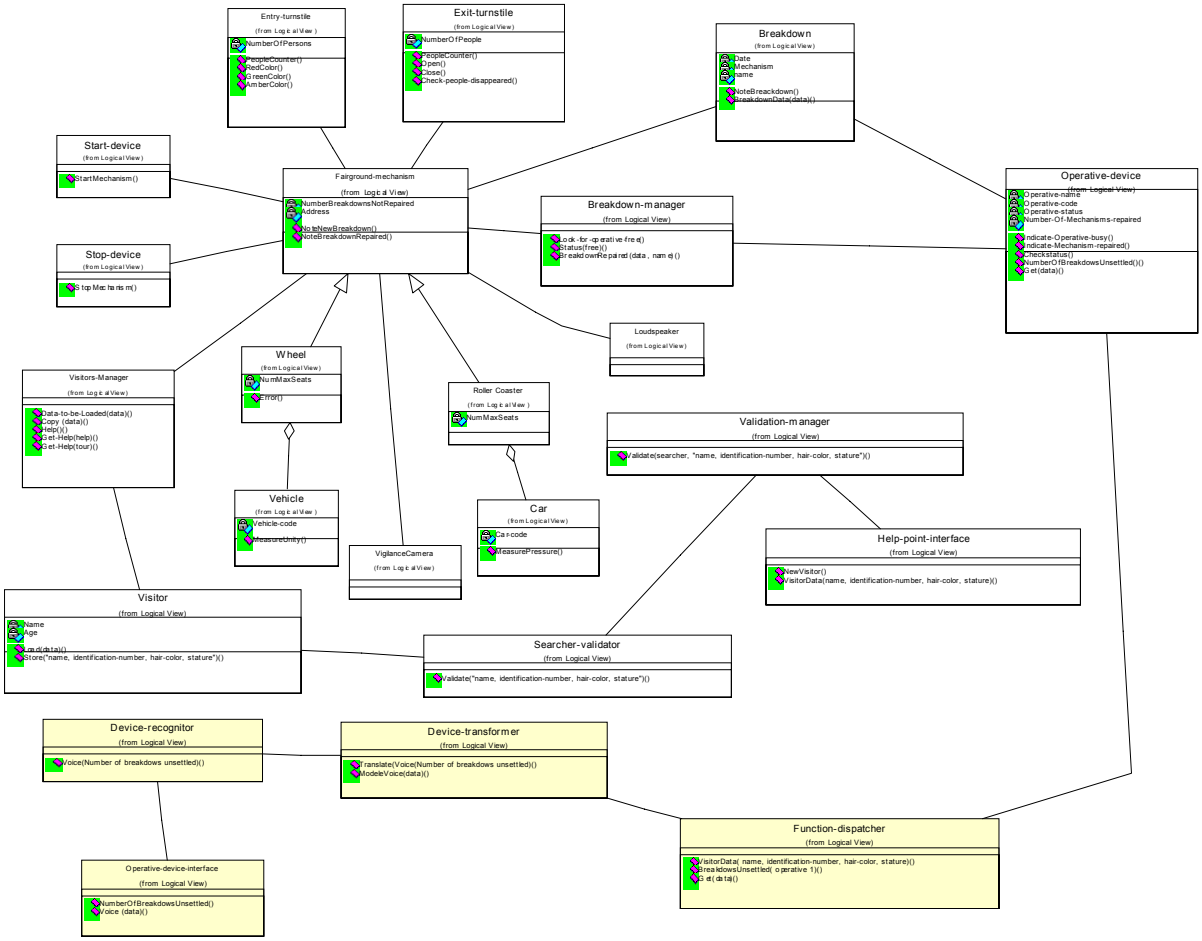


Figure D.14 Class diagram for the second application with Different Access Methods mechanism

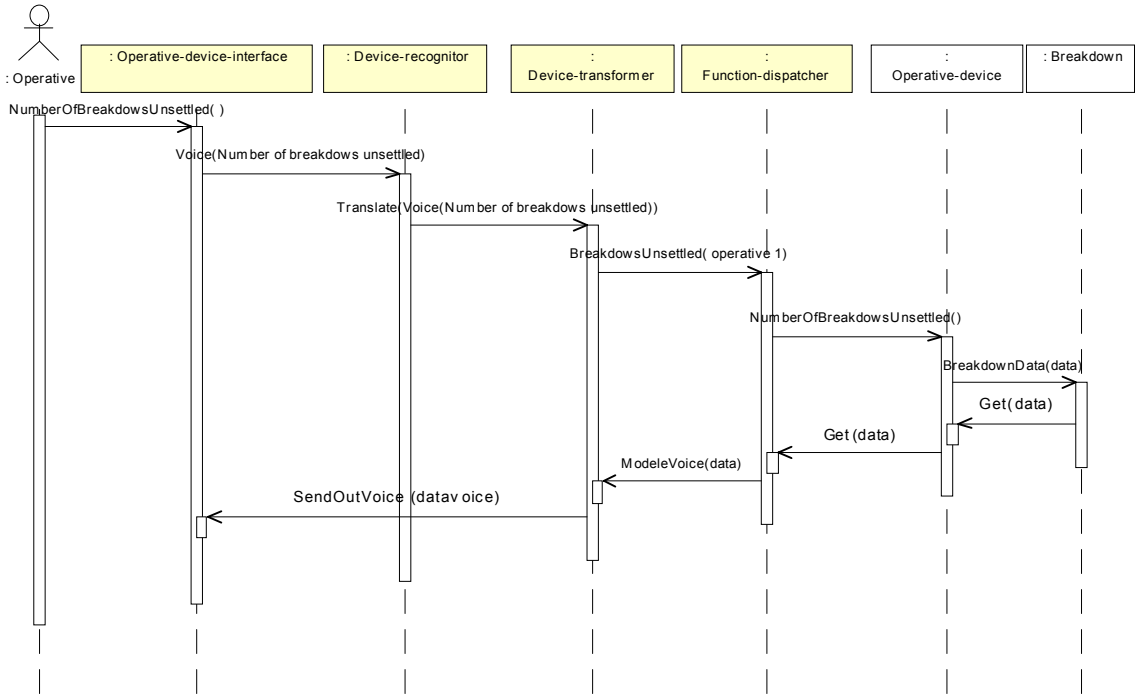


Figure D.15 Sequence diagram for the second application with Different Access Methods mechanism

STEP 3. Abstraction of the design solution for Different Access Methods

There are no modifications to the above generalisation presented in the first iteration.

D.6 Alerts Second Iteration

STEP 1. Design models without Alerts

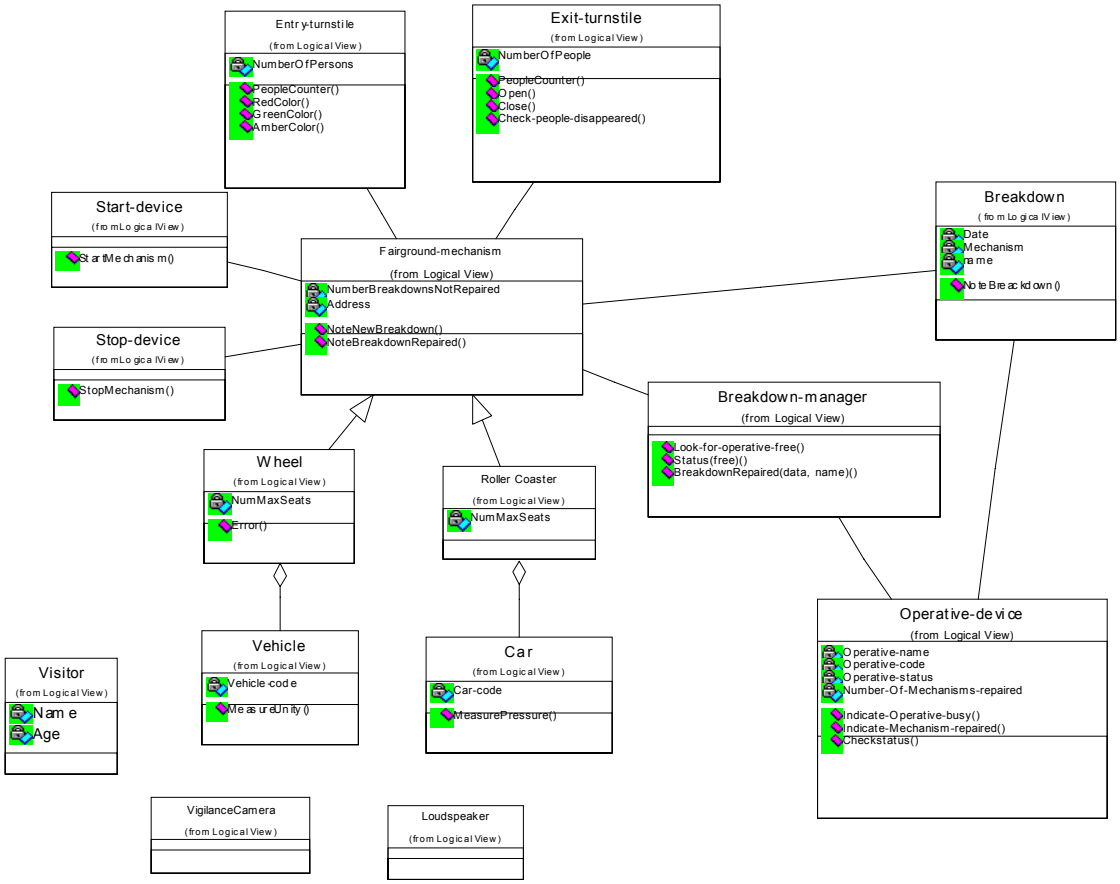


Figure D.16 Class diagram for the second application without Alerts mechanism

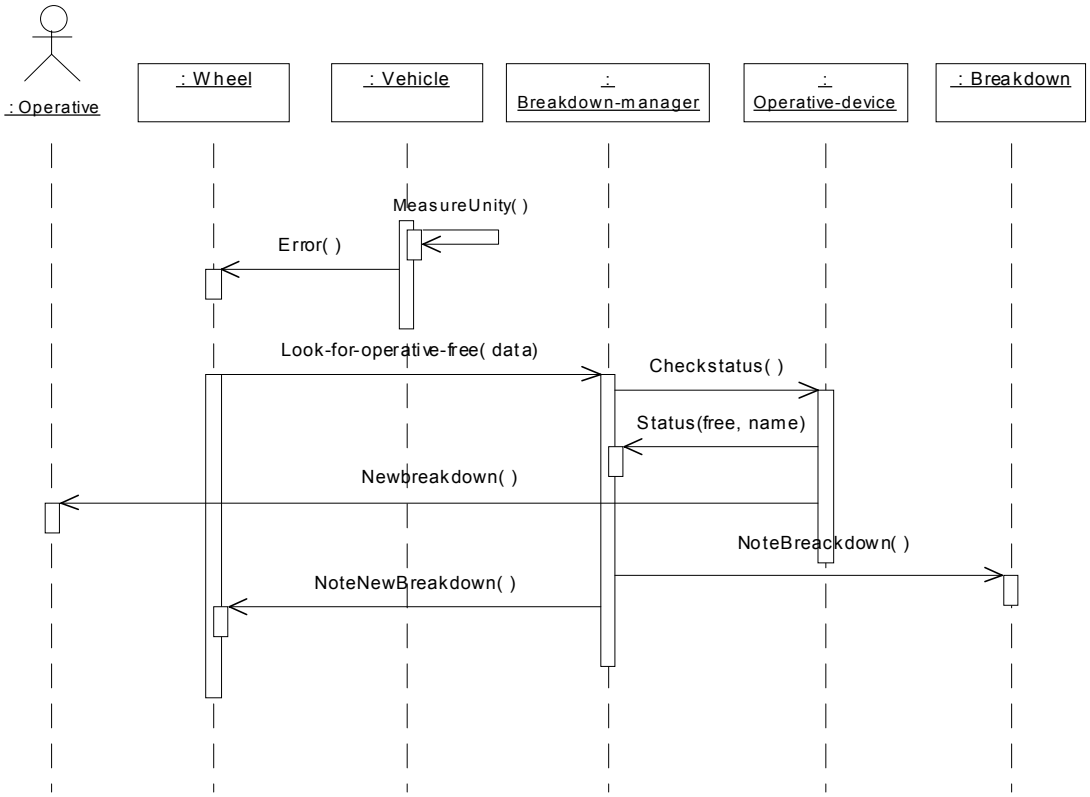


Figure D.17 Sequence diagram for the second application without Alerts mechanism

STEP 2. Design models with Alerts

Requirement: add a control of alerts within the park.

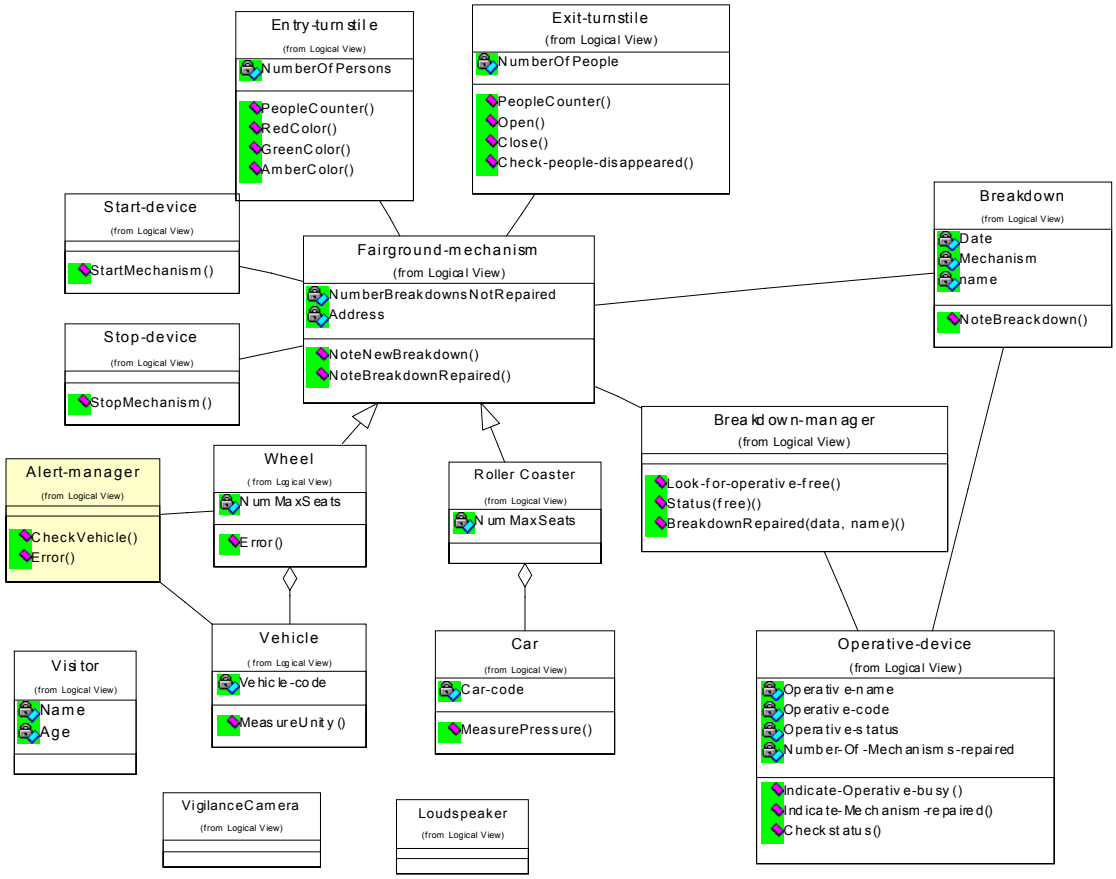


Figure D.18 Class diagram for the second application with Alerts mechanism

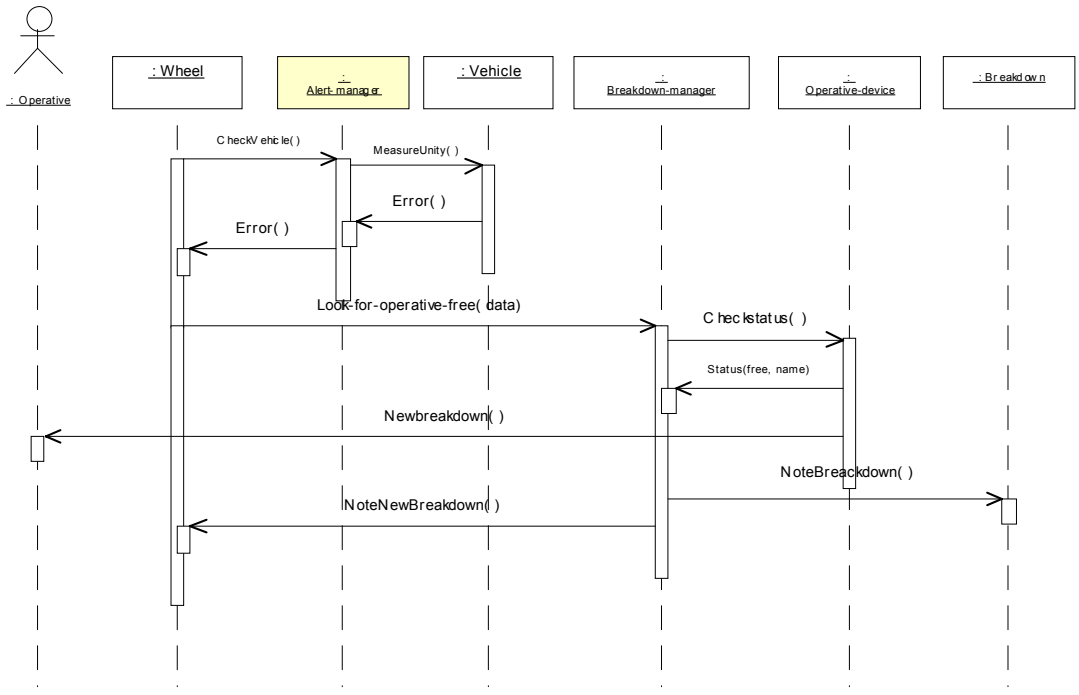


Figure D.19 Sequence diagram for the second application with Alerts mechanism

STEP 3. Abstraction of the design solution for Alerts

There are no modifications to the above generalisation presented in the first iteration.

D.7 Status Indication Second Iteration

STEP 1. Design models without Status Indication

I In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

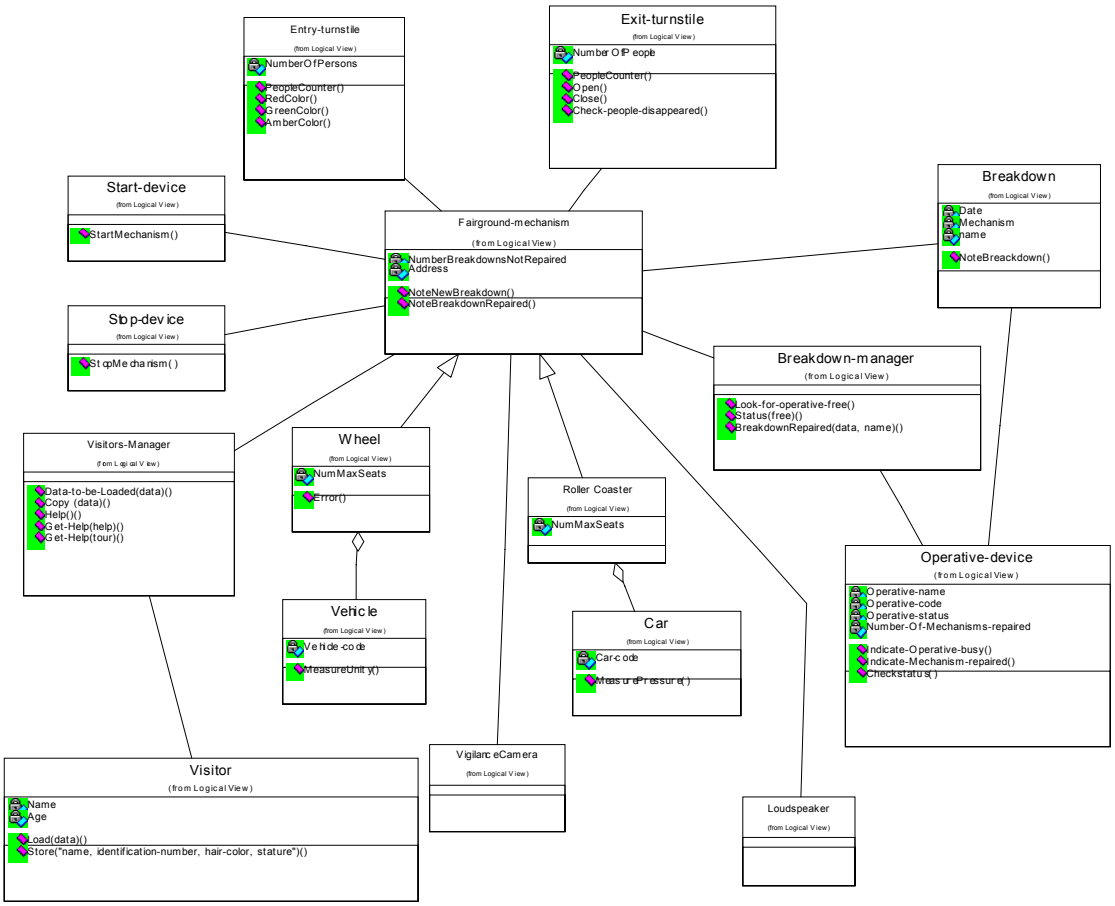


Figure D.20 Class diagram for the second application without Status Indication mechanism

STEP 2. Design models with Status Indication

Requirement: the operator must be informed of what the system is doing.

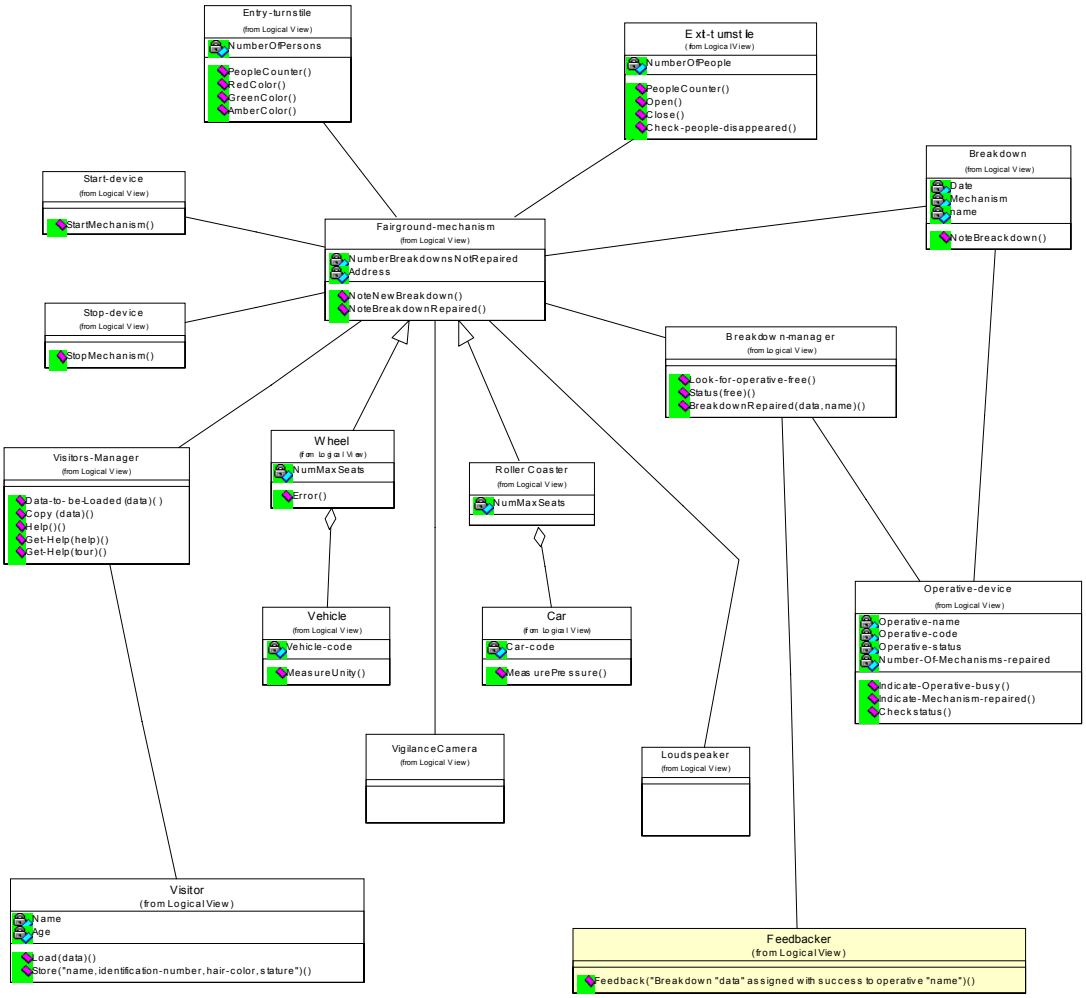


Figure D.21 Class diagram for the second application with Status Indication mechanism

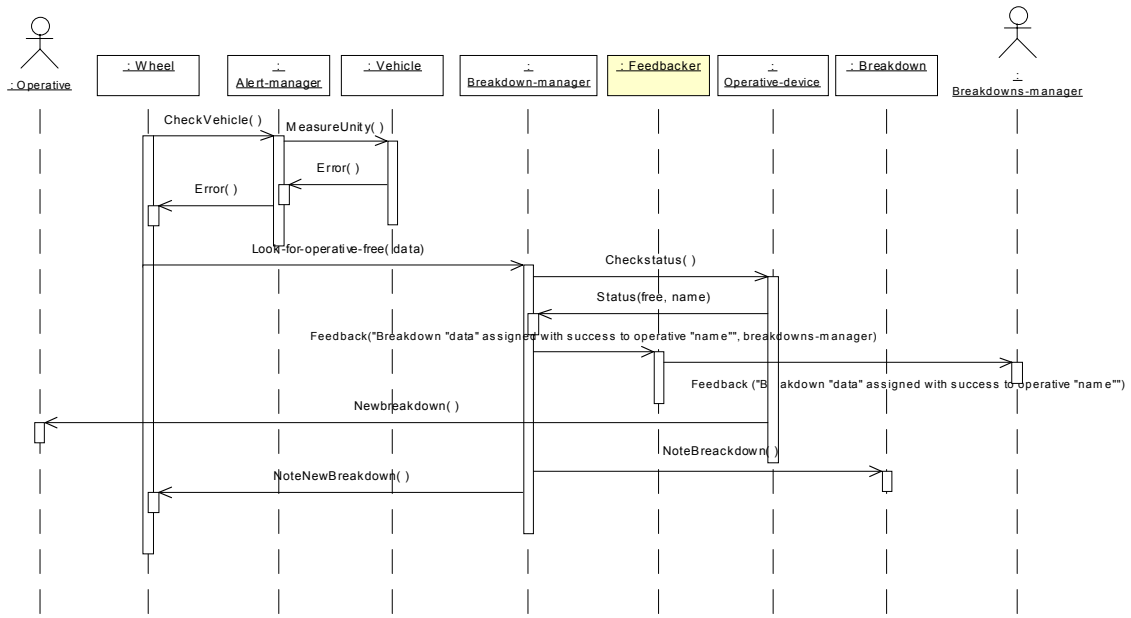


Figure D.22 Sequence diagram for the second application with Status Indication mechanism

STEP 3. Abstraction of the design solution for Status Indication

There are no modifications to the above generalisation presented in the first iteration.

D.8 History Logging Second Iteration

STEP 1. Design models without History Logging

Requirement: the user indicates that the ride has been successfully repaired.

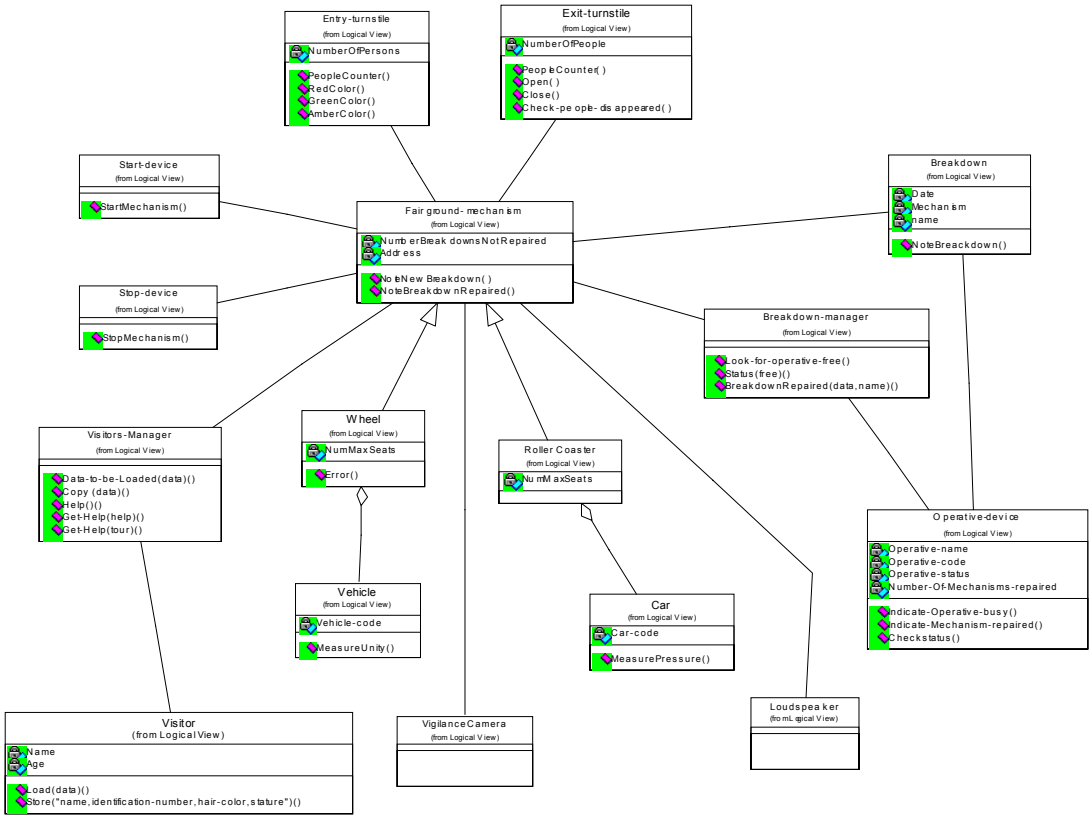


Figure D.23 Class diagram for the second application without History Logging mechanism

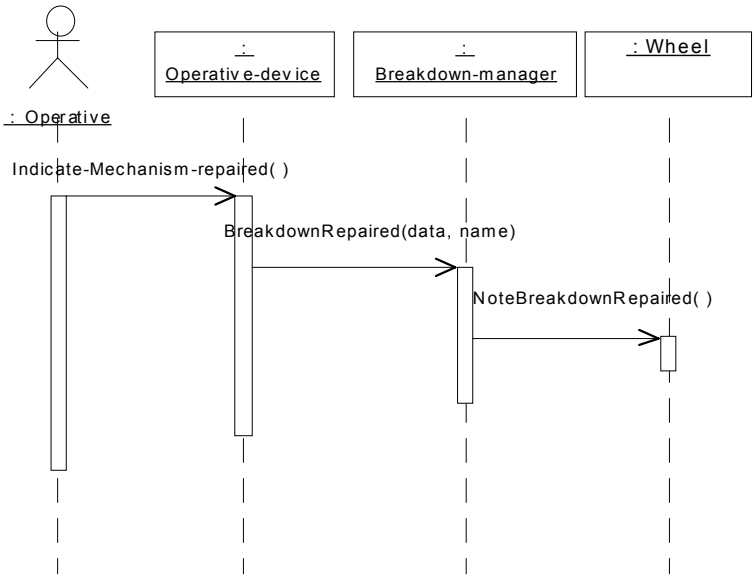


Figure D.24 Sequence diagram for the second application without History Logging mechanism

STEP 2. Design models with History Logging

Requirement: the user indicates that the ride has been successfully repaired and this notification is stored in the system.

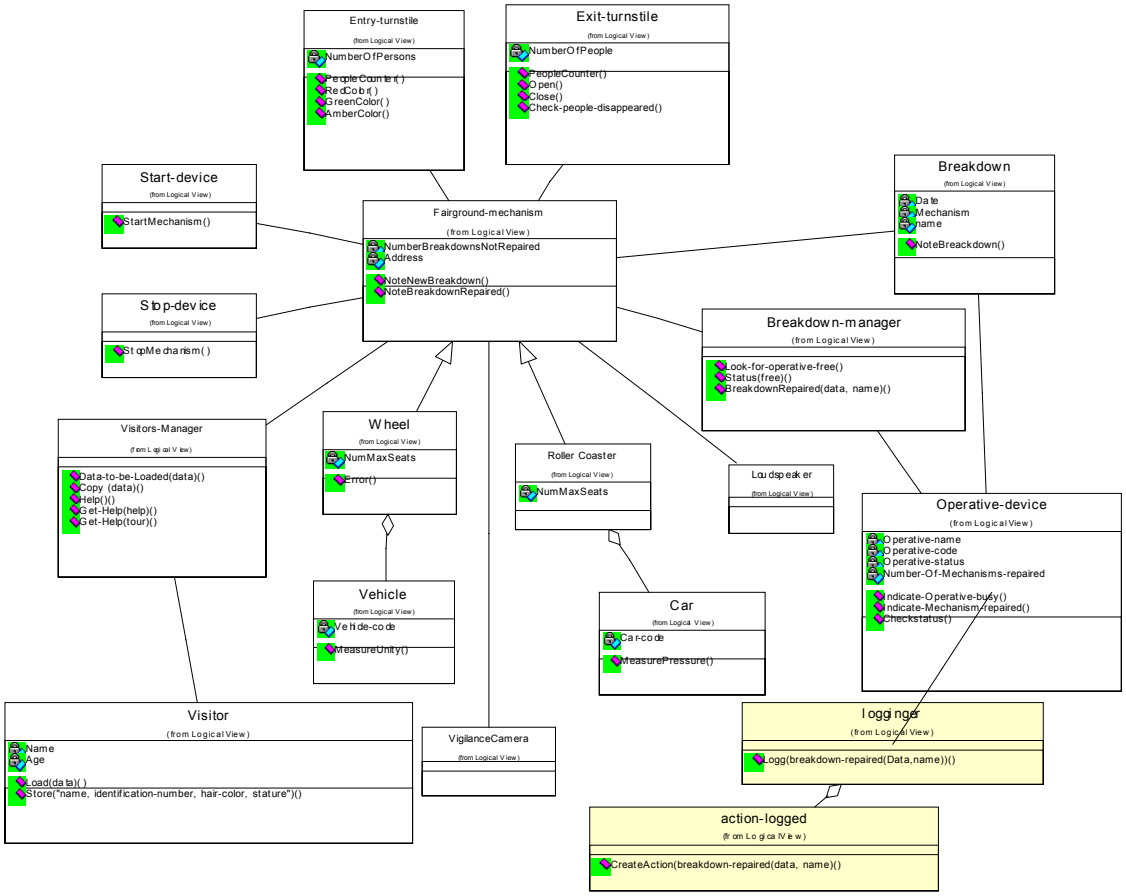


Figure D.25 Class diagram for the second application with History Logging mechanism

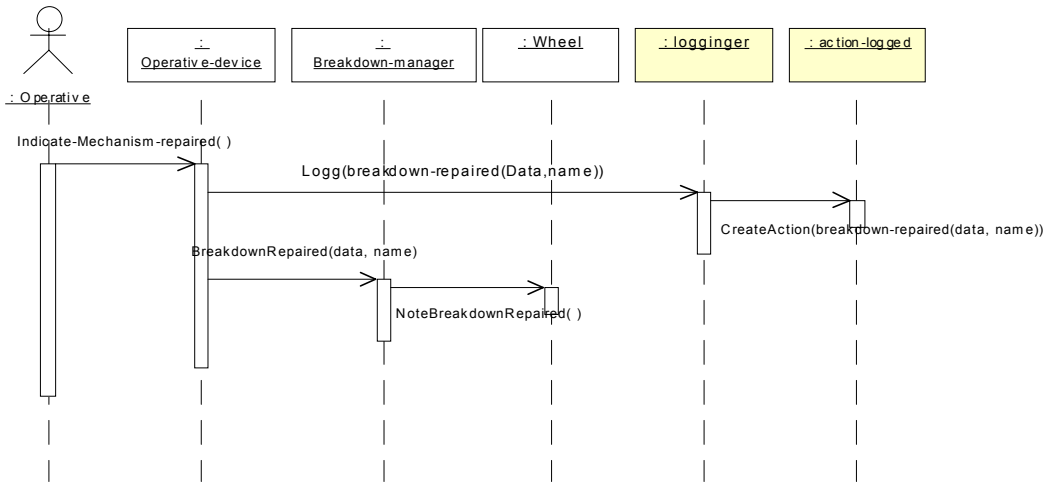


Figure D.26 Sequence diagram for the second application with History Logging mechanism

STEP 3. Abstraction of the design solution for History Logging

There are no modifications to the above generalisation presented in the first iteration.

D.9 Undo Second Iteration

In the case of the amusement park design, as it is essentially a control application where there is not much human intervention, it does not make sense to apply the Undo pattern, which explains why no second iteration appears for this pattern.

D.10 Form or Field Validation Second Iteration

STEP 1. Design models without Form or Field Validation

In this case, the interaction diagram does not appear because the "data validation" functionality is new.

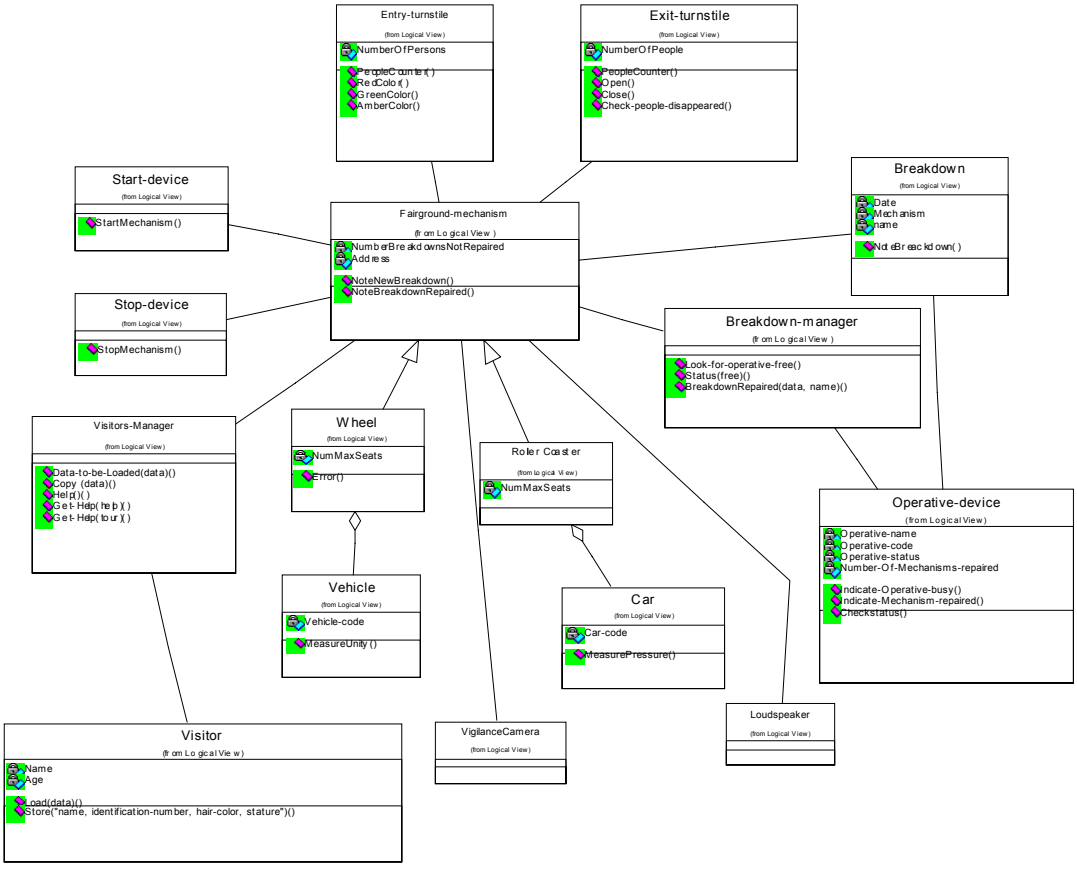


Figure D.27 Class diagram for the second application without Form or Field Validation mechanism

STEP 2. Design models with Form or Field Validation

Requirement: the visitor who wants to register correctly enters the data requested so that he can be later located in the park if he gets lost.

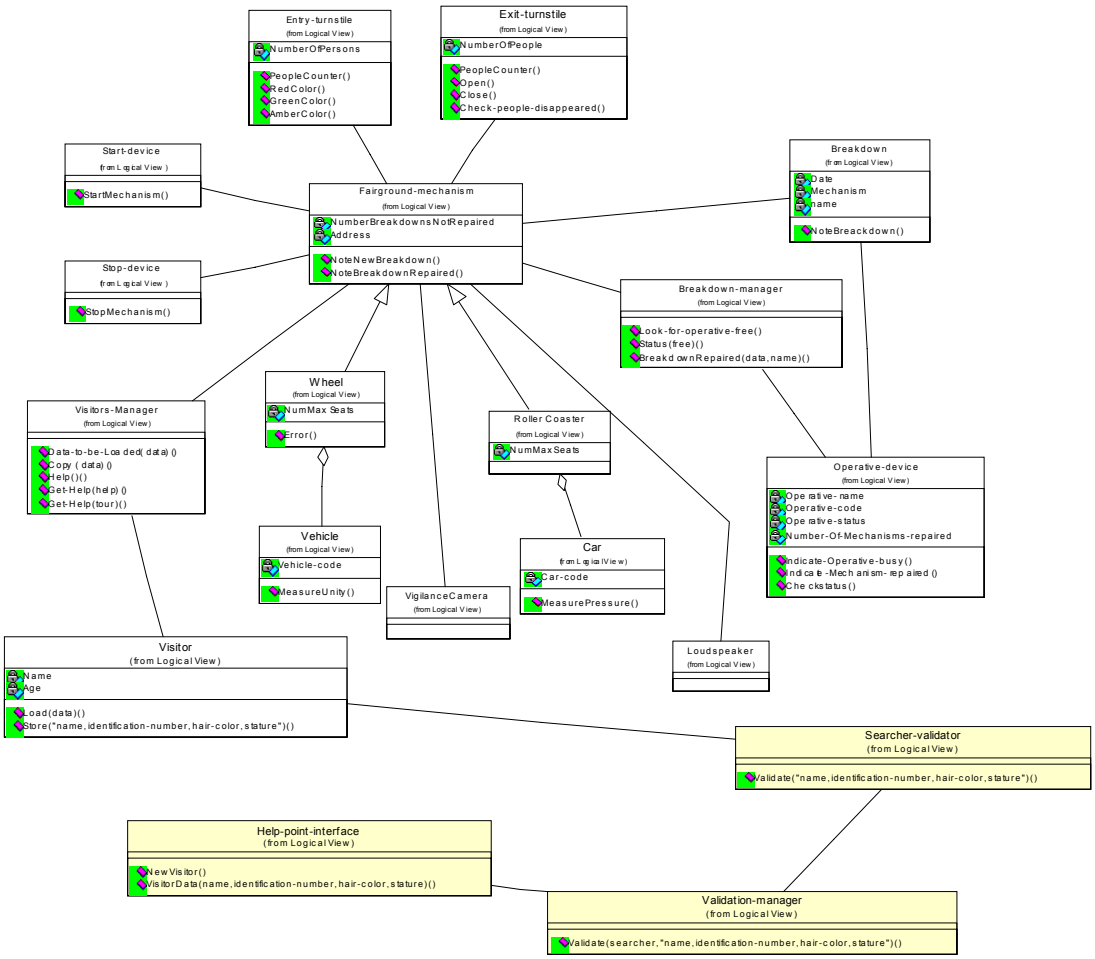


Figure D.28 Class diagram for the second application with Form or Field Validation mechanism

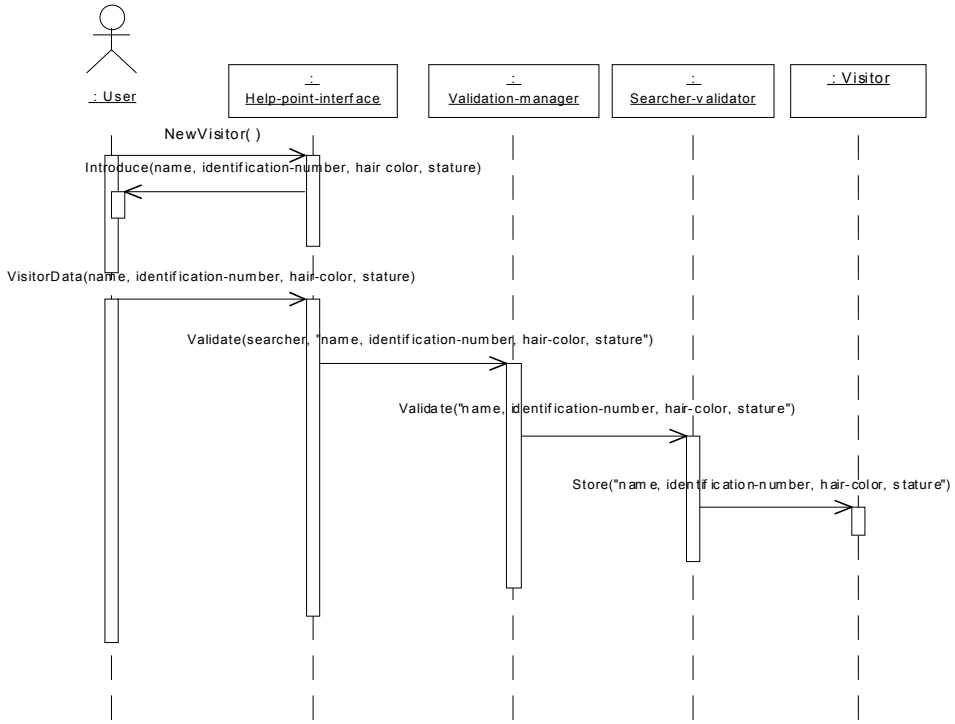


Figure D.29 Sequence diagram for the second application with Form or Field Validation mechanism

STEP 3. Abstraction of the design solution for Form or Field Validation

There are no modifications to the above generalisation presented in the first iteration.

D.11 Provision of Views Second Iteration

In the case of the amusement park design, as it is essentially a control application where there is not much human intervention, it makes not sense to apply the Provision of Views pattern, which explains why no second iteration appears for this pattern.

D.12 Workflow Model Second Iteration

STEP 1.1. Design models without Workflow Model

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

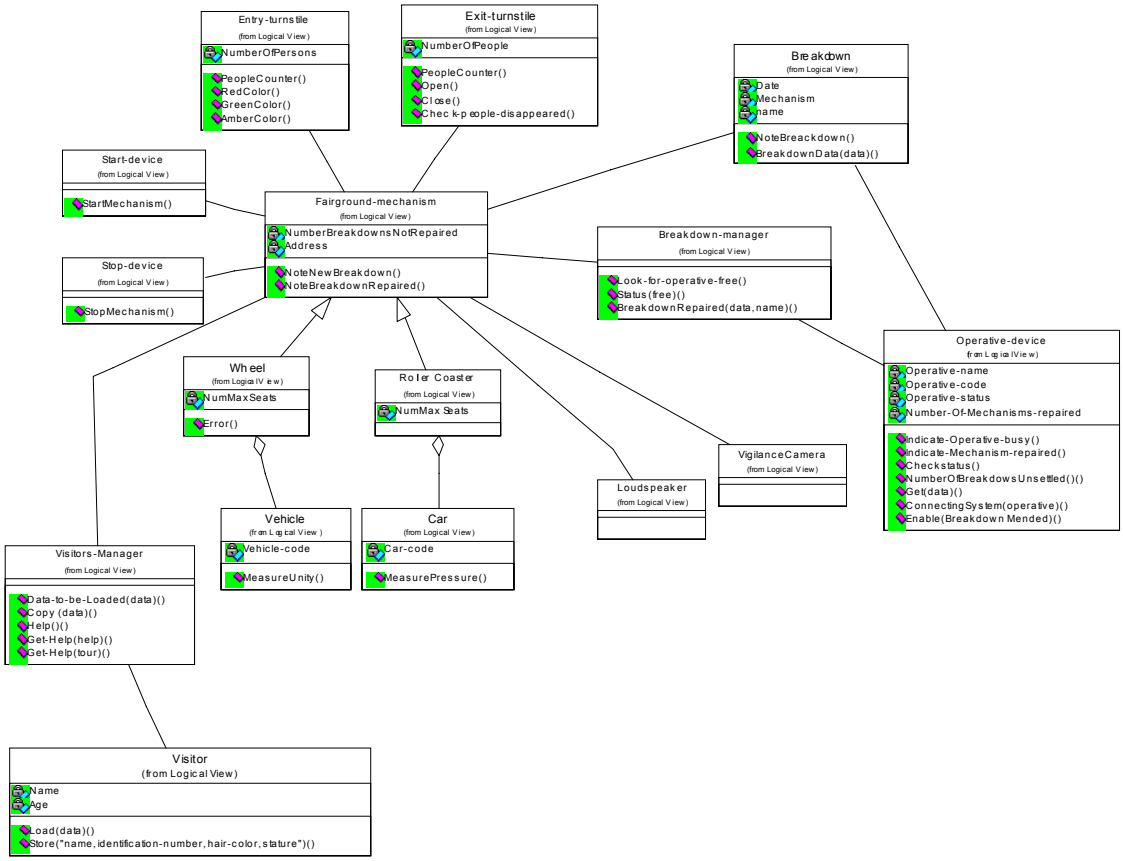


Figure D.30 Class diagram for the second application without Workflow model mechanism

STEP 1.2. Design models with Workflow Model

Requirement: when the operator connects to the system only the pending faults and the option of indicating repair completed appear.

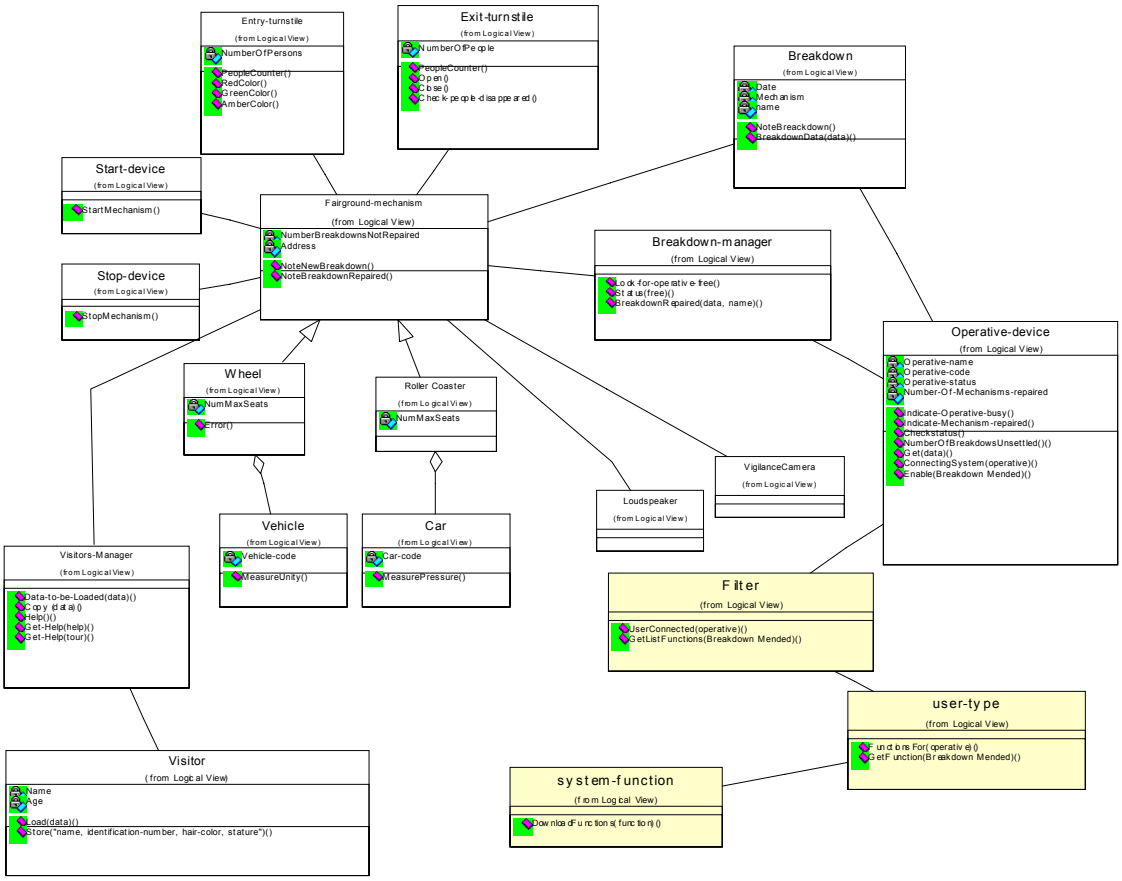


Figure D.31 Class diagram for the second application with Workflow model mechanism

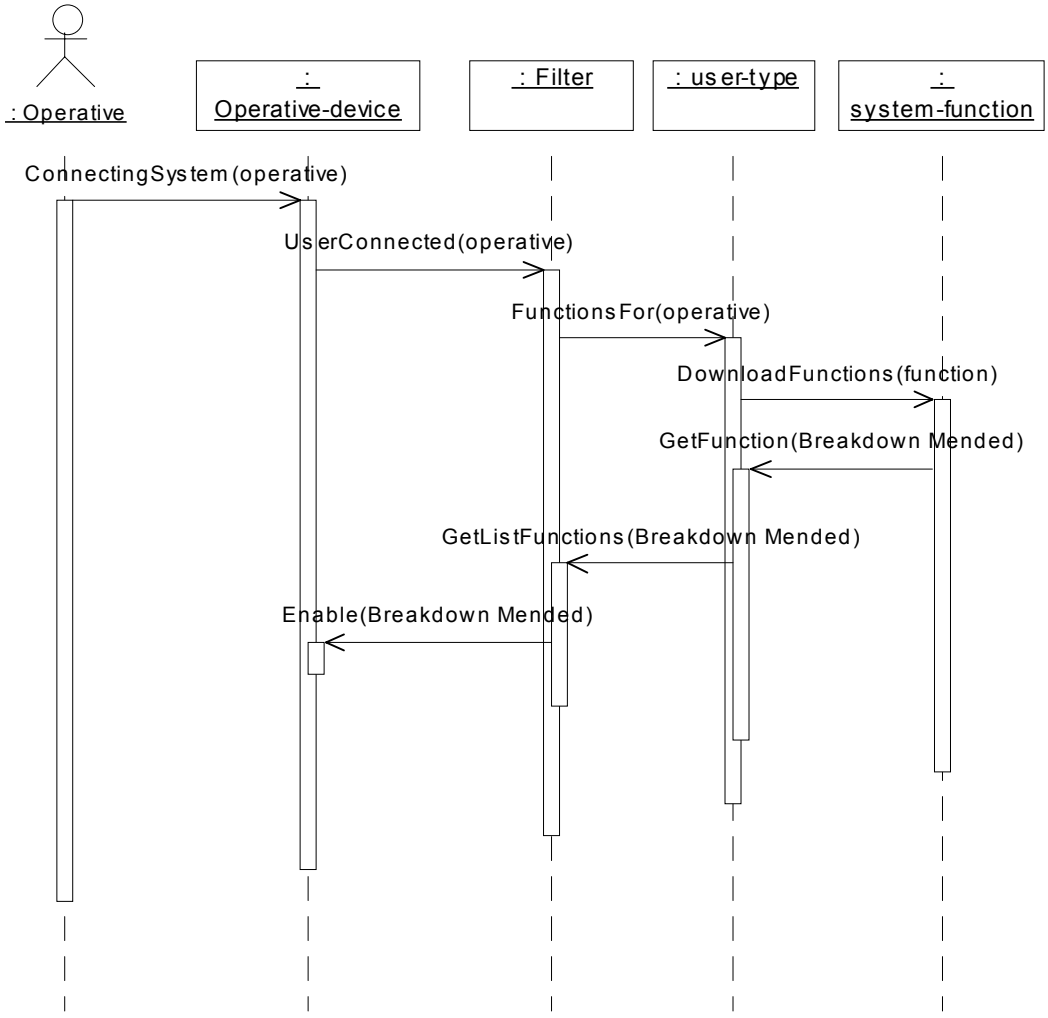


Figure D.32 Sequence diagram for the second application with Workflow model mechanism

STEP 3. Abstraction of the design solution for Workflow Model

There are no modifications to the above generalisation presented in the first iteration.

D.13 User Profile Second Iteration

STEP 1. Design models without User Profile

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

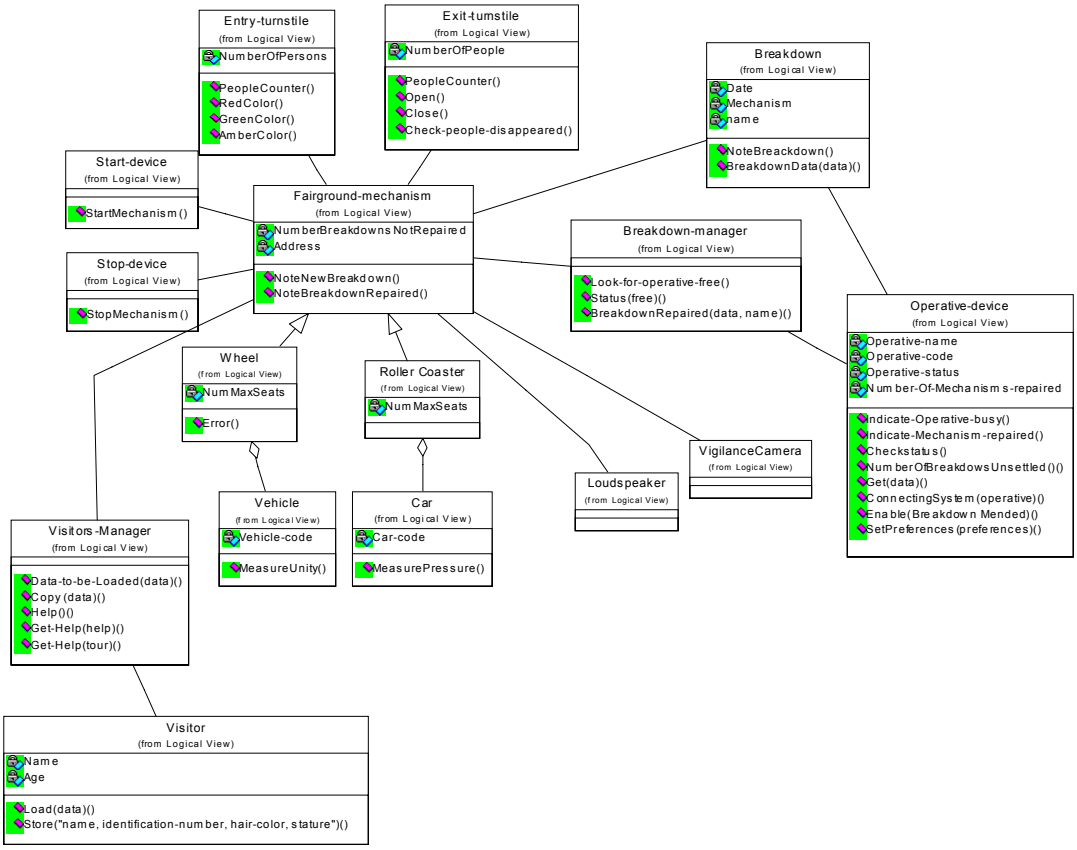


Figure D.33 Class diagram for the second application without User Profile mechanism

STEP 1.2. Design models with User Profile

Requirement: the operator connects to the system and his preferences are established in the operator device.

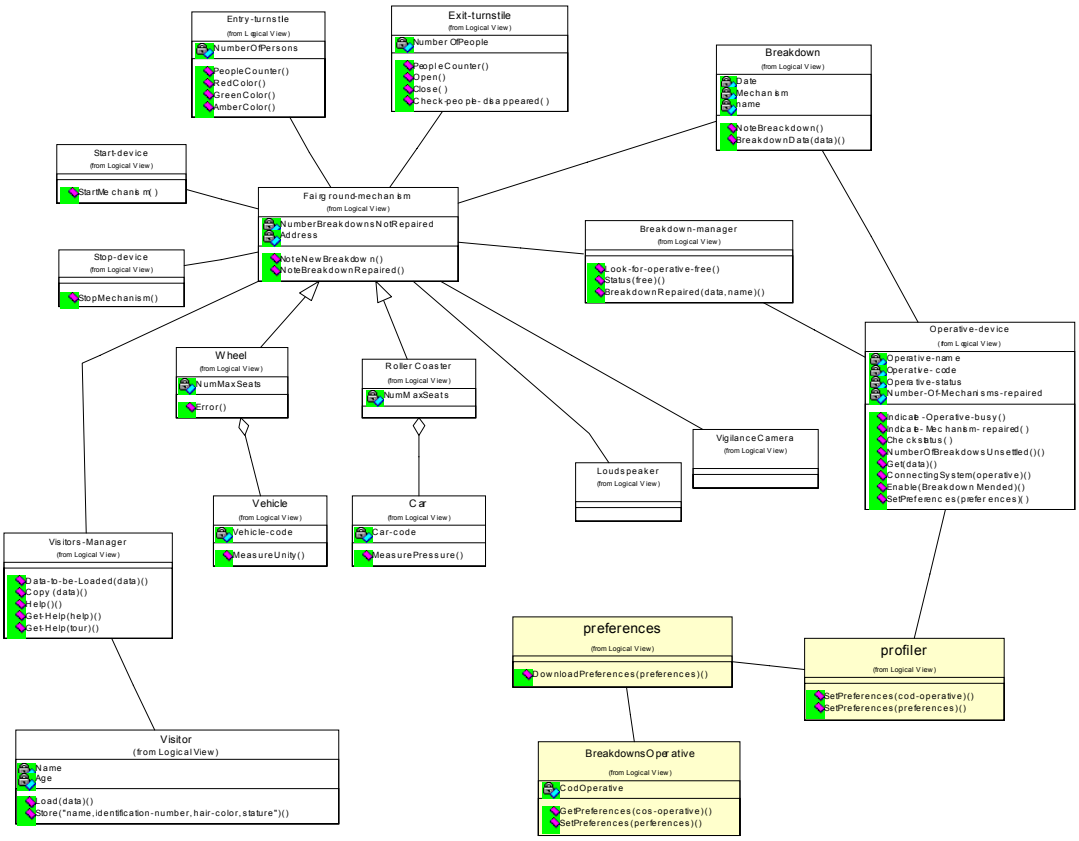


Figure D.34 Class diagram for the second application with User Profile mechanism

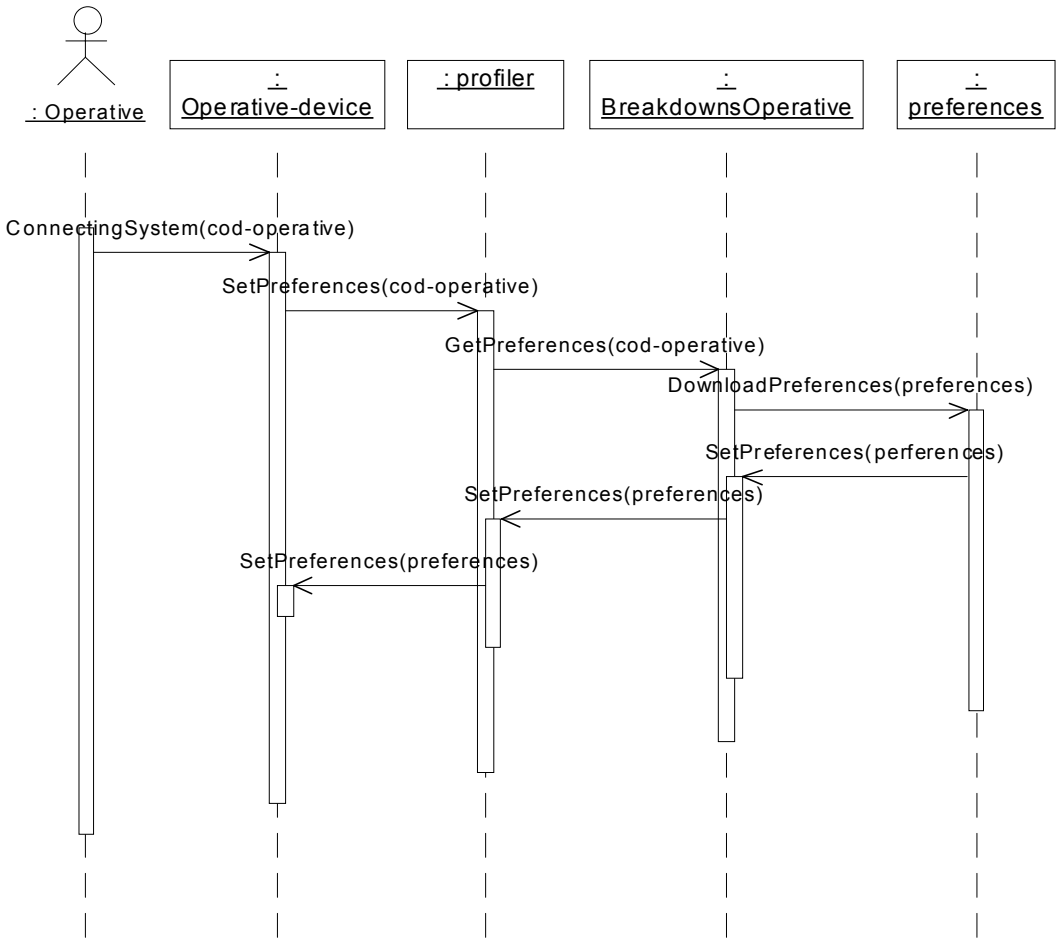


Figure D.35 Sequence diagram for the second application with User Profile mechanism

STEP 3. Abstraction of the design solution for User Profile

There are no modifications to the above generalisation presented in the first iteration.

D.14 Shortcuts Second Iteration

STEP 1. Design models without Shortcuts

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

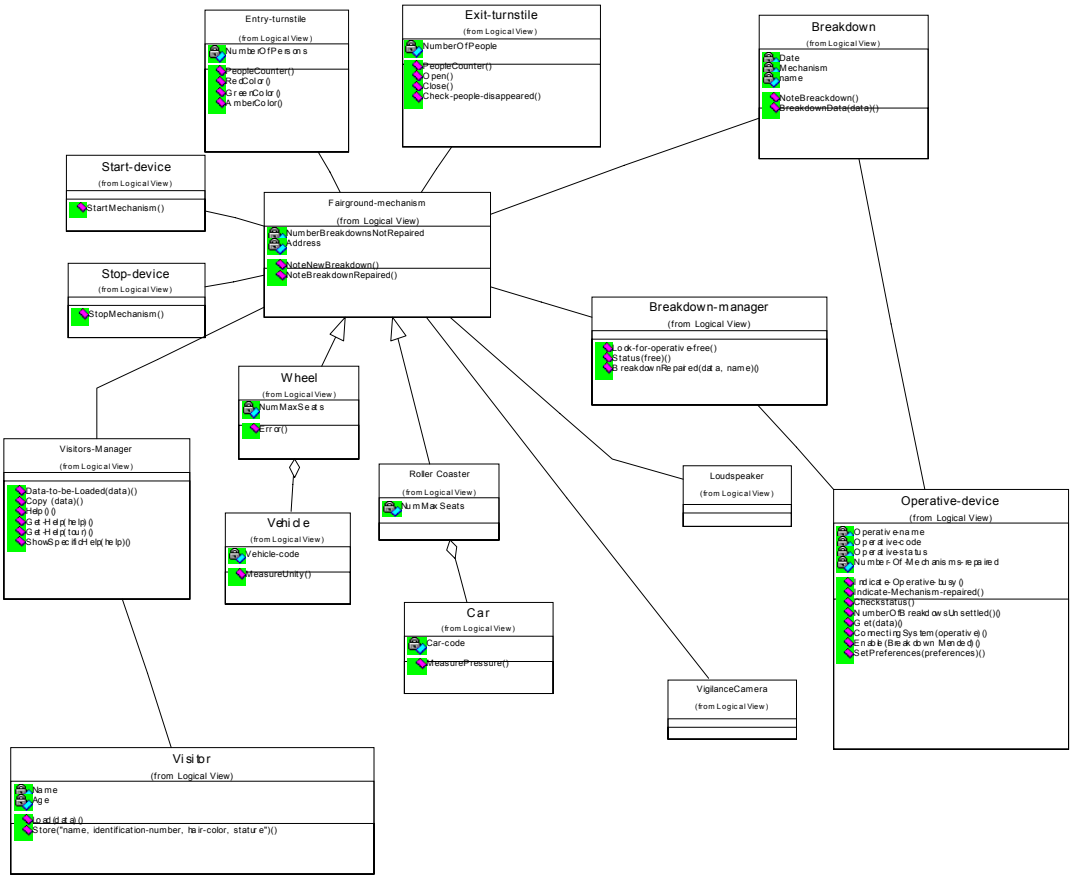


Figure D.36 Class diagram for the second application without Shortcuts mechanism

STEP 2. Design models with Shortcuts

Requirement: the operator creates a rapid access with F1 to indicate that an ongoing fault has just been repaired.

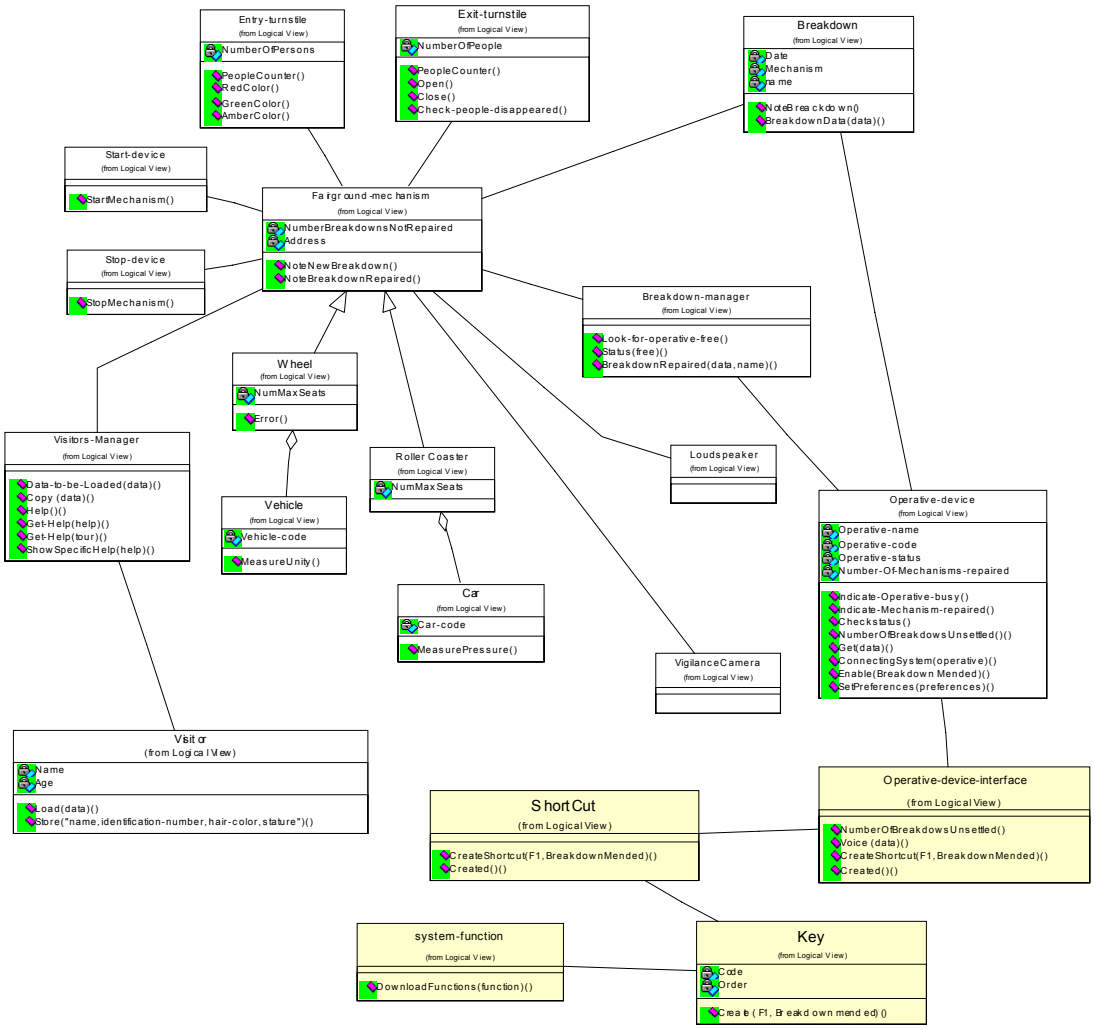


Figure D.37 Class diagram for the second application with Shortcuts mechanism

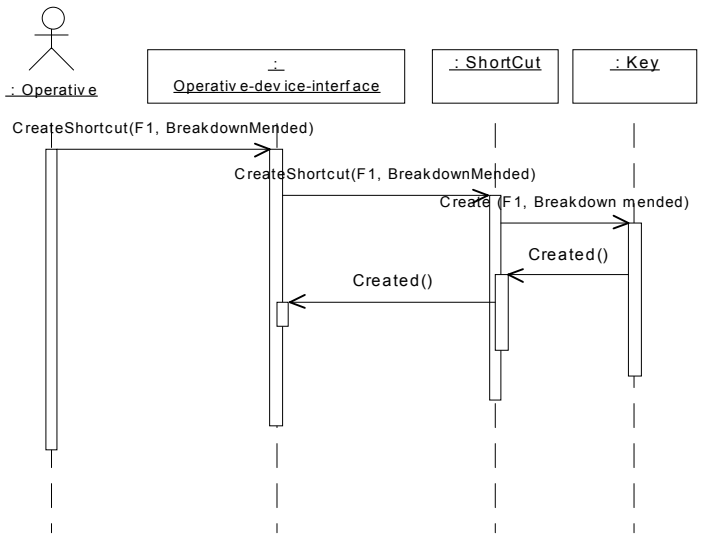


Figure D.38 Sequence diagram for the second application with Shortcuts mechanism

STEP 3. Abstraction of the design solution for Shortcut

There are no modifications to the above generalisation presented in the first iteration.

D.15 Context Sensitive Help Second Iteration

STEP 1. Design models without Context Sensitive Help

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

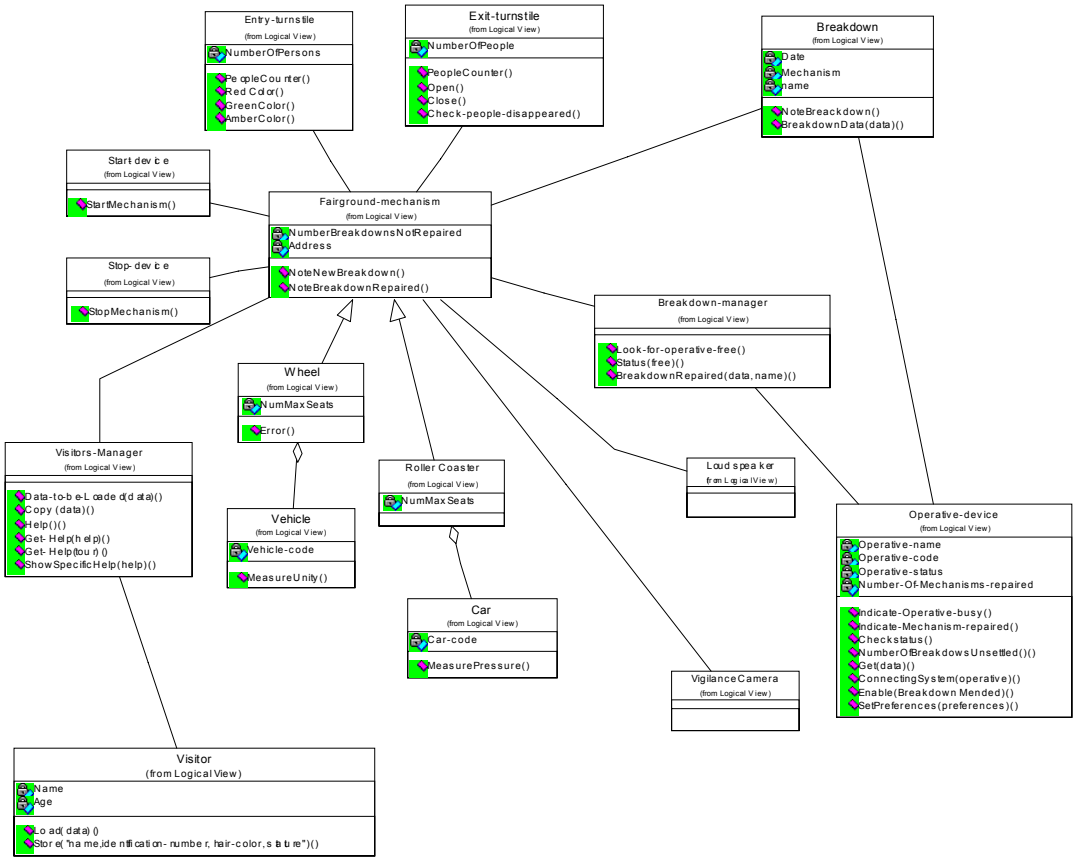


Figure D.39 Class diagram for the second application without Context Sensitive Help mechanism

STEP 2. Design models with Context Sensitive Help

Requirement: the user can get sensitive help when the cursor is placed over specific elements located in the interface.

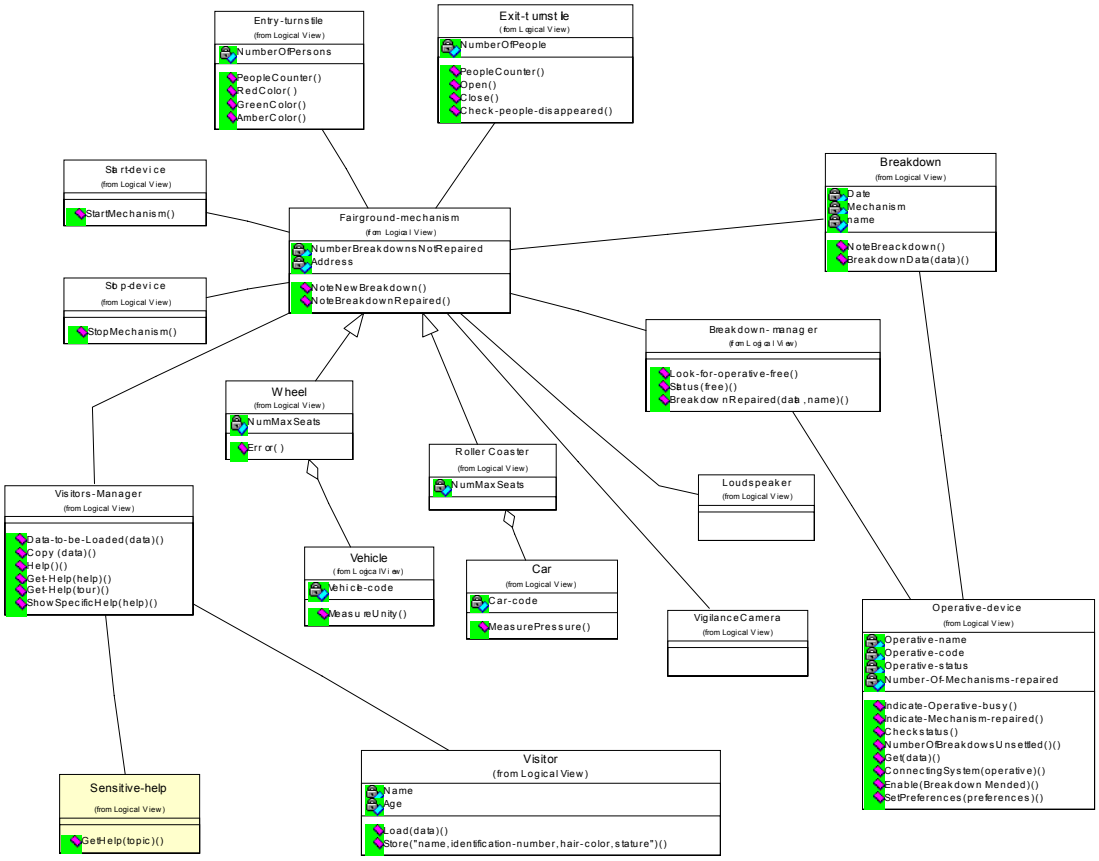


Figure D.40 Class diagram for the second application with Context Sensitive Help mechanism

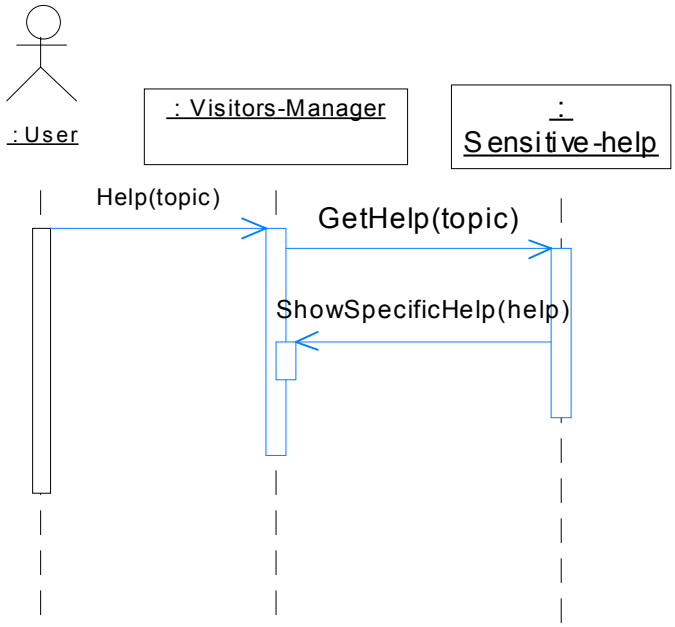


Figure D.41 Sequence diagram for the second application with Context Sensitive Help mechanism

STEP 3. Abstraction of the design solution for Context Sensitive Help

There are no modifications to the above generalisation presented in the first iteration.

D.16 Wizard Second Iteration

STEP 1. Design models without Wizard

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

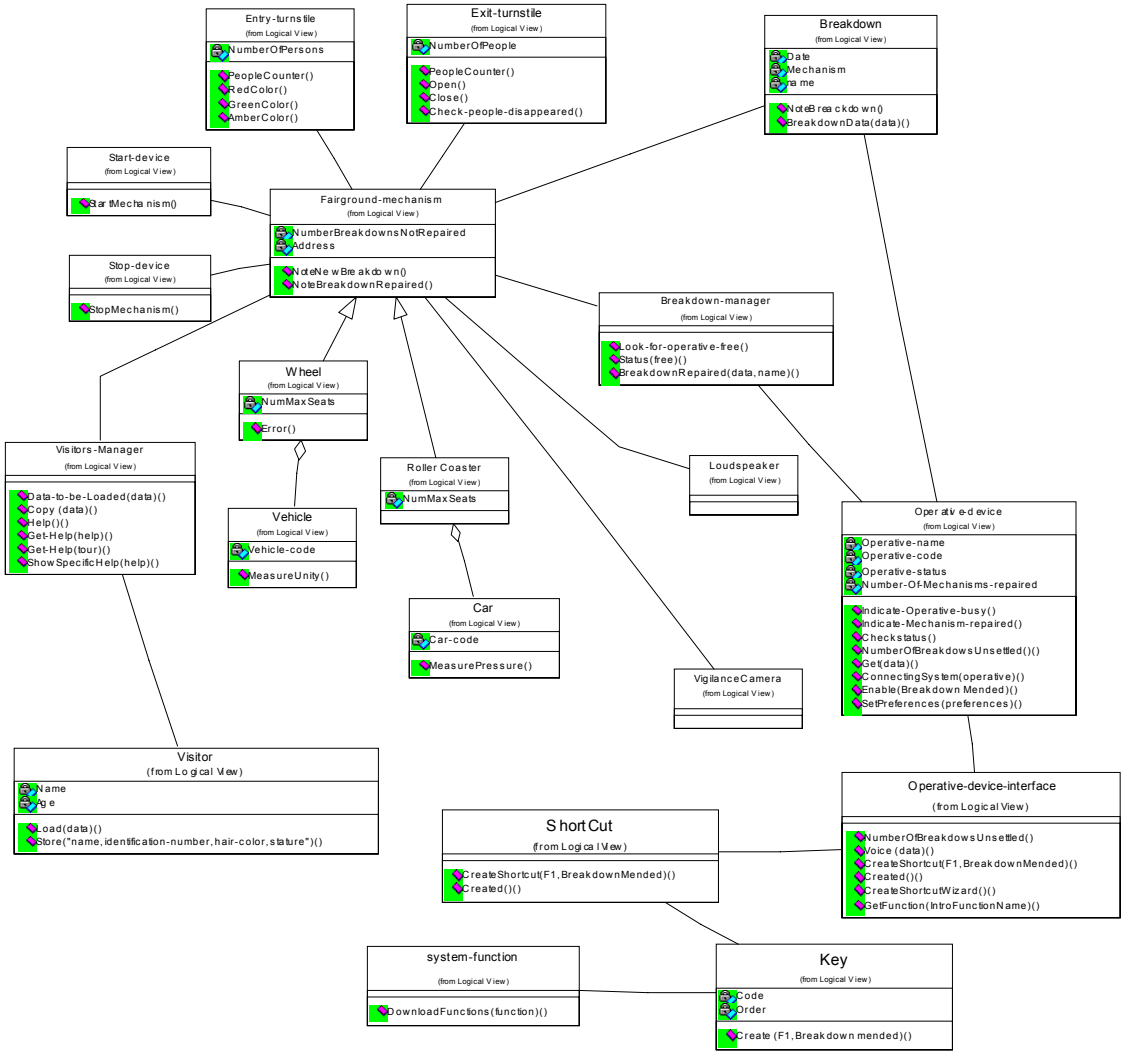


Figure D.42 Class diagram for the second application without Wizard mechanism

STEP 2. Design models with Wizard

Requirement: the operator creates a rapid access with F1 for the functionality “Fault repaired” using the Wizard

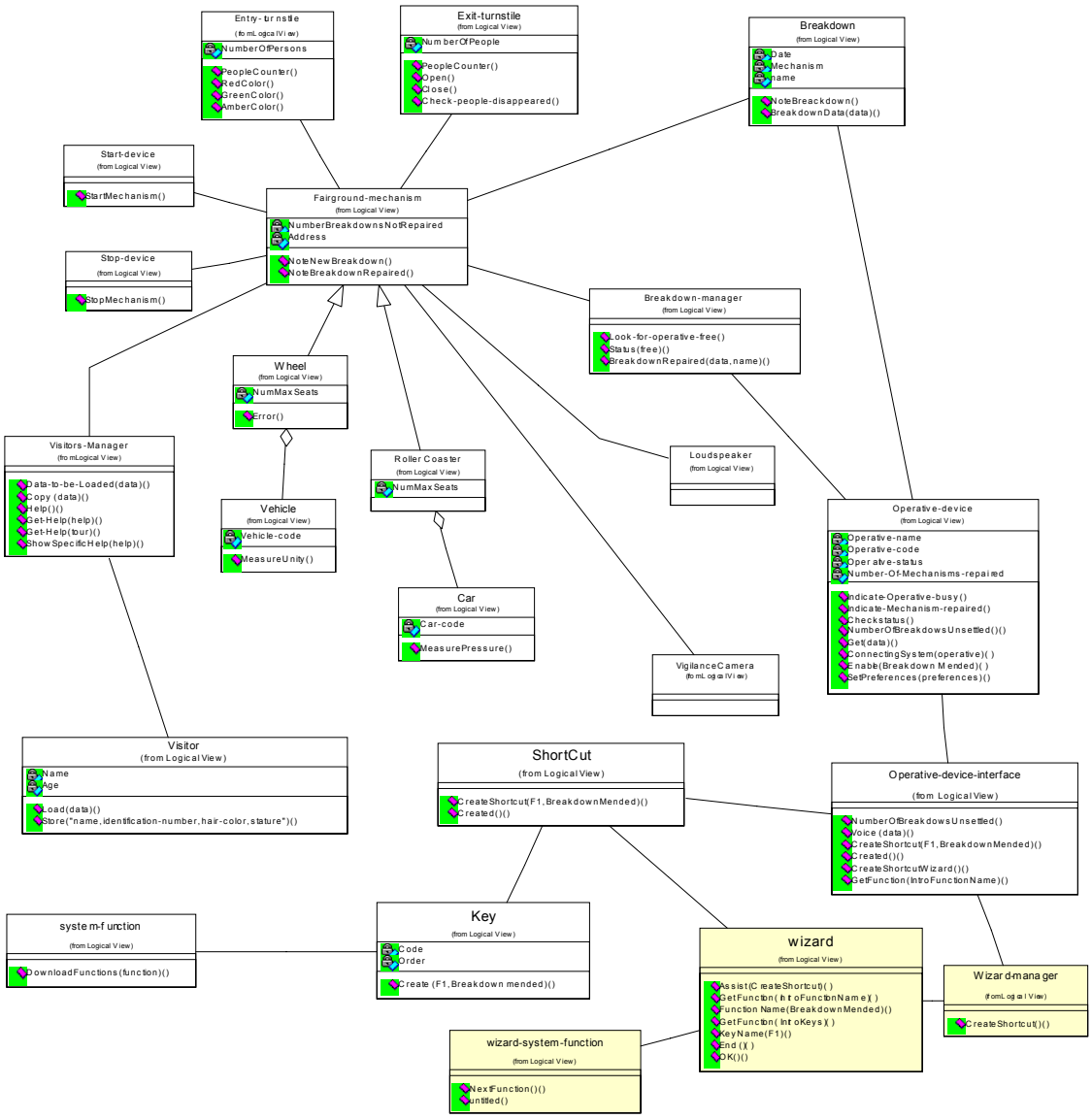


Figure D.43 Class diagram for the second application with Wizard mechanism

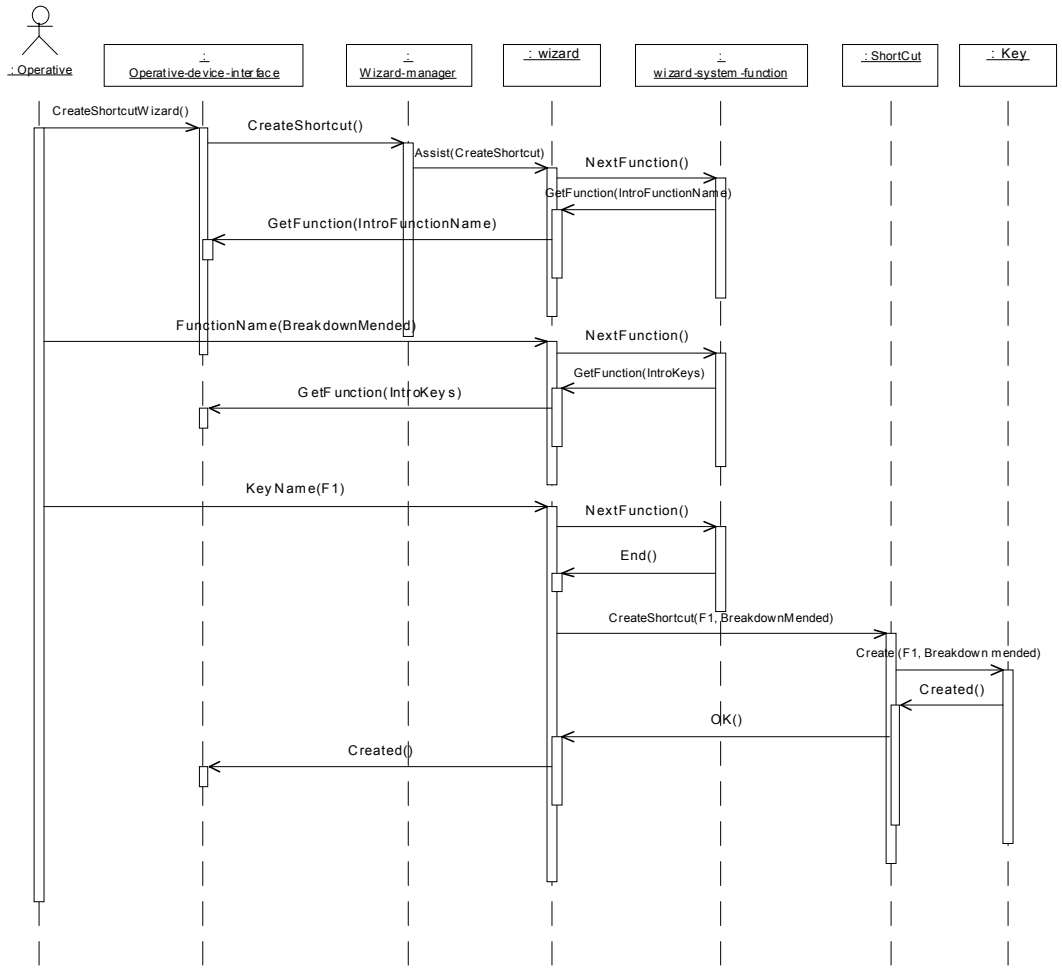


Figure D.44 Sequence diagram for the second application with Wizard mechanism

STEP 3. Abstraction of the design solution for Wizard

There are no modifications to the above generalisation presented in the first iteration.

D.17 Cancel Second Iteration

In the case of the amusement park design, as it is essentially a control application where there is not much human intervention, it makes not sense to apply the Cancel pattern, which explains why no second iteration appears for this pattern.

D.18 Multi tasking Second Iteration

STEP 1. Design models without Multi tasking

In this case, the interaction diagram does not exist in the original version of the system without the corresponding usability pattern.

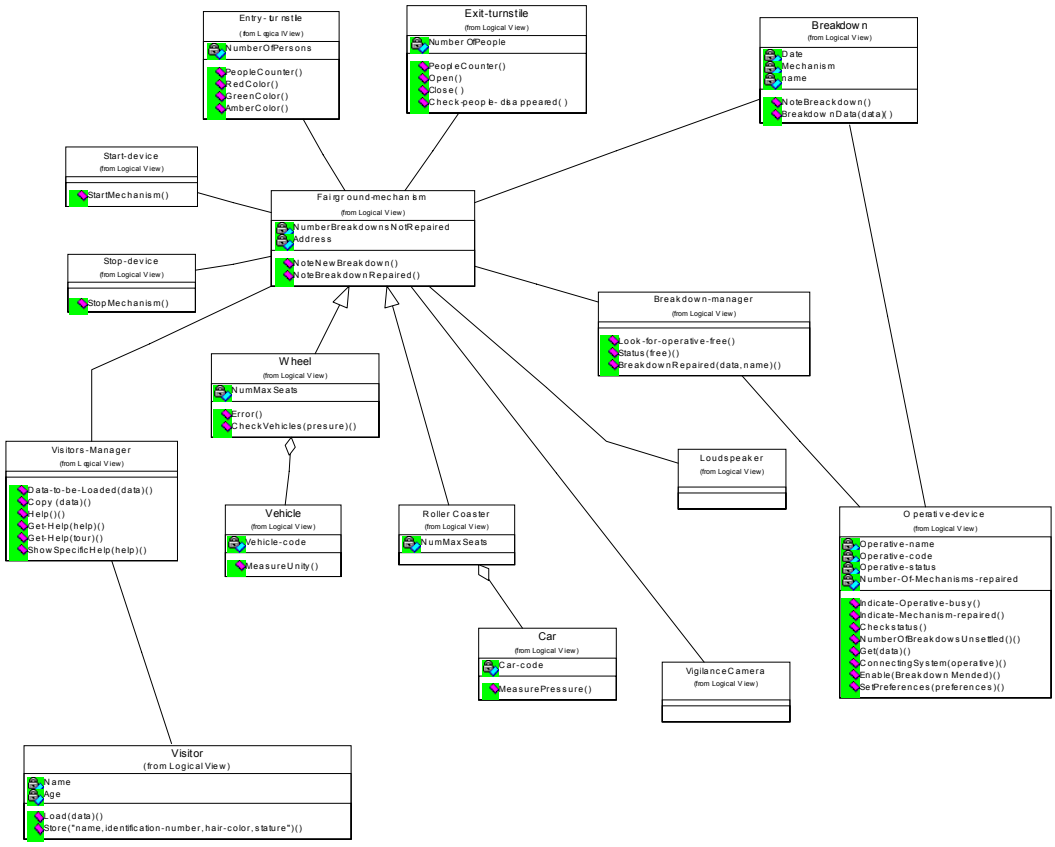


Figure D.45 Class diagram for the second application without Multi-Tasking mechanism

STEP 2. Design models with Multi Tasking

Requirement: the clock advises the dispatcher every three seconds to check all the rides.

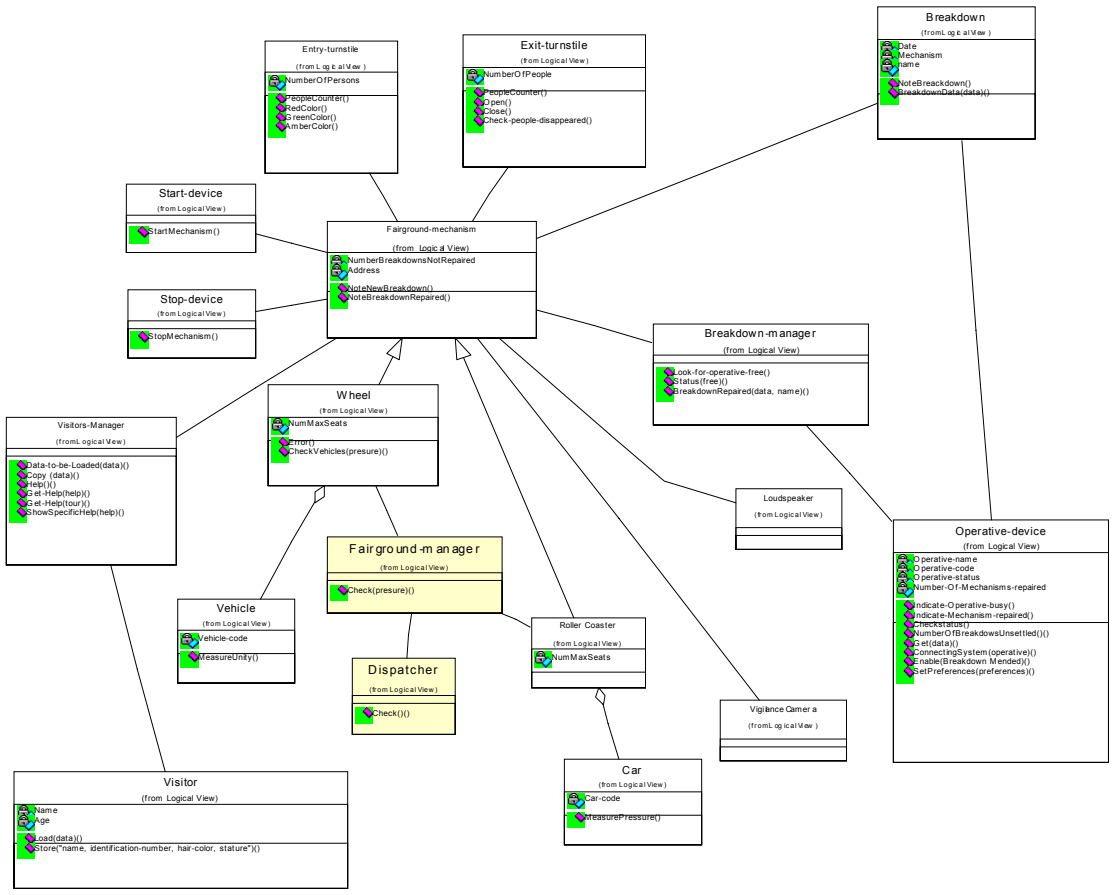


Figure D.46 Class diagram for the second application with Multi-Tasking mechanism

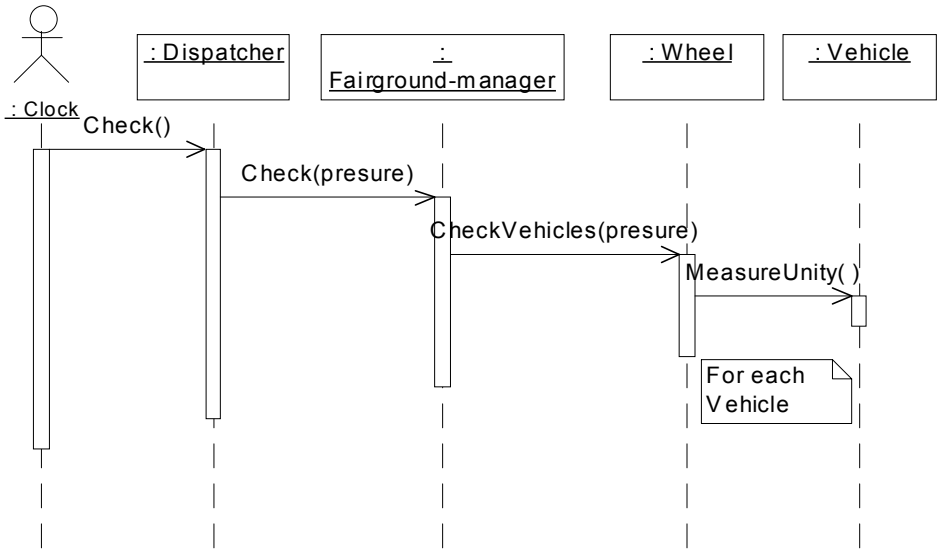


Figure D.47 Sequence diagram for the second application with Multi-Tasking mechanism

STEP 3. Abstraction of the design solution for Multi tasking

There are no modifications to the above generalisation presented in the first iteration.

D.19 Command Aggregation Second Iteration

In the case of the amusement park design, as it is essentially a real-time control application where no variables must be changed during the execution of the system, it makes no sense to apply the Command Aggregation pattern, which explains why no second iteration appears for this pattern.

D.20 Actions for Multiple Objects Second Iteration

In the case of the amusement park design, as it is essentially a real-time control application where nothing is batch processed, it makes no sense to apply the Actions for Multiple Objects pattern, which explains why no second iteration appears for this pattern.

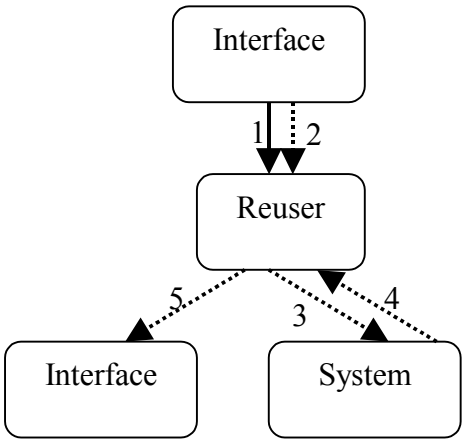
ANNEX E: PHASE 2 VALIDATION WITH PRACTITIONERS IN A REAL PROJECT

Because of its size, this annex has been stored in another file.

ANNEX F: CATALOGUE OF USABILITY PATTERNS

F.1 Reusing Information Architectural Usability Pattern

- **Pattern Name**: Reusing Information.
- **Usability Mechanism**: This pattern enables the user to move data from one part of a system to another. So users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system.
- **Solution**:
  - Diagram:



  - Participants:
    - Interface: collects the data to be processed by the reuser pattern and finally displays the operation results (if the user needs to see the result). Interface sends the data to be processed (1) and the function requested by the interface (2), i.e. copy, paste, move, etc., to Reuser. Also, once the reuser pattern has been applied the results of the requested function will be displayed on the interface (5), unless the requested function was "copy".
    - Reuser: is the module that gathers the information provided by the interface and manipulates these data according to the requested function (copy, paste, move, etc.). Reuser receives the data to be manipulated as well as the function to be executed (1) (2). If Reuser does not store the data to be manipulated internally, it has to send these data to the system (3), as happens, for instance, with the Copy function. Also if Reuser does not store the data internally, it has to ask for these data from the part of the system where they are stored (4) as happens with the paste or move functions.
    - System: this component is optional and is only necessary when the Reuser module does not store the data internally.
- **Usability benefits:** The reuse of data in an application as well as across different applications minimises users' cognitive load and also inputs fewer errors into the process. It also improves the adaptability of the application or applications that enable data reuse.
- **Usability rationale:** By preventing the error input by users, the application of this pattern improves system *reliability*. User *efficiency* is also improved. Additionally, by building a more adaptable system, the *satisfaction* of the end user is improved too.
- **Consequences:**

- **Related patterns:**
  - System performance will be better if the information to be reused is stored in the Reuser module rather than in another part of the system, because this reduces the system interaction level.
- **Pattern Implementation in OO:** Interface generates some classes. Reuser generates one or more "Reuser" classes, furnished with the manipulation methods (copy, paste, move, etc.) and "Data-to-be-reused" classes, which store the data to be manipulated in the class or through a link to another one. In this case, it was decided to store the data outside the reuser class to respect the encapsulation principle.
- **Example:** Suppose the waiter inputs the foodstuff code and, as the next consumption ordered is the same, the waiter uses the "duplicate last foodstuff" function.

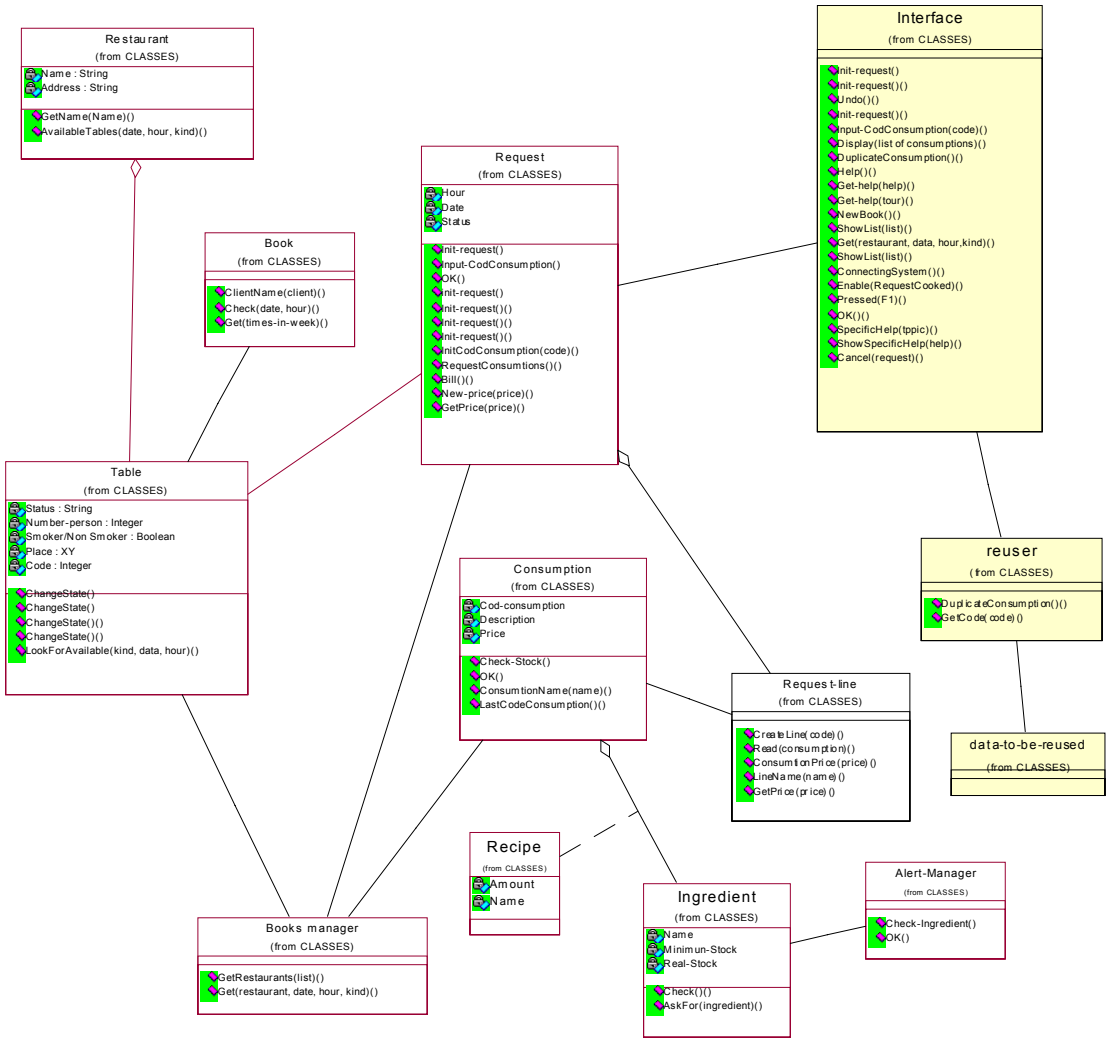


Figure F.1 Class diagram for restaurant management with Reusing Information mechanism

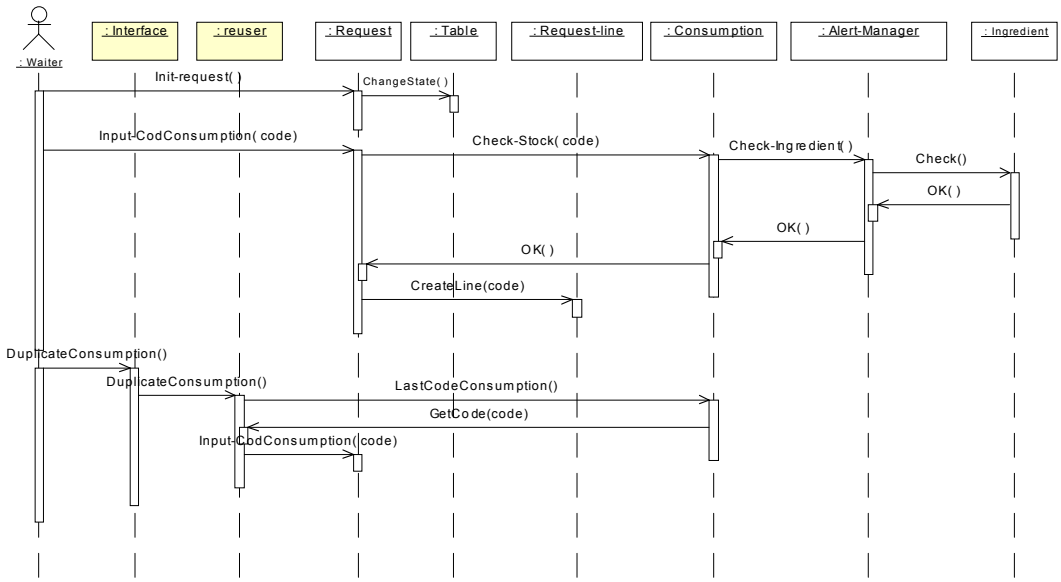
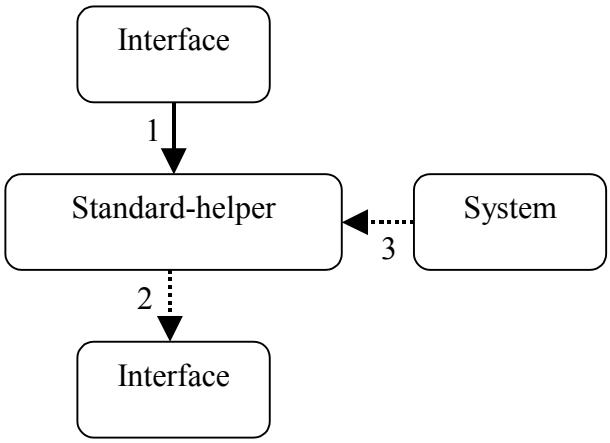


Figure F.2 Sequence diagram for restaurant management with Reusing Information mechanism

F.2. Standard Help Architectural Usability Pattern

- **Pattern Name:** Standard help.
- **Usability Mechanism**: The system must allow the user to ask for information and help about the tasks performed by the system.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: gathers the information from the help application and sends this information to the module which manages the help (1). Also it will show the help information sent by the Standard-helper (2)
    - Standard-helper: will show a general help (that is, not specialised) for the application. This help is usually identified as an html, doc, etc., document. This component receives the application from the interface (1) and sends the respective data to the interface (2). If the help is not stored in this component, the help will be provided for another component using the data flow from System (3).
    - System: this component is optional and represents the part of the system where the help is stored if the Standard-helper does not store the help internally. It will be the system that provides the Standard-helper with the help (3).
- **Usability benefits:** The provision of help will give the user guidance, and will improve error management, both error detection and error correction.
- **Usability rationale:** Help is essential in any system because it improves learnability by providing the user with guidance about system functions. *Efficiency* also improves, because this is one of the spaces used for error management. However, this help must be well organised and displayed in the user language, otherwise user efficiency may fall.
- **Consequences:** System performance will be better if the help files are stored in the same Standard-helper module rather than in any other part of the system.
- **Related patterns:** Guided-helper and Sensitive-helper, because both helps can be stored in the same "Help" class, furnished with special methods to manage each kind of help provided by either Standard-helper or Guided-helper.
- **Pattern implementation in OO:** The interface generates some classes. The Standard-helper component will be implemented using a class with an attribute that stores the help or a link to another class (represented by System) that contains the help.

- **Example:** The user can push the Help button

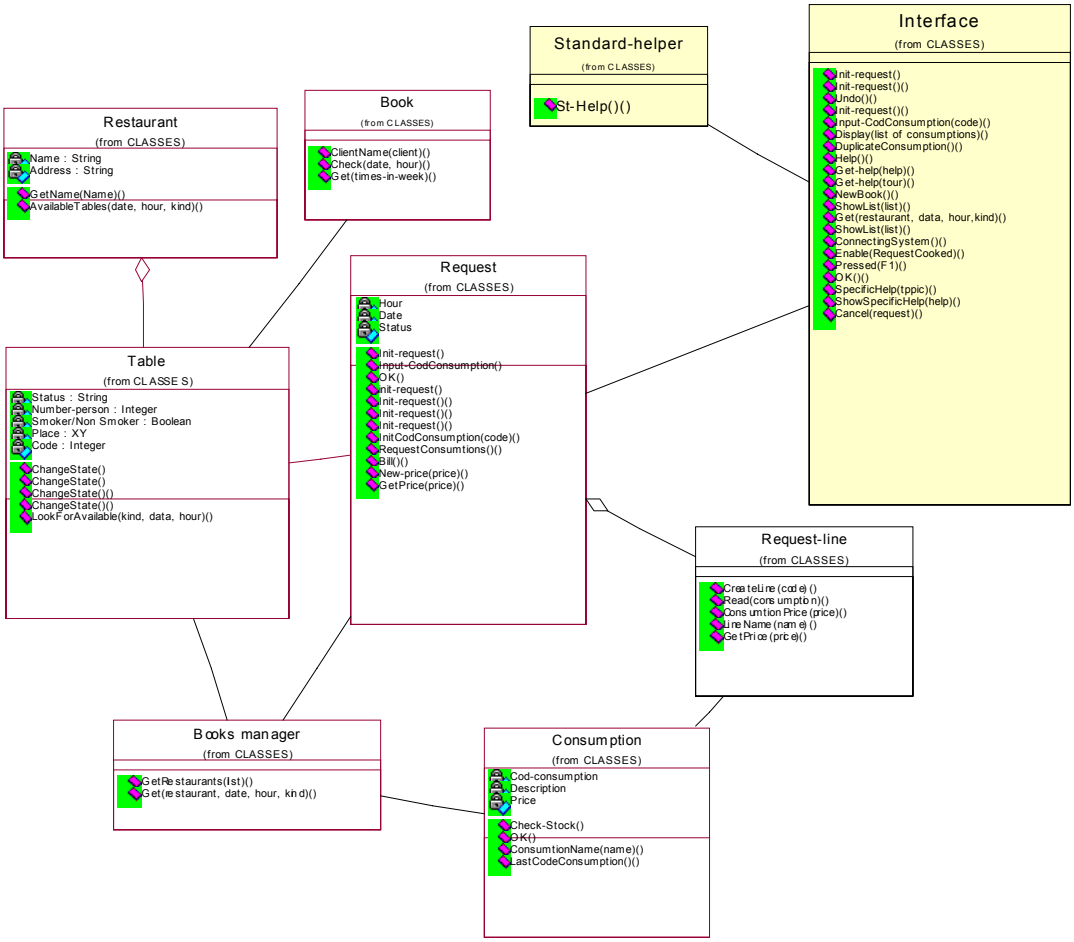


Figure F.3 Class diagram for restaurant management with Standard Help mechanism

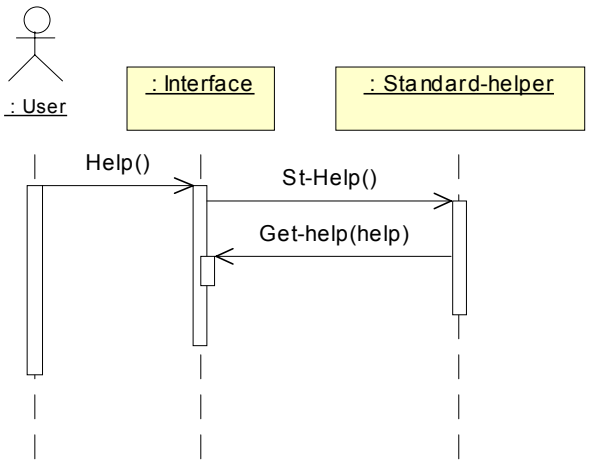
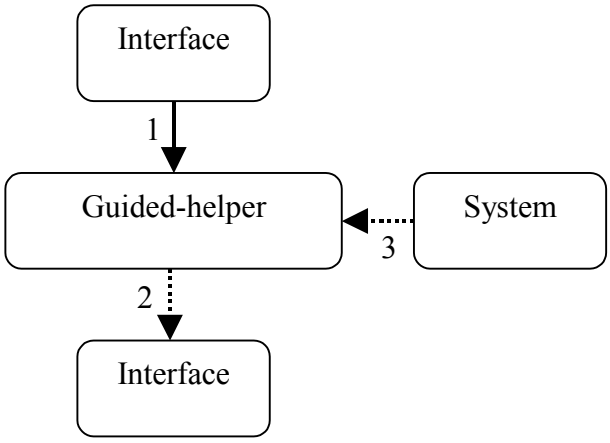


Figure F.4 Sequence diagram for restaurant management with Standard Help mechanism

F.3Tour Architectural Usability Pattern

- **Pattern Name:** Tour.
- **Usability Mechanism**: A tour presents users with information explaining how to do routine system tasks, providing step-by-step guidance.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: collects the guided help request and sends it to the Guided-helper (1). Additionally, it will display the help information it receives from the Guided-helper (2).
    - Guided-helper: displays a guided help for the application for which the help has been described (2). This help can range from a pre-recorded tour of the application, to an interactive tour, which involves the development of a separate application. If the help is not stored internally in this component, this help will be provided by any other part of the system through the information flow from system (3).
    - System: this is an optional component and represents part of the system in which the help will be stored if the Guided-helper does not store the information internally. System will, therefore, be responsible for providing the Guided-helper with the help through (3).
- **Usability benefits**: The provision of a tour will give the user guidance and will improve error management, both error detection and error correction.
- **Usability rationale:** System *learnablity* and *memorability* may be improved, but it may have a negative impact on *efficiency*, as users might be induced to follow a specific task order, passing up the use of possible shortcuts.
- **Consequences:** *System performance* will be better if the help files are stored in the Standard-helper module rather than in any other part of the system, as this requires fewer interactions.
- **Related patterns:** Standard-helper and sensitive-helper, because both helps can be stored in the same "Help" class, furnished with special methods to properly handle the two types of help provided by Standard-helper and Sensitive-helper.
- **Pattern implementation in OO:** The Interface component will generate one or more classes. The Guided-helper component will generate a class that will have an attribute that contains the help or

a pointer to another place (another class or another system) that is capable of providing this help. In this example, the Guided-helper class stores the help internally and does not need to ask another part of the system for the help.

- **Example:** the user can push the Guided Help button.

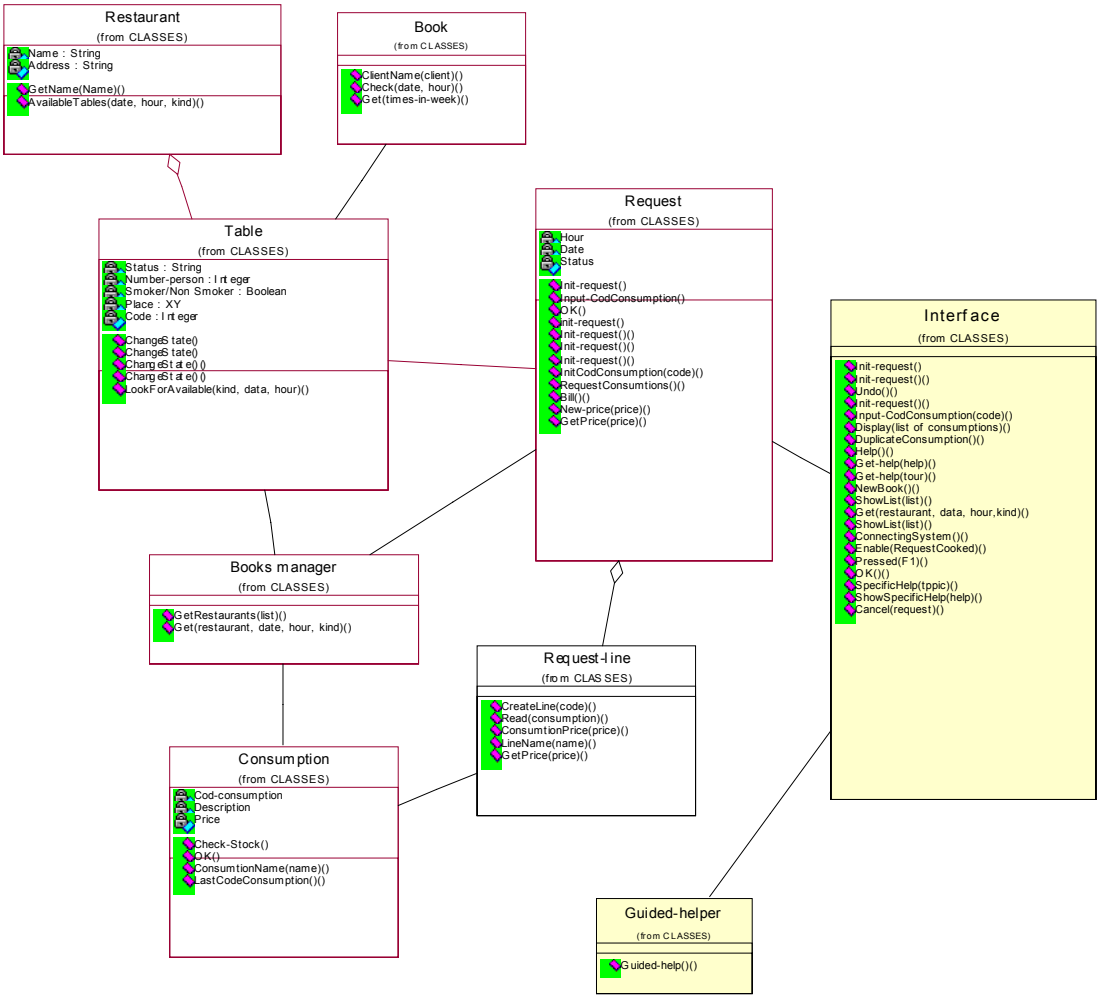


Figure F.5 Class diagram for restaurant management with Tour mechanism

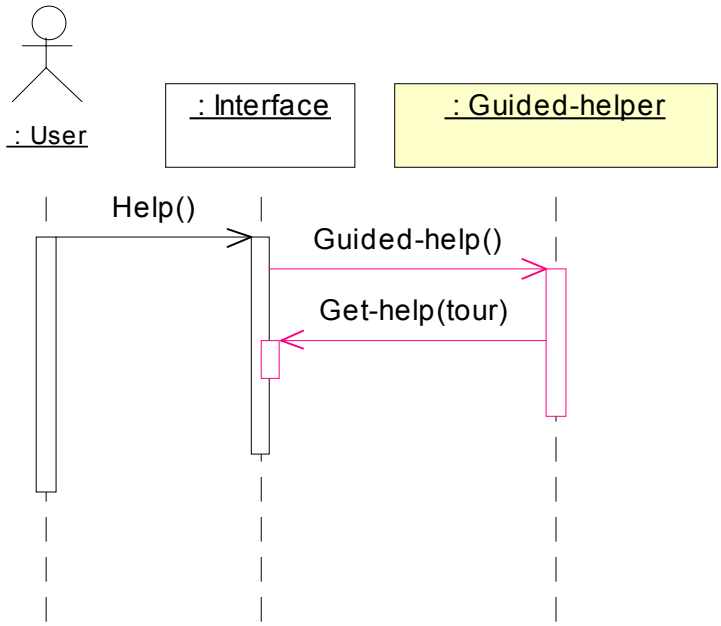
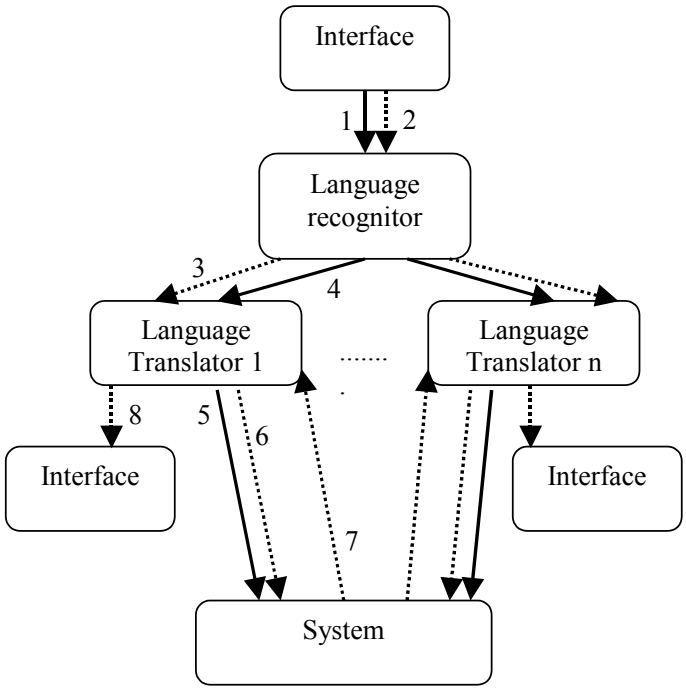


Figure F.6 Sequence diagram for restaurant management with Tour mechanism

F.4 Different Languages Architectural Usability Pattern

- **Pattern Name:** Different Languages.
- **Usability Mechanism:** Internationalisation refers to the capability of the software to interact with users in different languages.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: collects the operation to be performed and any associated data, which it sends to the Language-recogniser (1) (2). Additionally, once the respective functionality has been processed, the interface receives the data to be displayed to the user from the Language-translator in the language that originated the request (8).
    - Language-recogniser is a recogniser, not a translator, which determines the language in which a the respective functionality is requested and sends the data and the functionality request to the respective Language-translator (3) (4).
    - Language-translator (i): there may be one for each language that the system is capable of recognising. If there is one for each language, which would be advisable for reasons of system modularity, each Language-translator translates the functionality and any data it receives from the Language-recogniser (3) (4) to a common language understood by the system. Once they have been translated to the common language, it sends them to the system (5) (6). Once the functionality has been processed in the system, it again receives the response data for the executed functionality (7), and again translates them from the common language to the specific language in which the user requested the functionality. After translating, it sends the data to the user (8) through the interface.

- System: it performs the functionality requested by the Language-translator (i), in the common language (5) (6), and returns the respective response to the language-translator in the common language (7).

- **Usability benefits:** This pattern improves system *accessibility* by users using different languages. It also improves error prevention by giving a better understanding of the options and tasks to be performed by the system.
- **Usability rationale:** The use of this pattern improves *reliability* and user *efficiency*, as it eliminates possible sources of error in system use. User *satisfaction* may be increased by allowing the use of the system in different languages. However, system performance falls because of it having to manage these languages, which may also have a negative impact on satisfaction. *Learnability* also increases by enabling the user to use the system in the language with which he feels most at home.
- **Consequences:**
  - Decreased system *performance*, as it involves a longer translation time from one language to another.
  - Increased *portability*, as the system will be able to operate on the same platform in different locations.
  - If different Language-translator modules are used, increased *maintainability* of the system, as if a new language included, all that has to be done is to add a new module. Additionally, system maintainability is improved if the Function-Dispatcher class that appears in the implementation section is used.
- **Related patterns:** The diagram is similar to the Device-recogniser, but the functionality is different even though they share the Function-dispatcher.
- **Pattern implementation in OO:** This pattern generates a language recognition class (recogniser), a class for each language to be translated (translator) and another Function-dispatcher class, which, once a request has been translated to the generic language, is responsible for conveying this request to the respective class for processing.
- **Example:** When the user is booking a table from the terminal, the system should be able to understand the date, time and table time irrespective of the language used by the user. The interaction diagram does not show the full booking for reasons of visibility on the model.

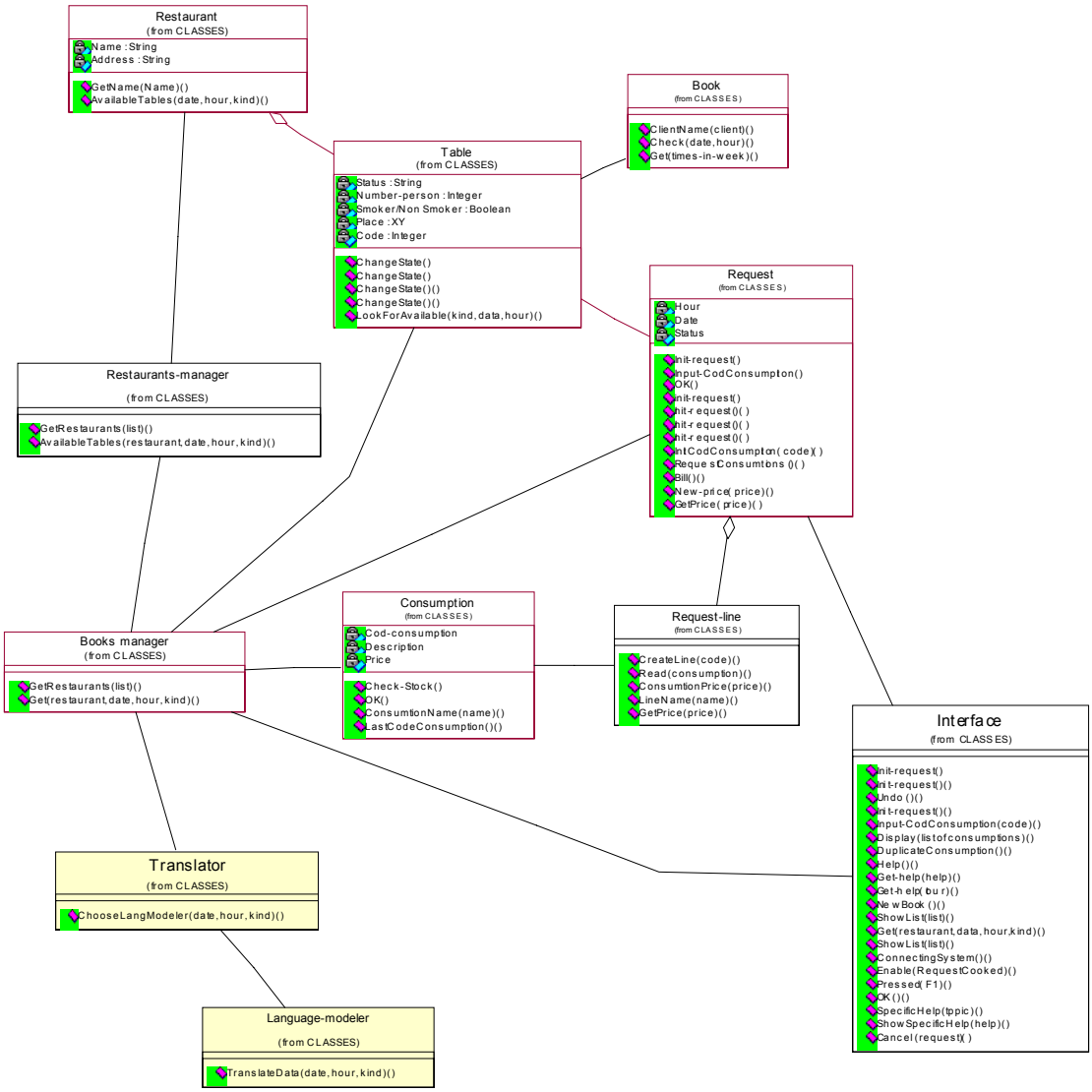


Figure F.7 Class diagram for restaurant management with Different Languages mechanism

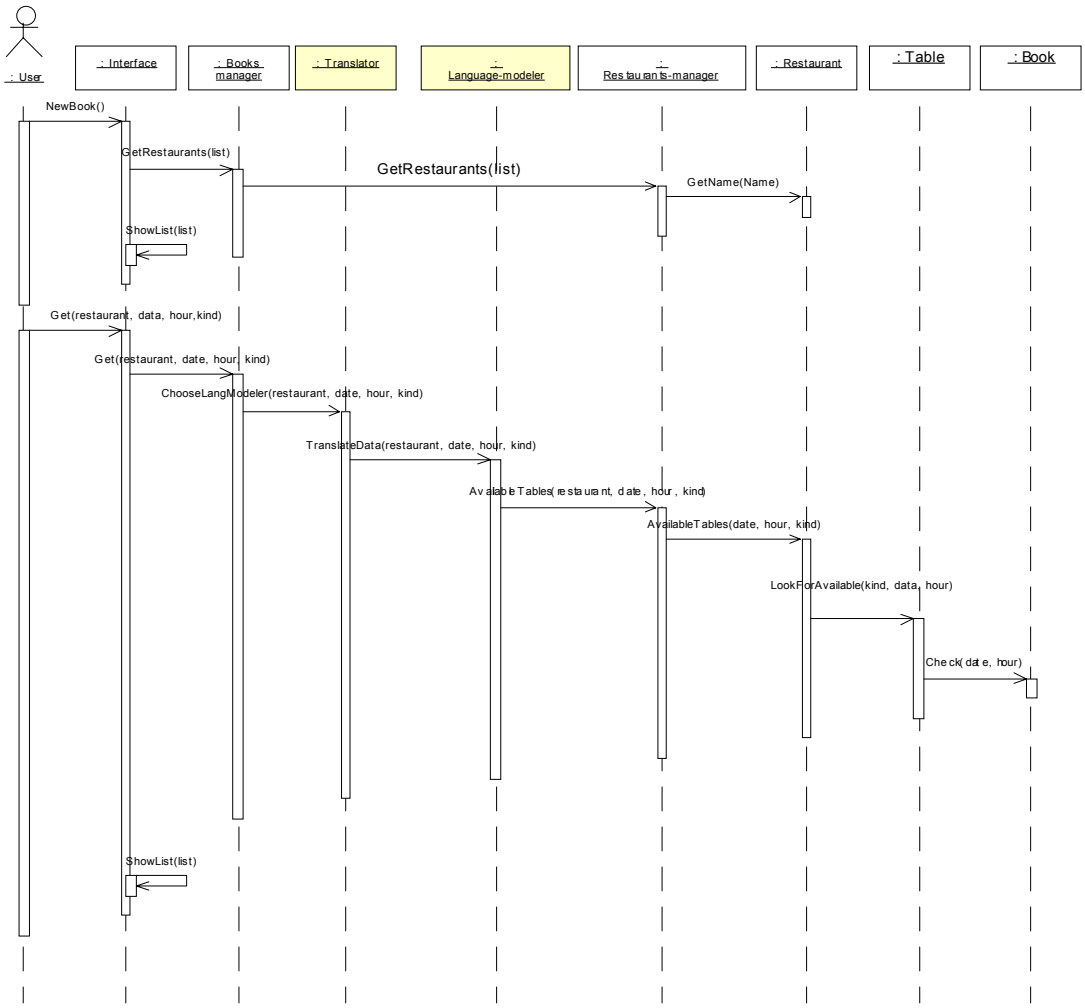
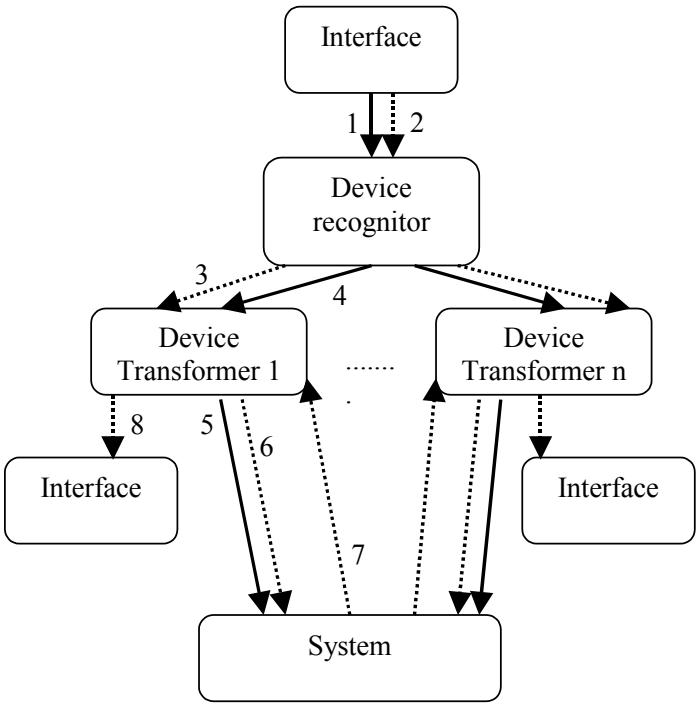


Figure F.8 Sequence diagram for restaurant management with Different Languages mechanism

F.5 Different Access Methods Architectural Usability Pattern

- **Pattern Name:** Different Access Methods.
- **Usability Mechanism**: Access method means the capability of the software of being accessed from different types of physical devices. So, this attribute will make the system easier to access not only from the desktop or laptop, but also using devices like WAP, Web, and interactive TV, for example.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: collects the operation to be performed and any associated data, which it sends to the Device-recogniser (1) (2). Additionally, once the respective functionality has been processed, the interface receives the data to be displayed to the user from the Device-transformer in the format in which the user placed the request (8).
    - Device-recogniser: is a signal format recogniser, which sends the signal to one device or another for interpretation, depending on the type of signal it receives. Additionally, it sends the data and the functionality request to the respective device-transformer (5) (6).
    - Device-transformer: (i) there may be one for each device that the system is able to recognise. If there is one for each device, which would be advisable for reasons of system modularity, each Device-transformer is responsible for converting both the functionality and any data it receives from the Device-recogniser (3) (4) to a general functionality understood by the system. Once the signal has been converted to a functionality and/or data that can be understood by the system, it is all sent to the system for it to perform the respective operation (5) (6). Additionally, once the functionality has been processed in the system, it again

receives the response data for the executed functionality (7), which it again translates to the specific signal format in which the user requested the functionality. After translation, it sends the data to the user (8) through the interface.

- System: it performs the functionality requested by the Device-transformer (i) in the common functionality format (5) (6) and returns the response to the respective device-transformer in the aforesaid common format (7).

- **Usability benefits:** This pattern improves system accessibility for users who use different devices.
- **Usability rationale:** User *satisfaction* may be increased by enabling access to the system through different devices. However, system performance falls because of having to manage these devices, which may also have a negative impact on satisfaction.
- **Consequences:**
  - Decreased system *performance*, as it involves a longer conversion time for signals interpreted by different devices.
  - If different Device-transformer modules are used, increased *maintainability* of the system, as if it is decided to include a new device, all that has to be done is to add a new module. Additionally, system maintainability is improved if the Function-Dispatcher class that appears in the Implementation section is used.
- **Related patterns:** The diagram is similar to the Language-translator, but the functionality is different even though they share the Function-dispatcher.
- **Pattern implementation in OO:** This pattern generates a device recognition class (device-recogniser), a class for each signal type to be interpreted (device-transformer) and another Function-dispatcher class, which, once a request has been translated to the generic language, is responsible for conveying this request to the respective class for processing.
- **Example:** The waiter can ask the waiter device what foodstuffs a table has ordered by simply saying “I want to know what foodstuffs table x has ordered”. Additionally, the waiter’s device is capable of verbally reproducing all the foodstuffs.

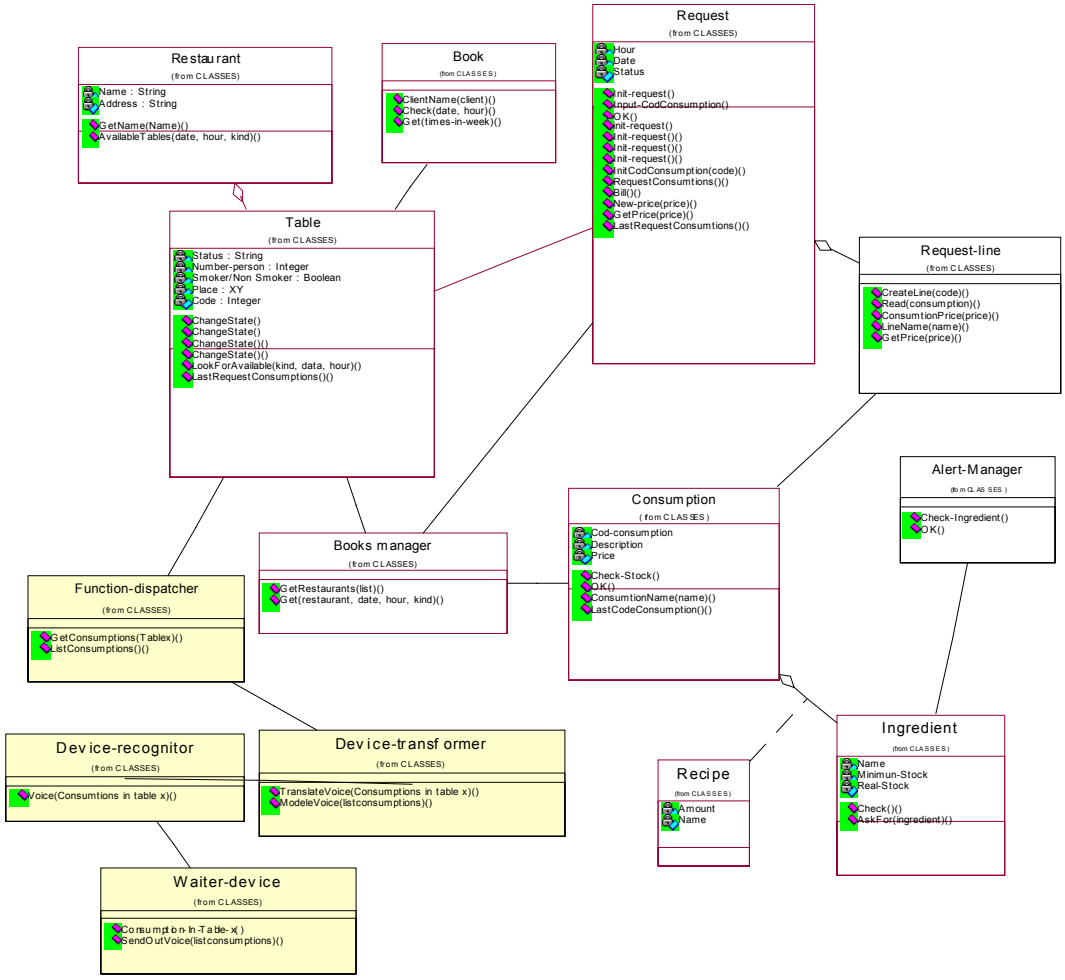


Figure F.9 Sequence diagram for restaurant management with Different Access Methods mechanism

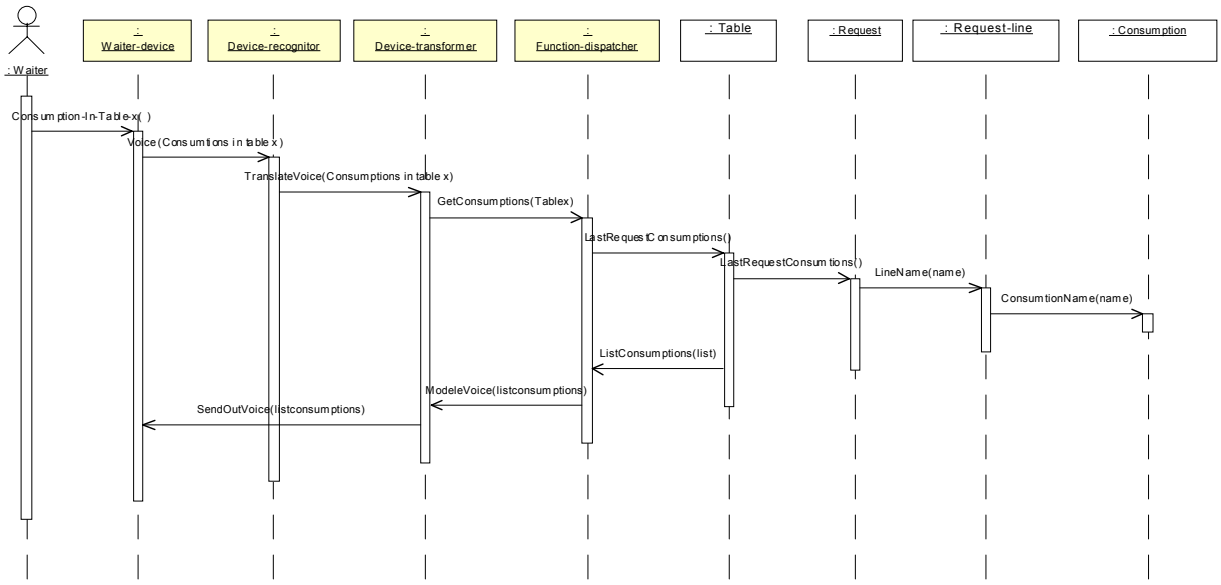
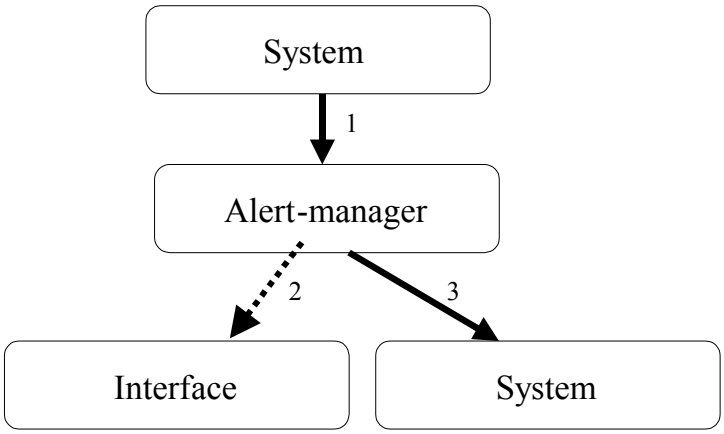


Figure F.10 Sequence diagram for restaurant management with Different Access Methods mechanism

F.6 Alerts Architectural Usability Pattern

- **Pattern Name:** Alerts
- **Usability Mechanism**: An alert is a message from the system to the user that a change of state has occurred that the user ought to know about. It can be used, for example, for e-mail arrival, stock control alerts, etc.
- **Solution:**
  - Diagram:



  - Participants:
    - System: represents the element of the system to be checked in order to identify anything of importance for this element. It is responsible for notifying the Alert-manager to check the state of the element to be checked within the system. (1). Depending on what is to be checked, it also sends the request to the part of the system responsible for running the check (3) and, when the check has been run, sends the respective results (if required) to the interface (2).
    - Alert-manager: represents a component of the system that is capable of receiving a checking order and forwarding this order to the part of the system that is capable of processing it. It receives the checking order from one part of the system (1) and forwards this request to the part of the system concerned (3). Finally, if applicable, it displays any alert information that is of interest to the user (2) to check that one or more system components are working correctly.
- **Usability benefits:** Alerts help to keep the user informed about the state of the system with respect to given actions so it provides feedback about the system state.
- **Usability rationale:** Informing the user about given effects of actions that occur in the system raises user *satisfaction*, as users know what is going on. On the other hand, satisfaction may also be affected by the decreased system performance due to alert processing. User *efficiency* may also increase, as they are alerted about given situations and do not have to waste time checking the system state under these circumstances.
- **Consequences:**
  - System *performance* may be affected when processing the respective checking.
- **Related patterns:**

- **Implementation in OO:** This pattern generates an Alert-manager class responsible for managing all the alert checking requests in the system. Each class that represents one of the components affected by the alert will have methods for controlling these alerts at the Alert-manager's request.
- **Example:** the foodstuff code cannot be entered until a check has been run of whether there is a stock of all the ingredients for the selected foodstuff.

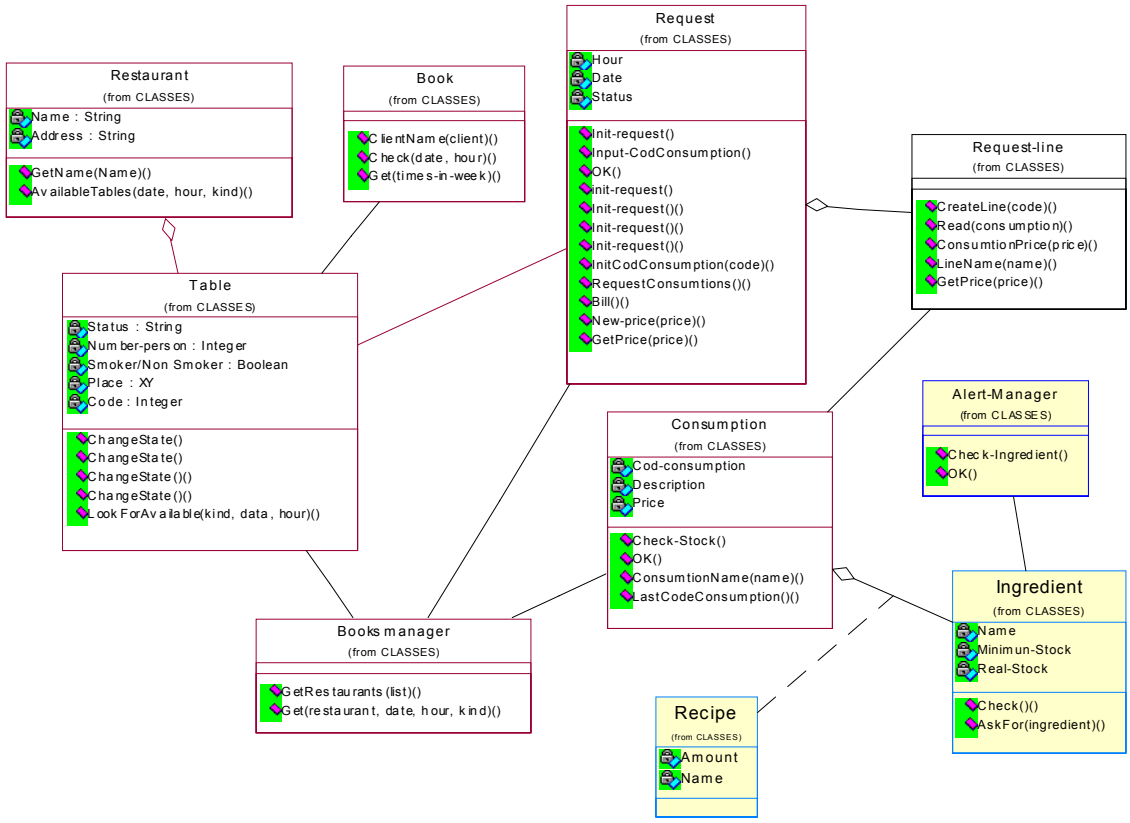


Figure F.11 Class diagram for restaurant management with Alerts mechanism

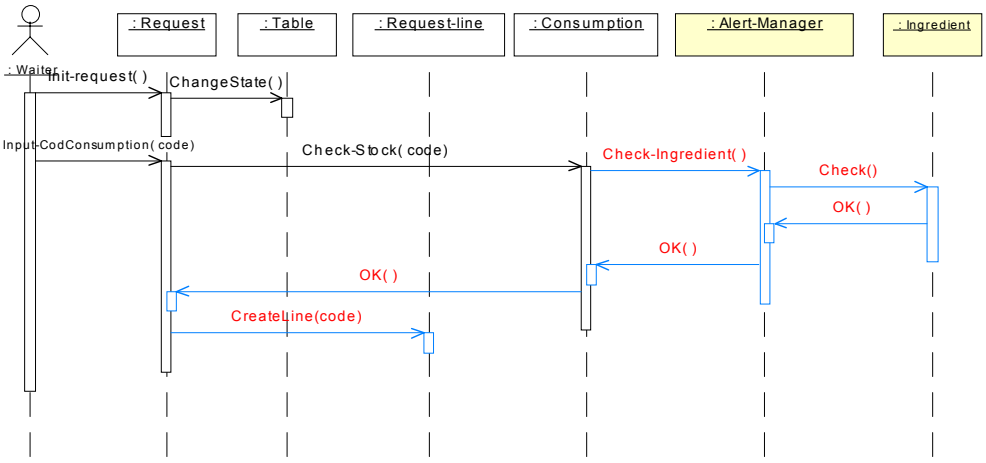
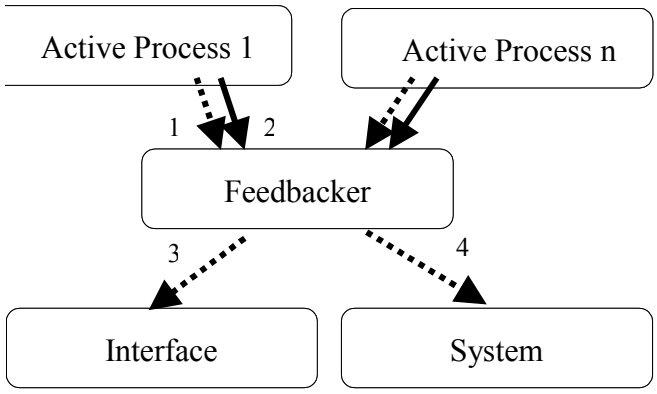


Figure F.12 Sequence diagram for restaurant management with Alerts mechanism

F.7 Status Indication Architectural Usability Pattern

- **Pattern Name**: Status Indication
- **Usability Mechanism**: The user should be provided with information pertaining to the current state of the system.
- **Solution:**
  - Diagram:



  - Participants:
    - Active-process i: this module has been represented more than once, because there may be several processes running simultaneously that request feedback (1) so that it will be each active process that sends the information that it wants to be fed back to Feedbacker (1).
    - Feedbacker: this module receives the request and data (1) (2), which indicates the desired type of feedback and the data to be fed back from each active process. Additionally, it needs to know the recipient of this feedback and will send this feedback either to another part of the system (4) and/or to the interface (3) to inform the user. For some guidelines on how to display this feedback on the interface, for example, how often it should be refreshed or where to place specific information, see [Welie, 00]. These details should be taken into account in low-level design.
    - Interface: it receives the feedback and displays it to user (3).
    - System: this component is optional and represents other parts of the system that must be informed of the feedback (4).
- **Usability benefits:** giving an indication of the system's status provides users with feedback about what the system is doing and what will the result of any action they carry out will be.
- **Usability rationale:** providing feedback gives the user information about what the system is working on and whether the application is still processing or has died. Accordingly, the pattern raises *satisfaction*.
- **Consequences:**
  - This pattern averts additional *system load* by discouraging retries from users [Welie, 00].
  - This pattern increases system *maintainability*, because it channels the feedback better than any existing feedback that is issued indiscriminately by any other system module.
- **Related patterns:**

- **Implementation in OO:** This architectural pattern will generate a Feedbacker class specialised in notifying the user and system about what is happening. This means that all the classes that want to report anything must inform the feedback manager, Feedbacker, so that this then properly distributes this information either within or outside the limits of the system.
- **Example:** the user must be informed about what is happening in the system.

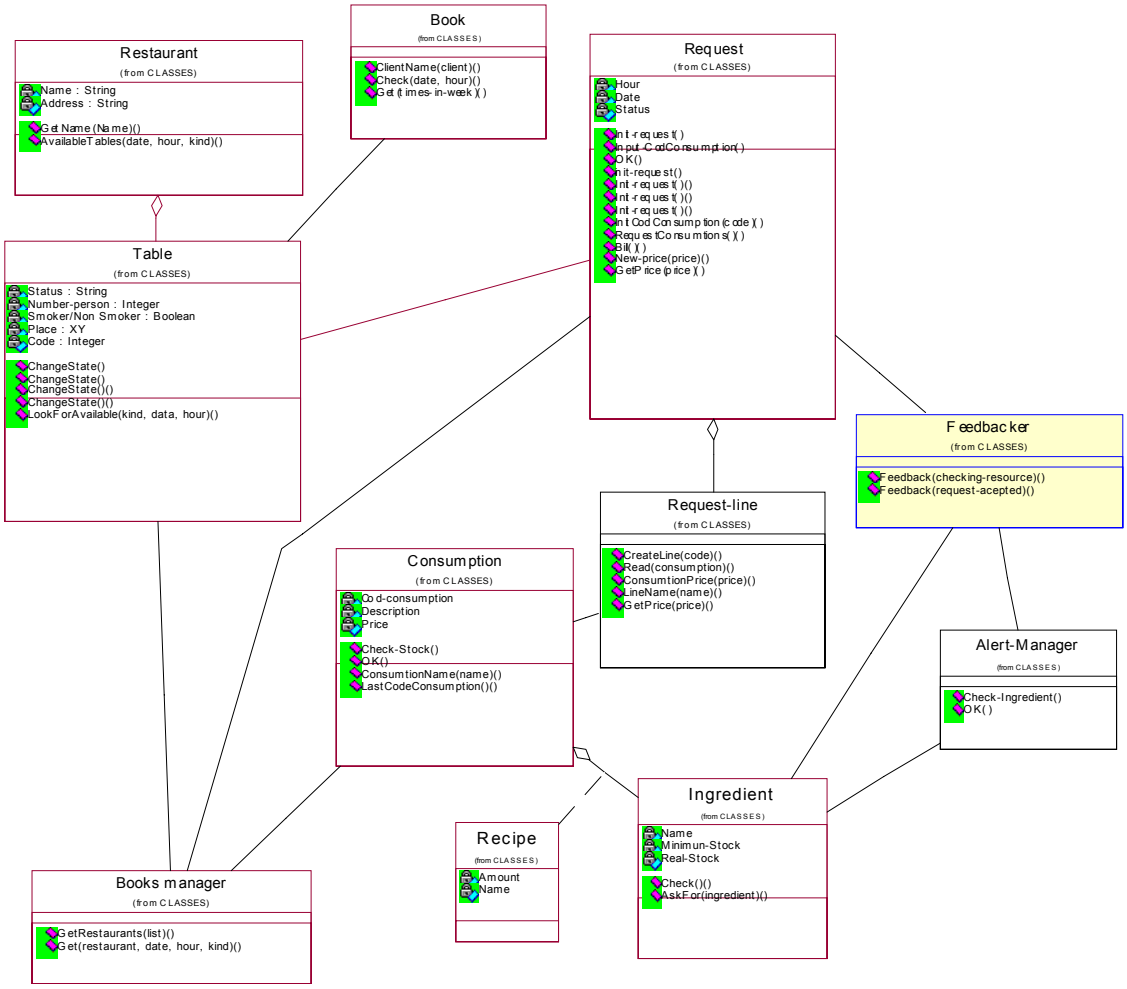


Figure F.13 Class diagram for restaurant management with Status Indication mechanism

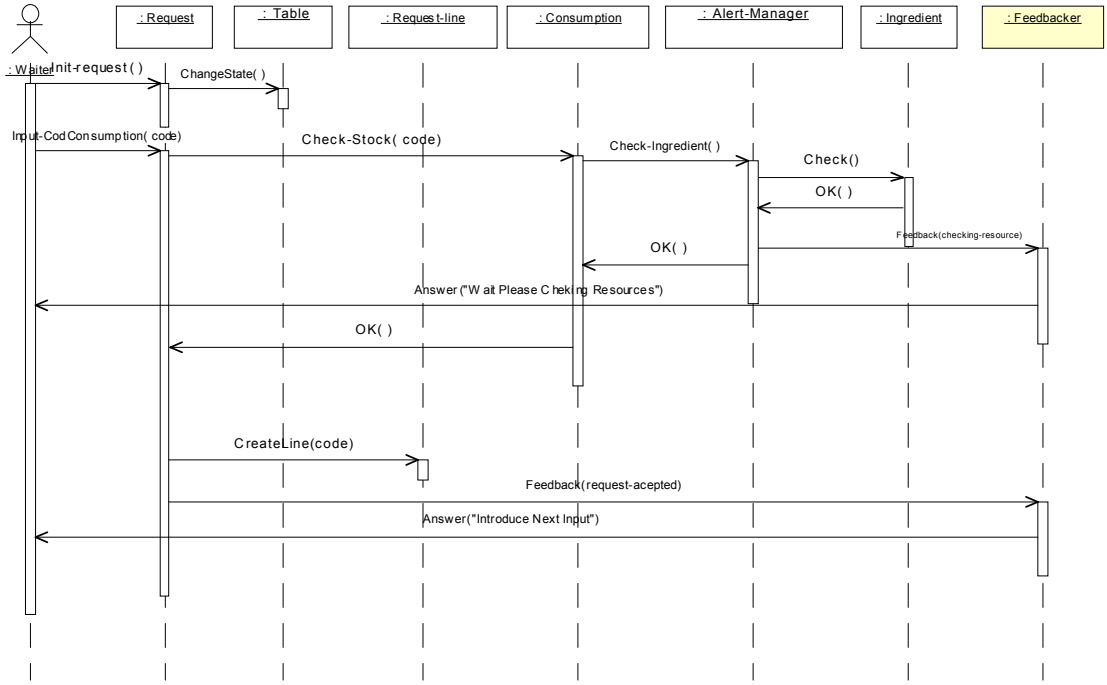
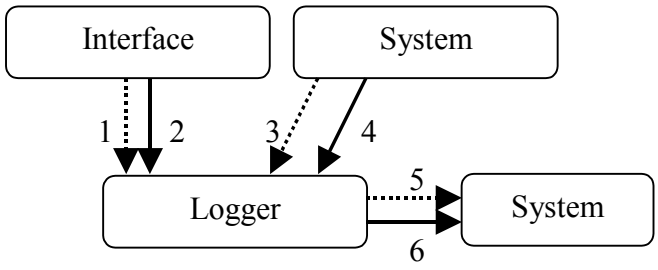


Figure F.14 Sequence diagram for restaurant management with Status Indication mechanism

F.8 History Logging Architectural Usability Pattern

- **Pattern Name:** History Logging
- **Problem:** Record a log of the actions that the user (and possibly the system) has performed in order to allow the user (or system) to look back over what was done previously.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it receives the request to execute an operation in the system, which may contain both the operation and data (1) (2). As we will see later, this execution request can also come from the actual system (3) (4).
    - Logger: this module receives the actions and the data requested by the user or from another part of the system (1) (2) (3) (4) and stores the logged action and data either internally or in another part of the system, in which case it will have to send this action and data to the system (5) (6) to be processed in the respective part of the system.
    - System: this module sends the functions and data that are executed in the system to the logger (3) (4), and also, optionally, if the logger does not store the logged actions internally, sends the information to the part of the system that manages these actions (5) (6).
- **Usability benefits:** Providing a log helps users to see what went wrong if an error occurs and may help them to correct that error. Being able to refer to actions that were carried out previously may help with "recognition rather than recall".
- **Usability rationale:** The provision of this pattern improves *reliability* in use, as it provides users with information on how to correct errors. It also has a positive effect on *learnability*, as the user learns how to work the system.
- **Consequences:**
  - *System performance* will be better if the logged actions are stored in the Logger module rather than another part of the system, as fewer interactions are required.
- **Related patterns:**
- **Pattern implementation in OO:** This pattern will generate a logger class that will send all the actions requested by users through the interface to the action-logged class, as we have implemented the case in which the logger does not store the logged actions internally but in another class.
- **Example:** When an order request is started, the system records that the user has opened an order.

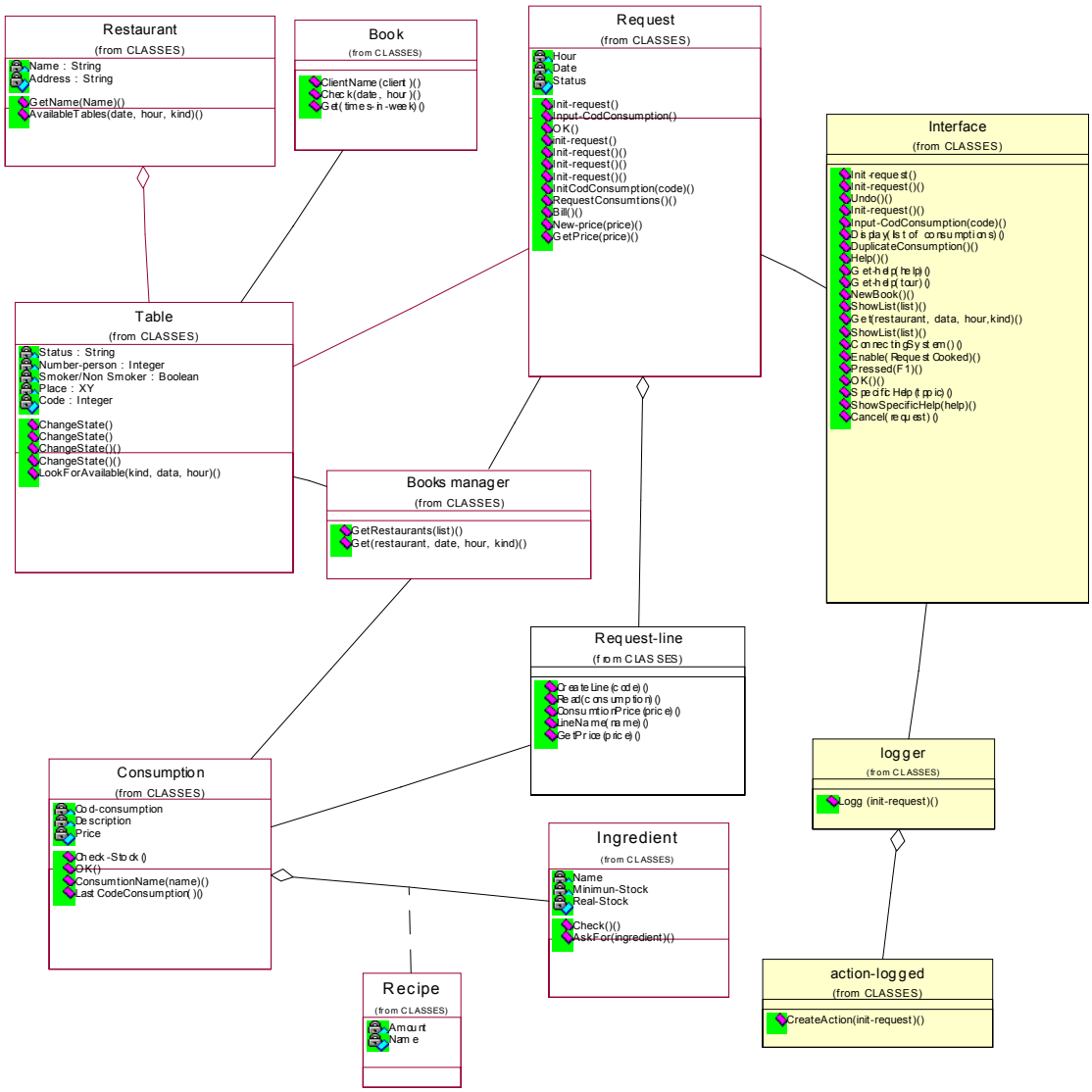


Figure F.16 Class diagram for restaurant management with History Logging mechanism

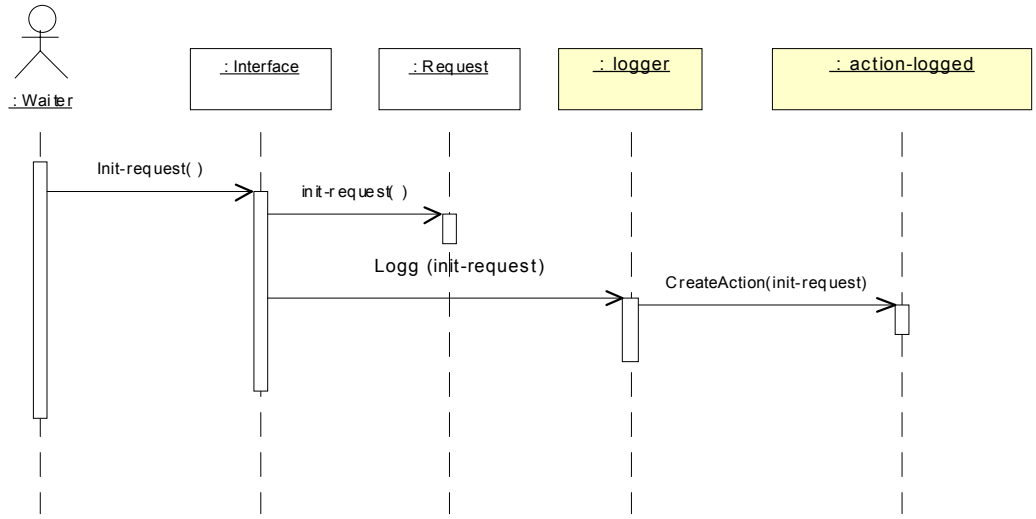
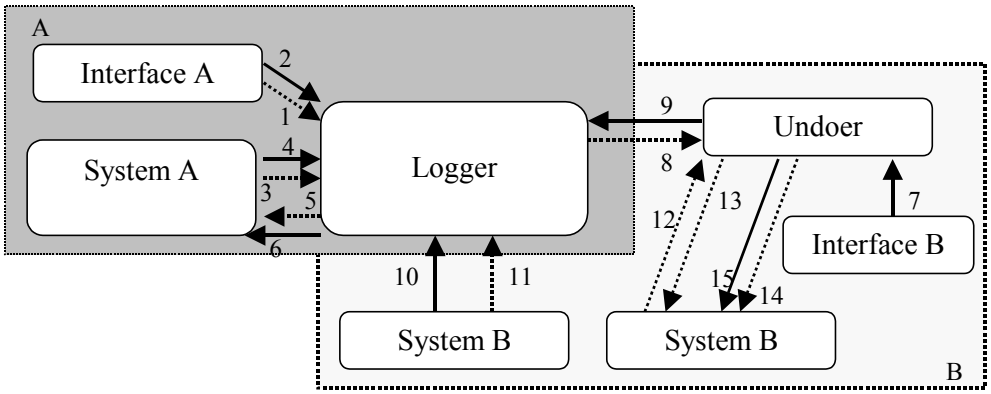


Figure F.17 Sequence diagram for restaurant management with History Logging mechanism

F.9 Undo Architectural Usability Pattern

- **Pattern Name:** Undo.
- **Usability Mechanism** The ability to undo an action and return to the previous state.
- **Solution:**
  - Diagram:



  - Participants: This pattern has two clearly separate parts. These parts have been labelled in the illustration as A and B, respectively. Part A collects the actions performed in the system (the number of actions to be stored will have to be specified when the system is developed) so that they can be later undone. Part B manages the respective undo.
    - InterfaceA: receives the request to execute an operation in the system, which may contain both the operation and data (1) (2). As we will see later, this execution request can also come from the actual system (3) (4).
    - SystemA: this module sends the functions and data executed in the system to the logger (3) (4) and also, optionally, if the logger does not store the actions internally, will send the information to the part of the system that manages these actions (5) (6).
    - Logger: this module receives the actions and the data requested by the user or from another part of the system (1) (2) (3) (4) and stores the logged action and data either internally or in another part of the system, in which case it will have to send this action and data to the system (5) (6) to be processed by the respective part of the system. Logger receives the undo request from Undoer (9) and, if the logged actions are stored in the logger, it then sends them one by one to Undoer (8). If they are not stored in the logger, it will receive both the data and the operation to be undone from another part of the system that we have named System B through (11) and (10), respectively.
    - Interface B: receives the undo request and sends it to Undoer through (7).
    - Undoer: sends the undo request to logger (9) and also sends each of the actions to be undone that it receives from logger to System B (13), as well as receiving the opposite operation to the one performed from System B (12). When it knows which opposite operation is to be performed, it sends the operation to System B along with the data associated with the operation in question through (14) and (15).
    - System B: it will search the system for both the action performed and the data associated with this operation (10) (11) if the data are not stored internally in the

logger. It receives the actions to be undone (13) and provides the opposite operation (12) (for which purpose it will have to store what the opposite is for each action, see implementation section, for example). The opposite action and the respective data will be sent to the respective part of the system (15) and (14).

- **Usability benefits:** Providing the ability to undo an action helps the user to correct errors if they make a mistake. It helps the user to feel that they are in control of the interaction.
- **Usability rationale:** This patter improves *reliability*, as it makes it possible to correct any errors made by the system and also improves user *efficiency*.
- **Consequences:**
- **Related patterns:** History Logging is equivalent to part A of this pattern. Therefore, if undo is provided, it would also be advisable to provide History Logging without any additional cost.
- **Pattern implementation in OO:** This pattern will generate an "undoer" class responsible for triggering the entire undo process. Additionally, the listener and action-done classes appear, which are used to store the actions that are performed as the system operates. It is also necessary to include a "system-action" class to establish what the opposite is for each action that can be undone through the "is-the-opposite" relationship.
- **Example:** the user can push the undo button.

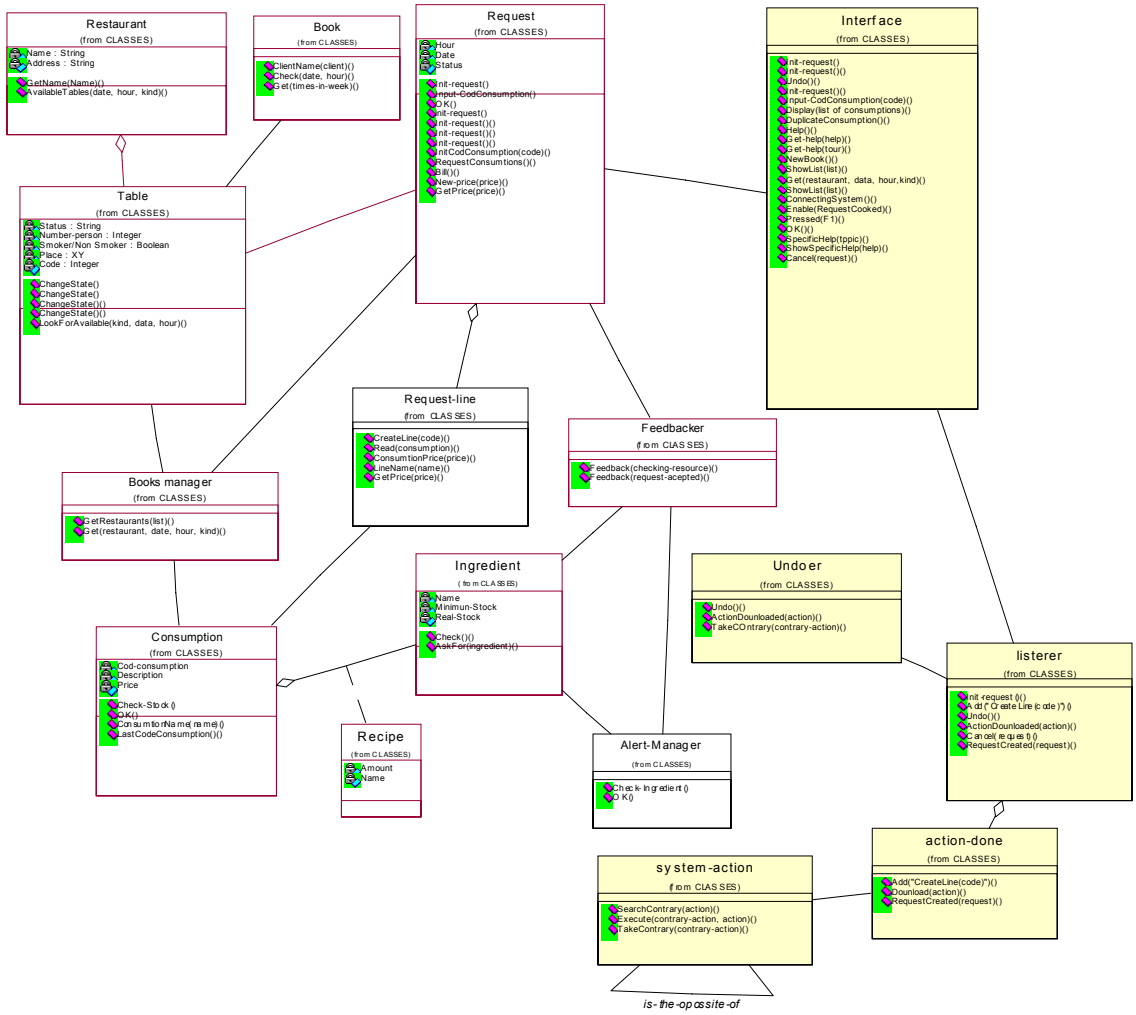


Figure F.18 Class diagram for restaurant management with Undo mechanism

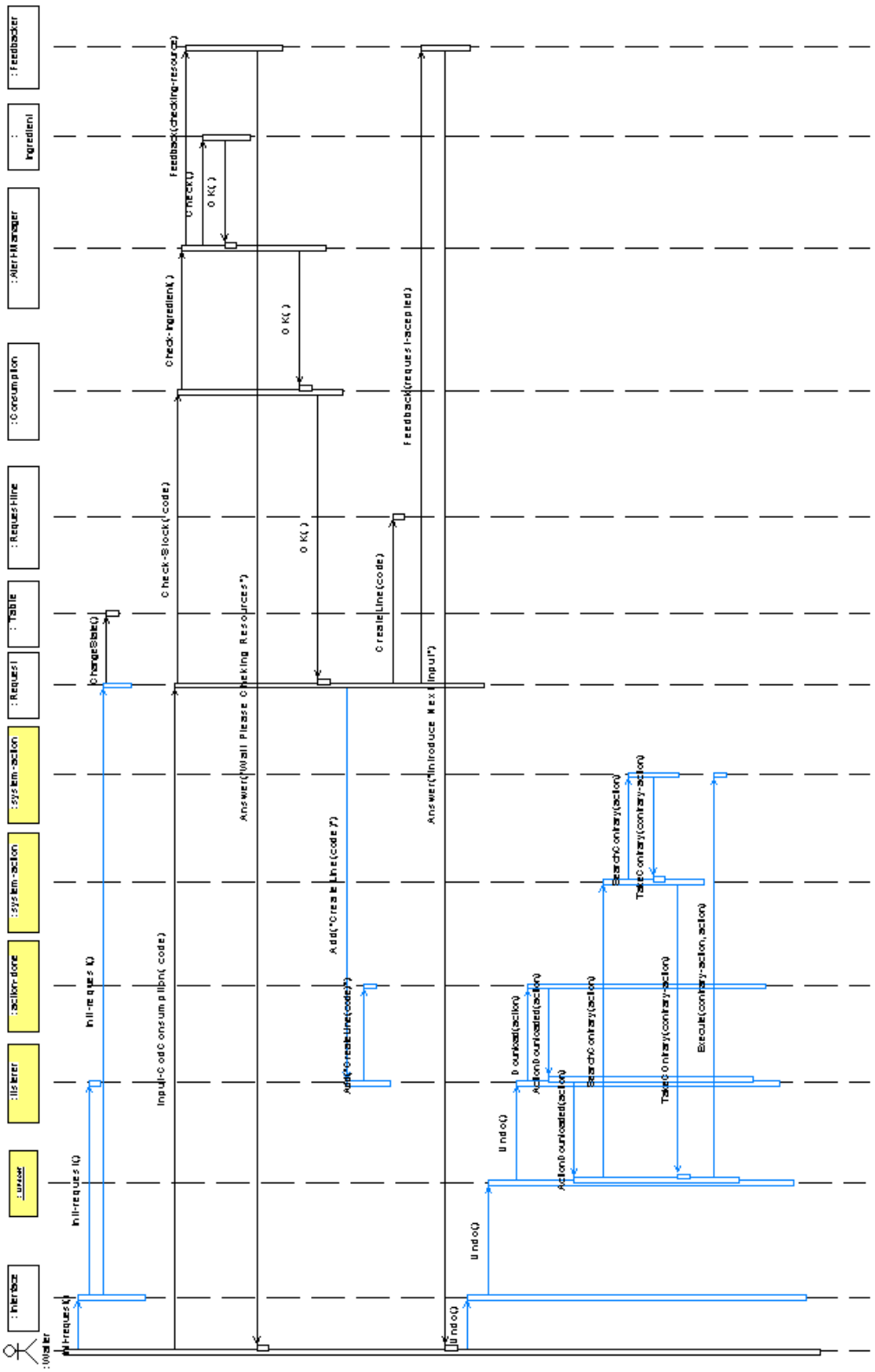
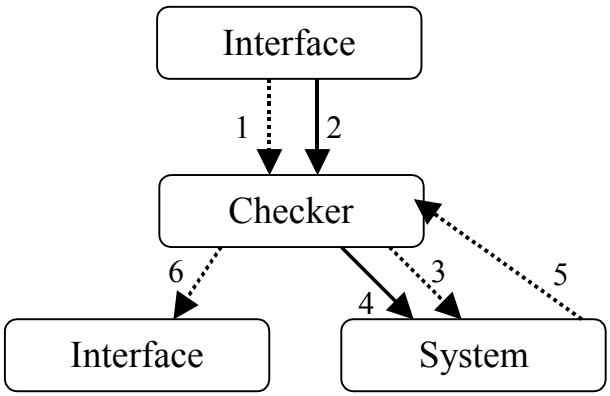


Figure F.19 Sequence diagram for restaurant management with Undo mechanism

F.10 Form or Field Validation Architectural Usability Pattern

- **Pattern Name:** Form or Field Validation.
- **Usability Mechanism**: If a user is entering multiple items of data on one screen, it is possible to check that each field contains valid data either all at once when the "submit" or "ok" button is pressed (form validation), or individually each time a data item is entered (field validation). With form validation, one invalid entry may lead to the whole form having to be filled in again.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends a data set (1) and the function requested by the user (2) to Checker for validation. Additionally, after data validation, it will receive error data or OK from the Checker to be displayed to the user if the system is to be designed this way (6).
    - Checker: it collects an operation requested by the user through the interface (2) as well as a data set (1). This module can be designed to validate the data or to send the data to another system component for validation (3) (4). In the latter case, it also receives the result of the validation (OK or error) (5) and, in any case, will send the result of the validation to the user if so required (6).
    - System: this component will be optional and will only exist if the Checker is not capable of validating the data. If necessary, it receives both the function and the associated data for validation from Checker (3) (4) and, after validation, returns the result of the validation to Checker.
- **Usability benefits:** This pattern relates to a provision for error prevention.
- **Usability rationale:** The application of this pattern reduces the number of errors, increasing *reliability* and user *efficiency*.
- **Consequences:**
  - System *performance* might be affected depending on when the validation is done. In client/server applications, in particular, validation should be done whenever possible at the client site in order to avoid interactions between both parts. Additionally, better performance might be achieved if validation is done inside the Checker component, which, however, violates the encapsulation principle in the object-oriented paradigm.
- **Related patterns:**
- **Pattern implementation in OO:** This pattern will generate a "validation manager" responsible for asking the specific validator to validate the sent data depending on the operation received.

There will be as many specific validators (e.g., foodstuff-validator) as different data sets to be validated. All this applies if the implementation takes into account that the checker does not run the validations internally.

- **Example**: The system should validate the foodstuff code when it has been entered by the waiter and before it is copied to the order.

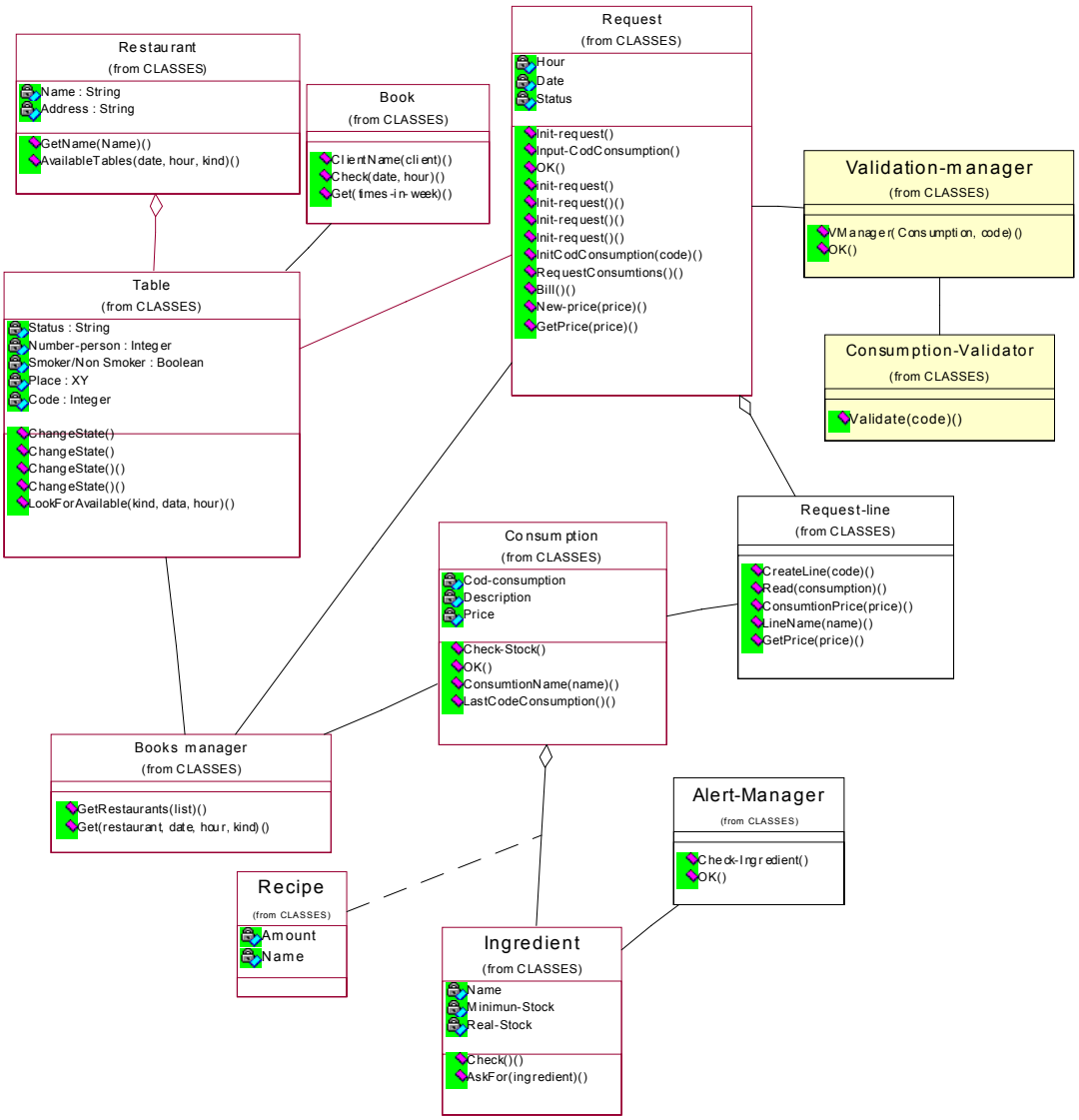


Figure F.21 Class diagram for restaurant management with Form or Field Validation mechanism

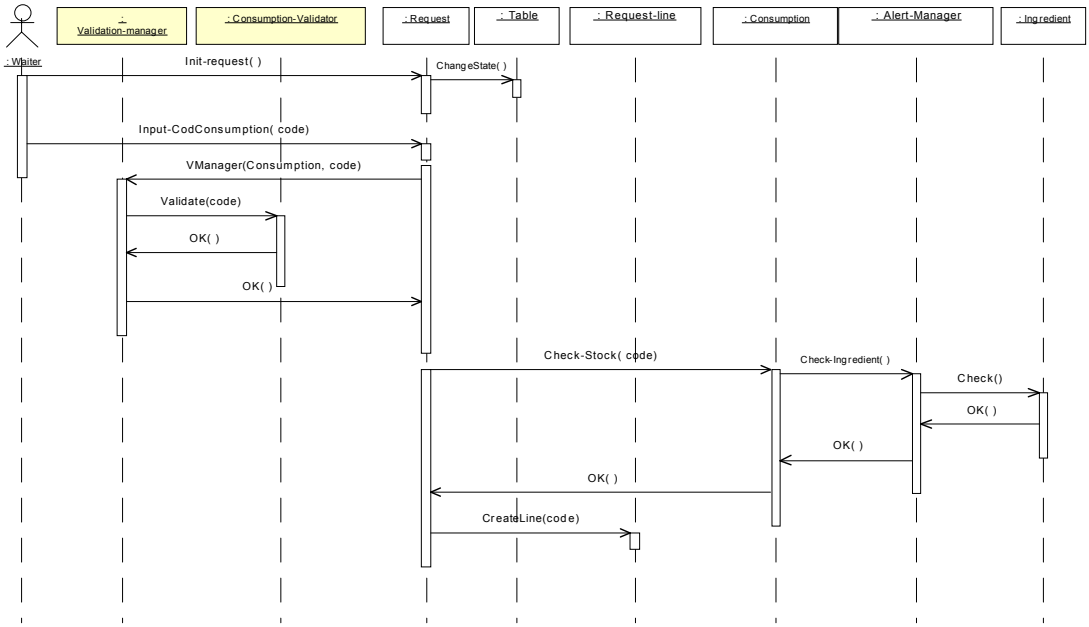
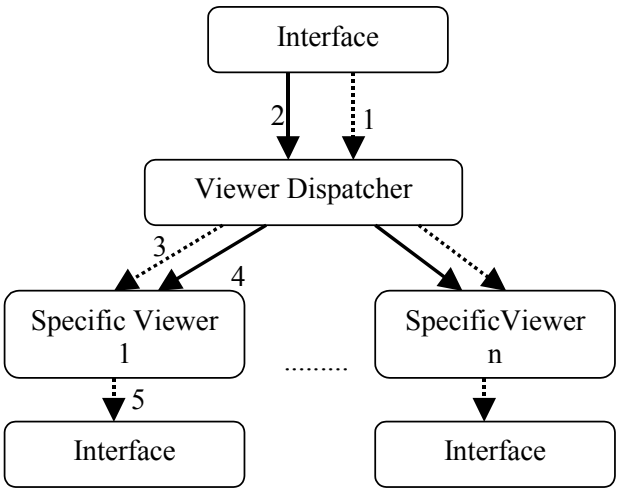


Figure F.22 Sequence diagram for restaurant management with Form or Field Validation mechanism

F.11 Provision of Views Architectural Usability Pattern

- **Pattern Name:** Provision of Views
- **Usability Mechanism**: The system must provide users with different views so that they can see what data they are working on at any time.
- **Solution:**

  - Diagram:



  - Participants:
    - Interface: it sends the data received (1) and the specific function requested by the user (2) to viewer-dispatcher. Additionally, when the data have been transferred to the specific viewer that knows how to interpret them, they are displayed by the interface (5). For information about how to present some views in the interface, see [Welie, 00]
    - Viewer Dispatcher: it receives the data (1) and the requested function (2) and, depending on this information, decides which viewer should interpret the operation and data. These (3) and (4) are sent to the respective Specific Viewer.
    - Specific Viewer i: it receives a request (4) and data to be viewed (3), which it interprets as befits the viewer in question, sending them to the interface (5).
- **Usability benefits:** Having data-specific views available at any time provides the user with guidance and will contribute to error prevention.
- **Usability rationale:** Error prevention improves user efficiency, in which case satisfaction will also be increased. Additionally, specific viewers usually consume fewer resources than the original action, for which reason users will also work with the system more efficiently.
- **Consequences:**
  - Having different specific viewers increases system maintainability, as adding or modifying a view has less impact on the system.
- **Related patterns:**
- **Pattern implementation in OO:** This pattern generates a "viewer-manager" class that is responsible for selecting the specific viewer to be used in each case. Additionally, a specific class appears for each viewer.

- **Example**: The system should be able to provide the customer with the list of things ordered so far at any time while the order is being placed.

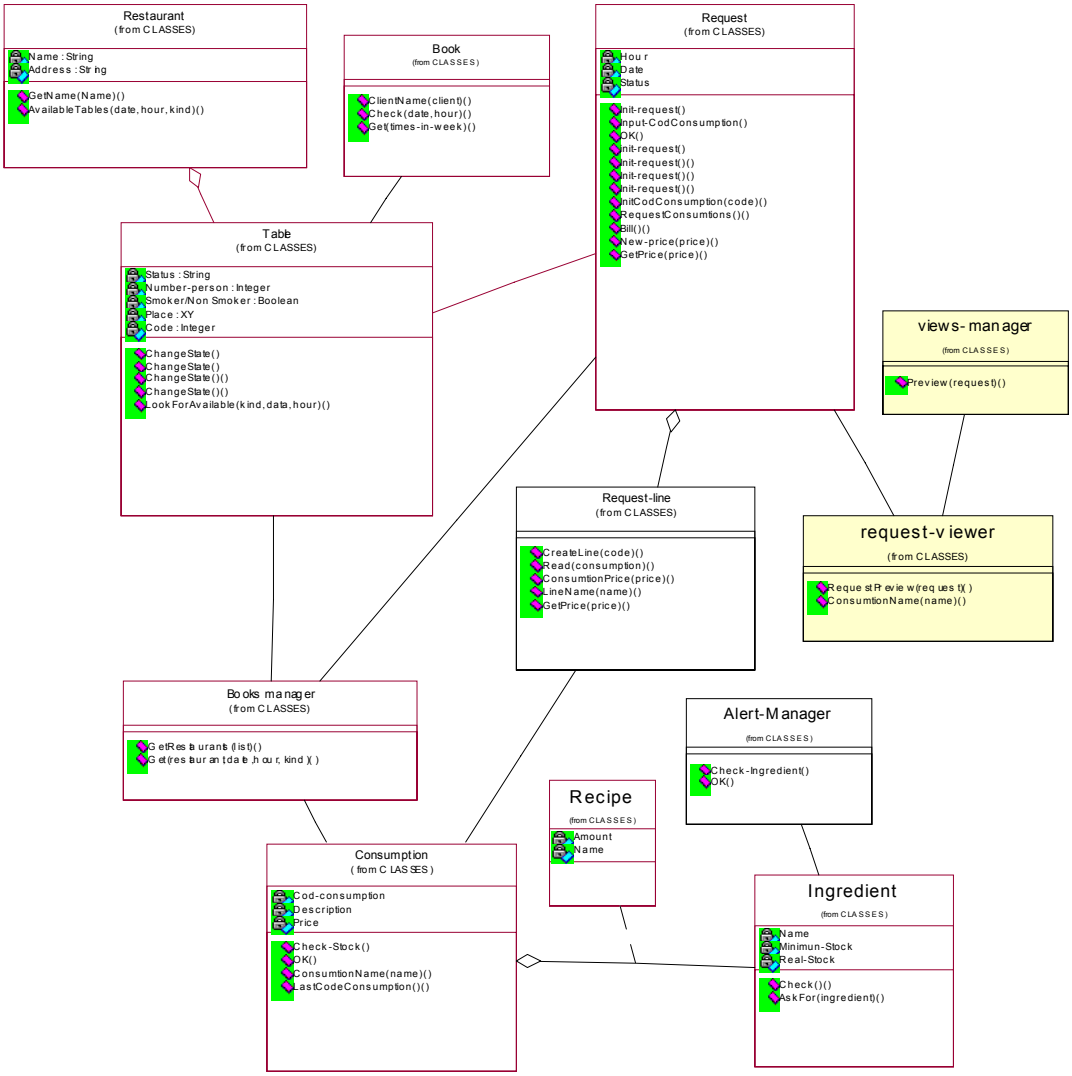


Figure F.23 Class diagram for restaurant management with Provision of Views mechanism

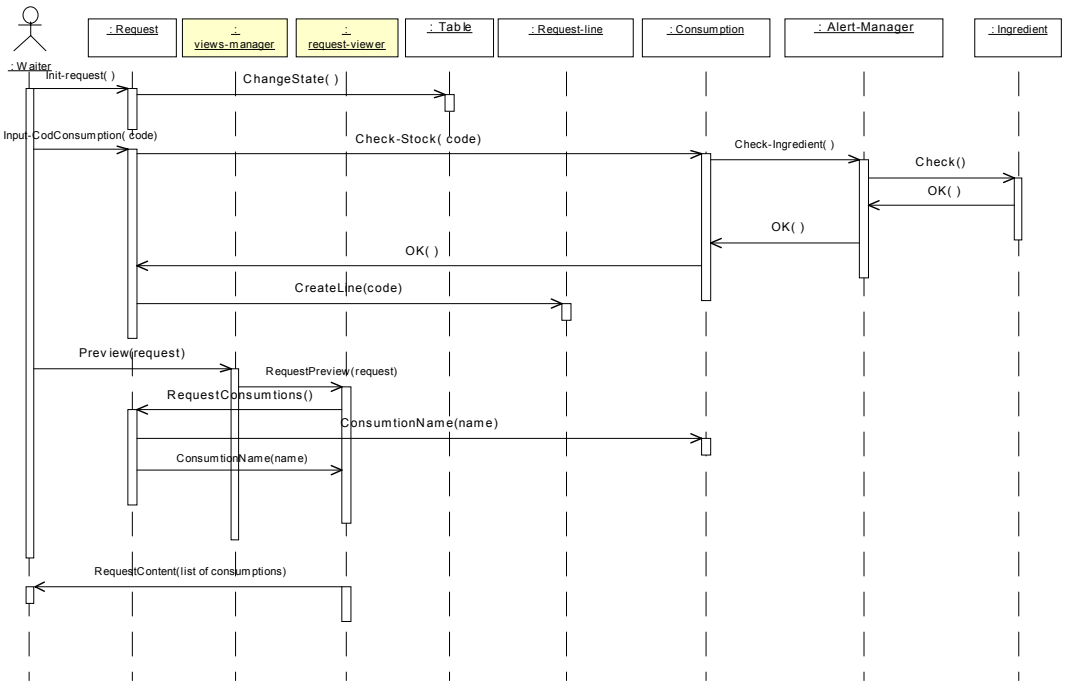
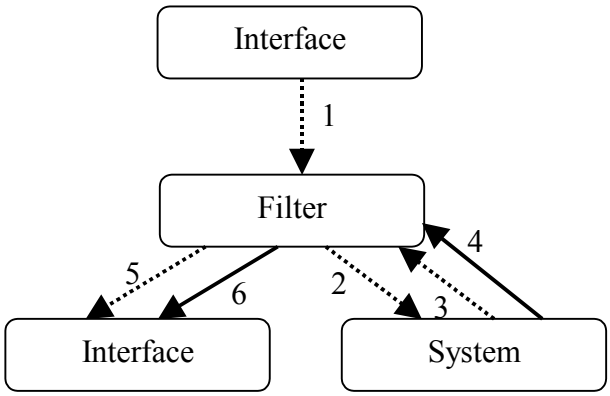


Figure F.24 Sequence diagram for restaurant management with Provision of Views mechanism

F.12 Workflow Model Architectural Usability Pattern

- **Pattern name:** Workflow Model
- **Usability Mechanism**: Modelling workflow provides different users with only the tools or actions that they need to perform their particular tasks.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends the data related to the user who is trying to access the system (1) to the system. Additionally, the interface receives the data and operations (5) (6) that make up the interface for the user in question from Filter.
    - Filter: it receives the type of user who wants to connect to the system (1) from the interface. Additionally, if it does not store all the functionality that should be associated with each user internally, it sends the data about the user in question to another system component (2) and receives both the data (3) and the operations (4) to which this user should have access from this component. When it has this information, it then passes it on to the interface for proper display (5) (6).
    - System: this component is optional and will only exist if the Filter is not capable of storing the functionalities associated with each system user internally. Accordingly, this component receives the data on the user type who has connected from Filter (2) and returns both the data and operations that this user type can access from the interface (5) (6) to Filter.
- **Usability benefits:** Targeting the user interface specifically to each user, depending on the tasks that they need to perform in the workflow, minimises the user's cognitive load and prevents errors.
- **Usability rationale:** This pattern improves user *efficiency* and *reliability*, as the user will only see the information and tasks corresponding to the operations to be done.
- **Consequences:**
- **Related patterns:**
- **Pattern implementation in OO:** If the Filter stores only the relationship between users and the functionality which each one can access, this pattern generates the filter class, which is responsible for building the interface suited for the user type that has connected to the system. Additionally, it will generate a "user" class, where all the possible system user types are stored, and a "system-function" class, where the different functionalities provided by the system are stored. These two classes will have to be linked by an association that determines what function each user can

perform. For example, supposing that user 1 can only access F1 and F3, then the "user" and "system-function" relationship would contain the following information:

	SYSTEM FUNCTIONALITY		
USER TYPE	F1	F2	F3
USER 1	X		X
USER 2		X	
USER 3	…..	……	……

- **Example:** when the cook connects to the system, the only enabled function will be enter as cooked when he has finished cooking an order.

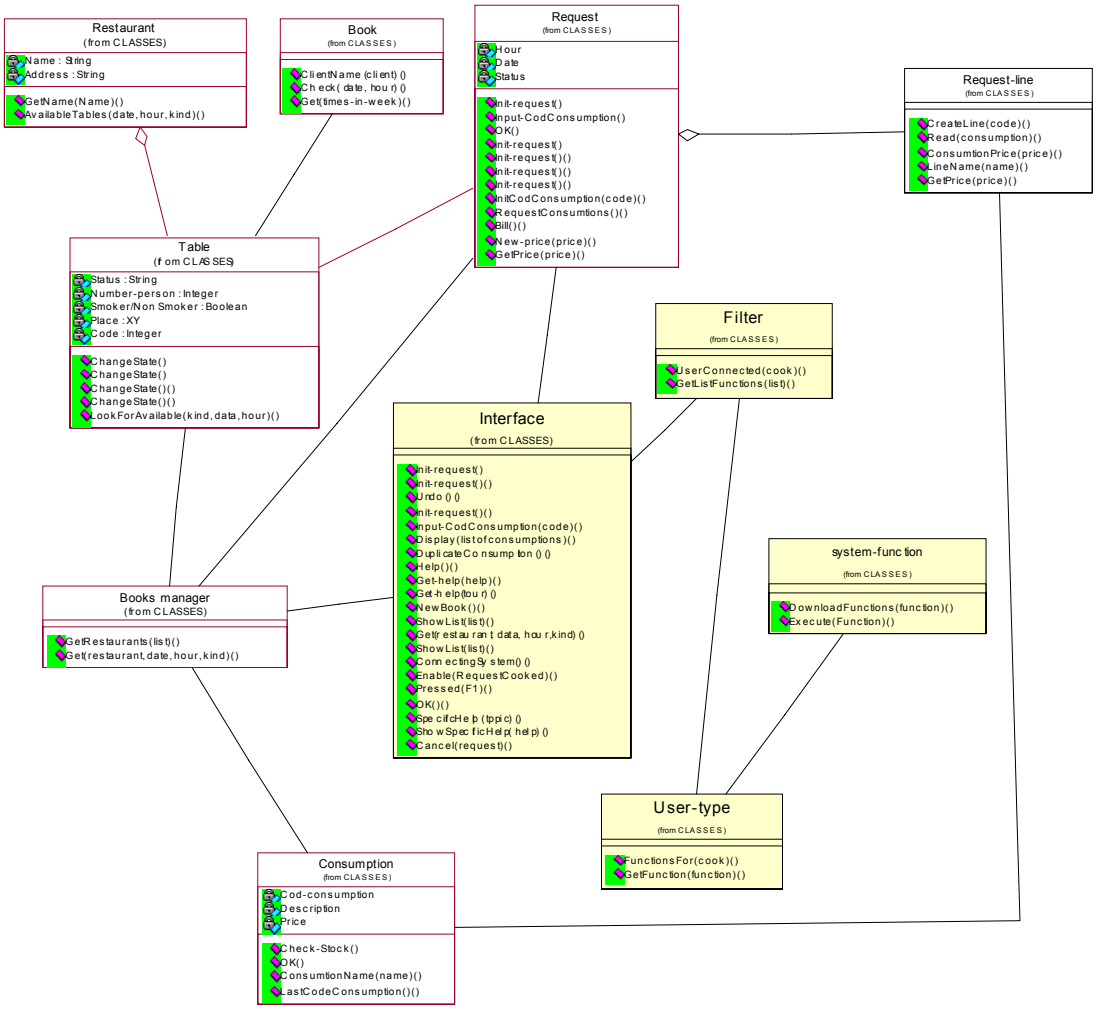


Figure F.25 Class diagram for restaurant management with Workflow Model mechanism

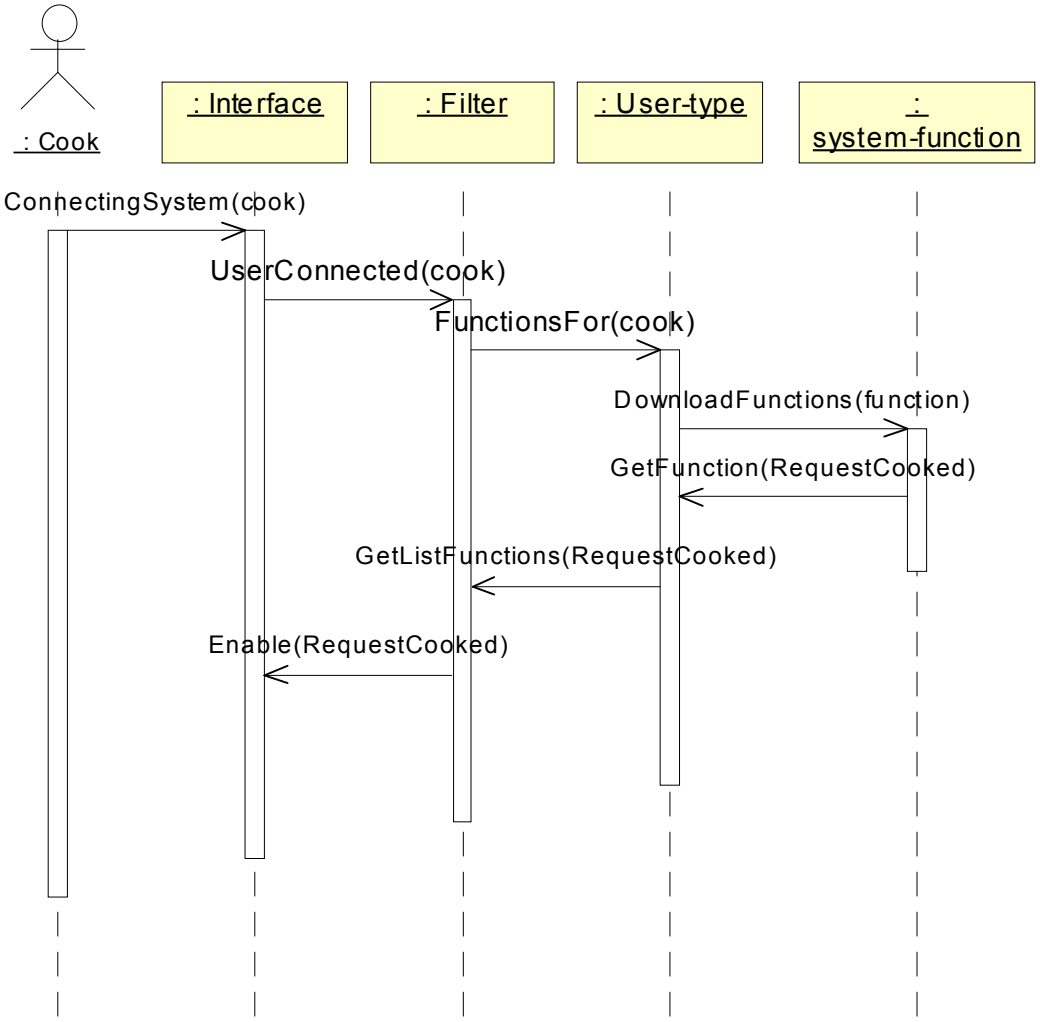
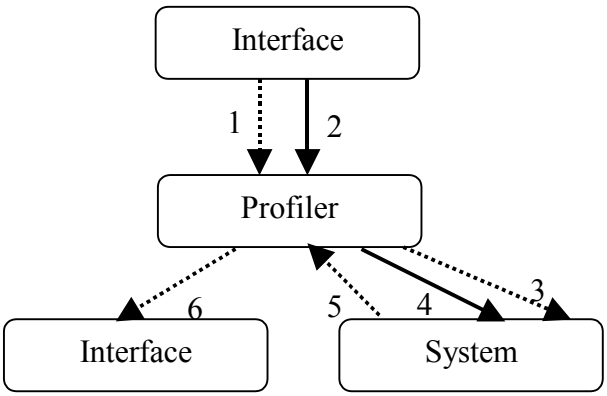


Figure F.26 Sequence diagram for restaurant management with Workflow Model mechanism

F.13 User Profiler Architectural Usability Pattern

- **Pattern Name:** User Profiler
- **Usability Mechanism**: The software system builds and records a profile of each user so that specific system attributes (concerning the layout of the user interface, the data or options to show, etc.) can be set and reset each time that a different user accesses the system. Different users may have different roles and require different things from the software.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface:
      - For profile information creation, it sends both the data (1) and the operation (2) that the user defines for his system to the profiler.
      - For profile retrieval, the interface sends the profile data (1) to the profiler. Additionally, profiler sends the data associated with this profile to the interface.
    - Profiler: .
      - For profile information creation, it receives the data (1) and the operation (2) that the user defines for his system from the interface. If it is not capable of storing this profile information internally, it will send it to another system component through (3) and (4).
      - For profile retrieval, it receives the data of the profile to be retrieved (1) from the interface. If it does not store the profile information internally, it will ask another system component to process the requested information and/or operation (3) (4) and will receive the information associated with the required profile (5) from this system component. Then, if this information is to be displayed by the interface, it will send it to the interface through (6).
    - System: this component is optional and will only exist if profiler is not capable of storing the information associated with each system profile internally. It receives the data and/or operations of the required profile type (3) (4) from profiler and sends the data associated with this profile (5) to profiler.
- **Usability benefits:** Providing the facility to model different users allows a user to express preferences , thereby increasing system adaptability.

- **Usability rationale:** Expert users can tweak the application for their particular purposes, which increases satisfaction and possible performance, but this solution decreases memorability and learnability [Welie, 00].
- **Consequences:**
- **Related patterns:**
- **Pattern implementation in OO:** This pattern generates a "profiler" class that is responsible for performing given operations depending on who the requested profile belongs to.
- **Example:** The system should be able to identify the customer who is making the order at a given table so that he can be given personalised treatment depending what type of customer it is.

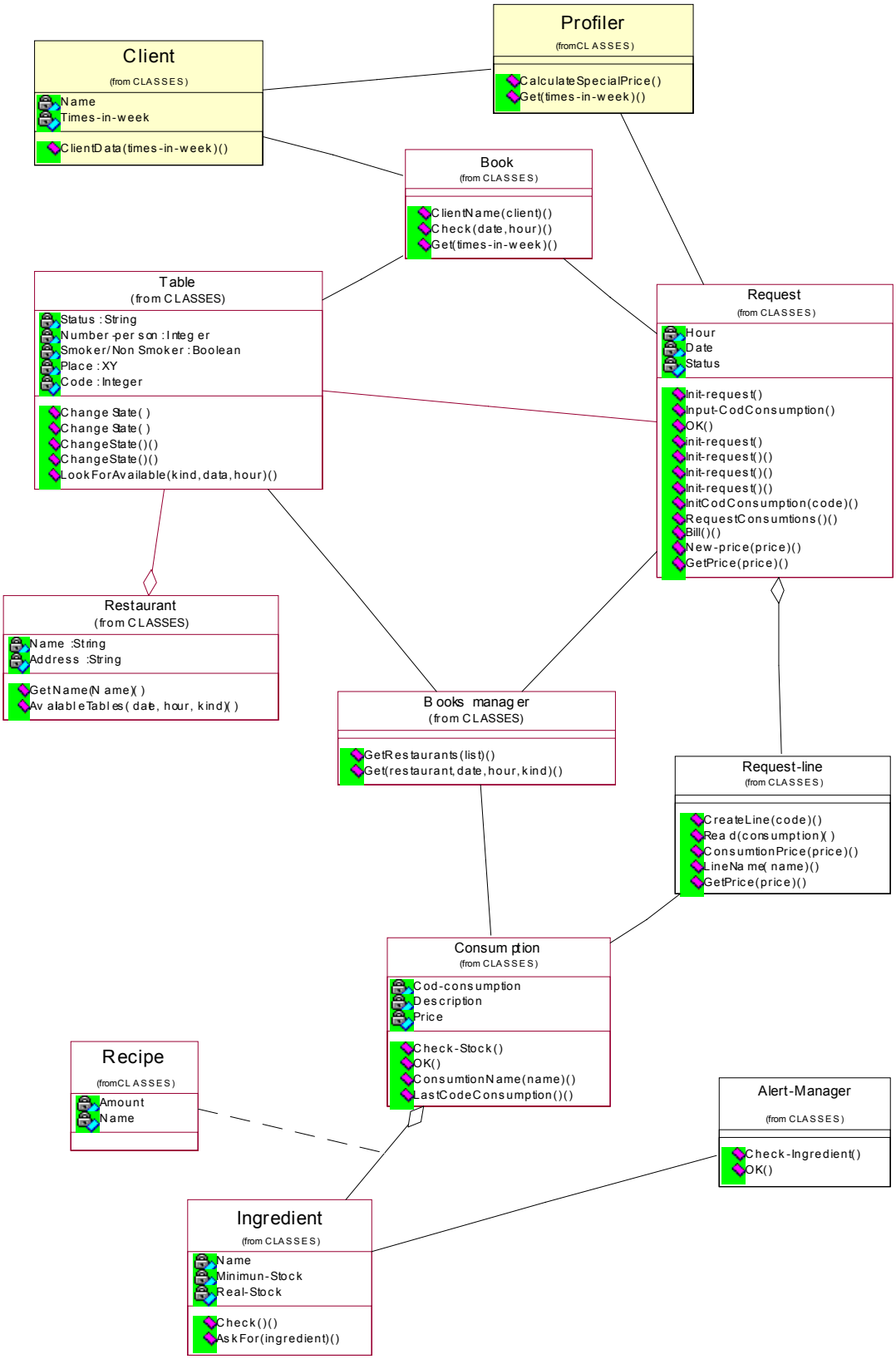


Figure F.27 Class diagram for restaurant management with User Profiler mechanism

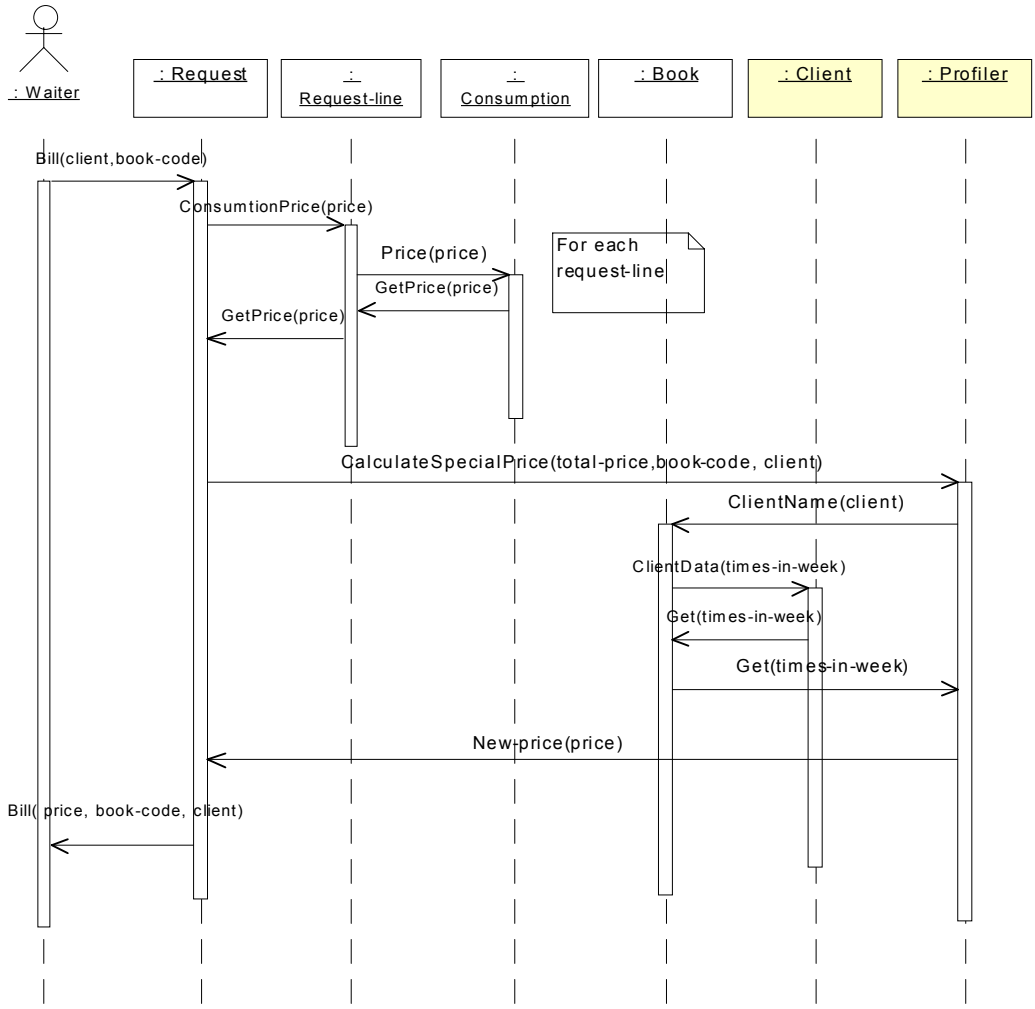
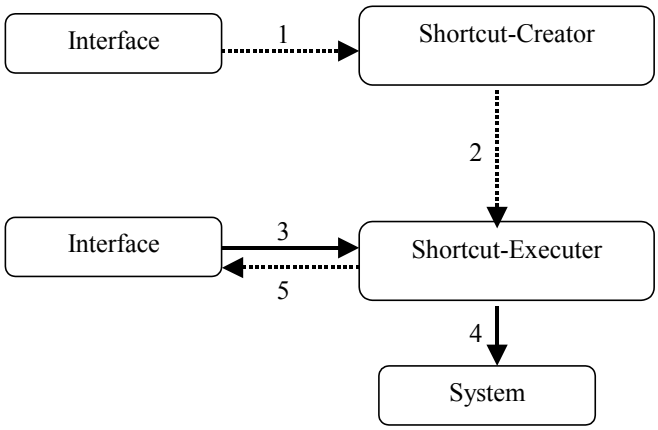


Figure F.28 Sequence diagram for restaurant management with User Profiler mechanism

F.14 Shortcuts Architectural Usability Pattern

- **Pattern Name**: Shortcuts
- **Usability Mechanism**: A shortcut allows an experienced user to activate a feature that may be hidden "under the surface" of the interface with one quick manoeuvre.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends a data set (1) corresponding to a given system function, as well as the key combination that activates this function, to Shortcut-creator. Additionally, if a shortcut is to be executed, it sends a key combination (3) to the Shortcut-executor. When the shortcut has been executed, it will receive the result of the requested functionality or error if it cannot be executed (5) from Shortcut-executor.
    - Shortcut creator: it fills in a sort of array in which the name of the shortcut, the commands that activate it and the system function to be activated with these quick commands are stored. For this purpose, it receives a data set and the function to be executed when these keys are combined (1) from the interface, which it sends to Shortcut-executor for storage (2).
    - Shortcut executor: it receives a set of commands (3) from the interface and checks whether they match a set of commands associated with a given function. If the command set matches a system functionality, it requests the system to execute the function associated with this shortcut (4). In any case, whether they match a function or not, it sends the result of executing this function to the interface through (5).
    - System: it receives the order to execute the function associated with this key combination (4) from shortcut-executor.
- **Usability benefits:** The provision of shortcuts allows the system to match the user's level of expertise. An experienced user will use the shortcut, whereas a novice will navigate a longer path through the user interface, perhaps receiving more guidance. Shortcuts also provide the user with explicit control of the system.
- **Usability rationale:** This pattern enables expert users to work with the system more *efficiently*. Nevertheless, for non-expert users, this option might decrease *learnability*. Also shortcuts might be inefficient for long-term *memorability*.

- **Consequences:**
  - This pattern improves system *performance*, users do not need to follow a specific sequence of steps to perform an action.
  - *Performance* is also better if Shortcut-creator stores the shortcut information internally rather than in other system modules.
- **Related patterns:**
- **Pattern implementation in OO:** This pattern will generate a "shortcut"· class associated with another "key" class responsible for recording the keys and the order in which they should be pressed to activate a given system functionality. Additionally, there will be "system-function" class associated with the key set defined for each shortcut. The "shortcut", "key" and "system-function" classes represent the shortcut-executor module, because this example shows a shortcut that has already been created.
- **Example:** the waiter presses F1, which corresponds to the function that tells the waiter to go and collect a given order that has now been cooked.

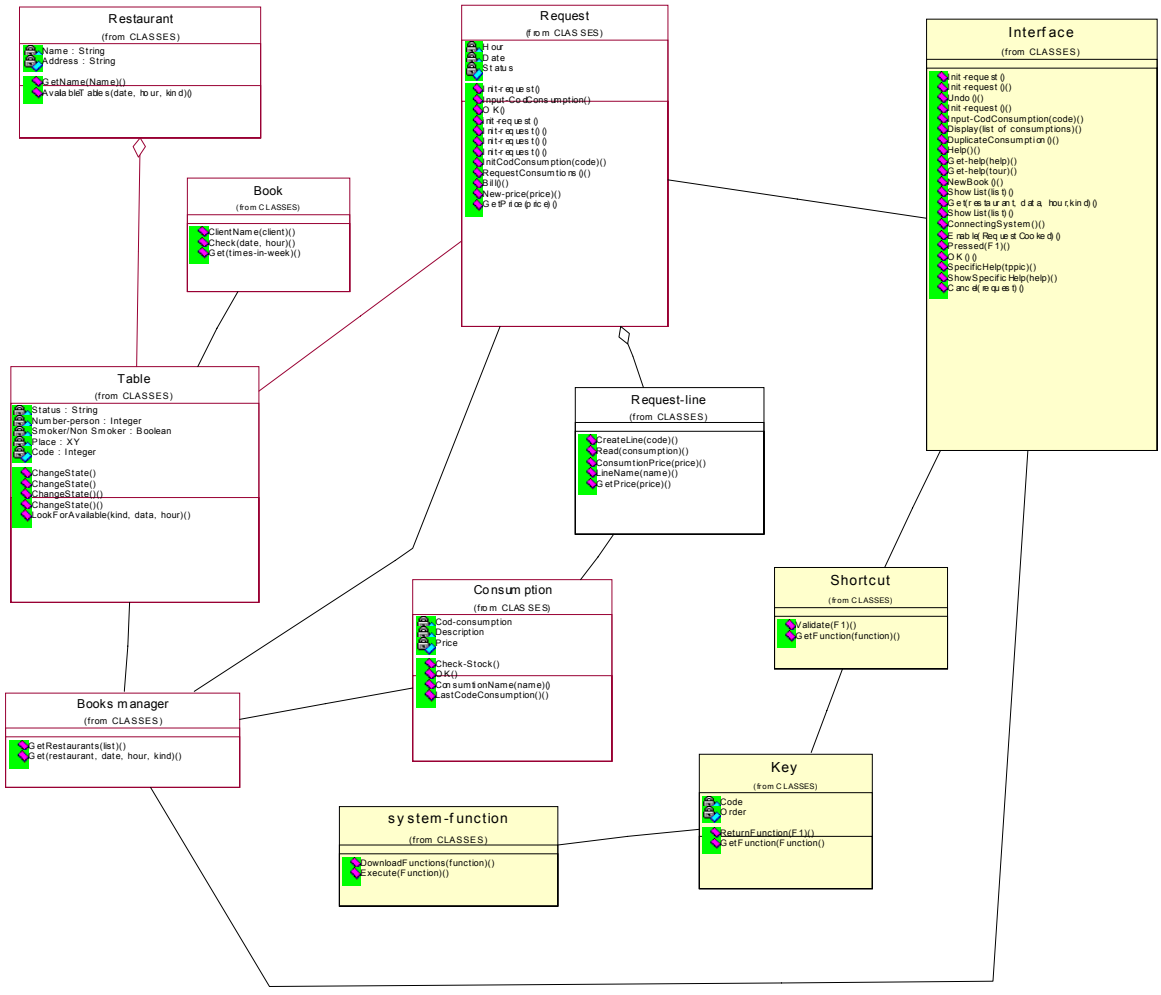


Figure F.29 Class diagram for restaurant management with Shortcuts mechanism

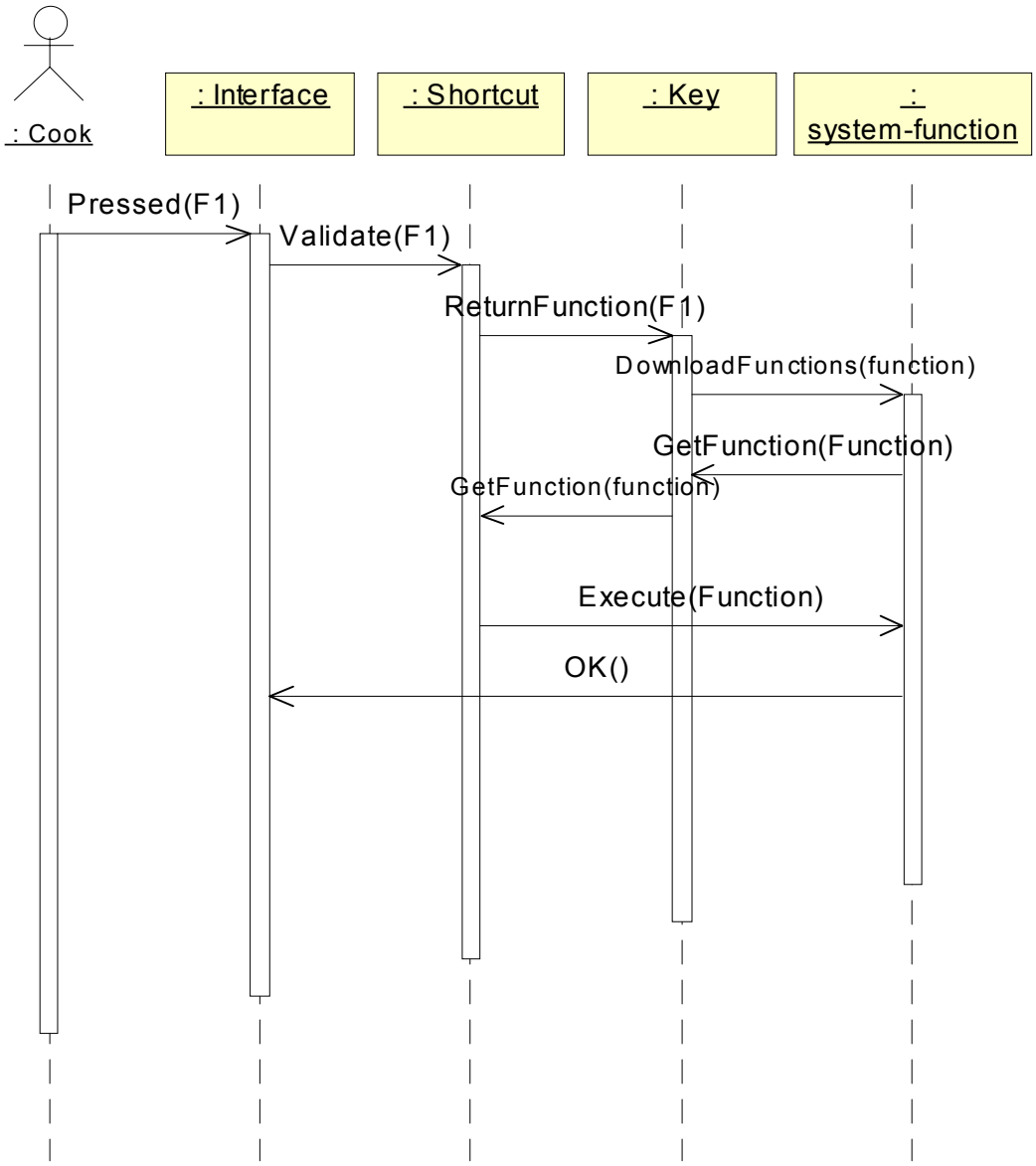
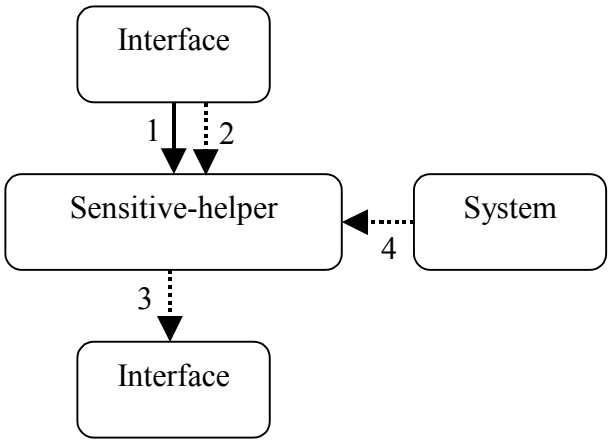


Figure F.30 Sequence diagram for restaurant management with Shortcuts mechanism

F.15 Context Sensitive Help Architectural Usability Pattern

- **Pattern Name**: Context Sensitive Help.
- **Usability Mechanism**: Context-sensitive help monitors what the user is currently doing and supplies information relevant to the completion of the task in question.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it warns the sensitive-helper through (1) that the cursor is on top of the element specified in (2). Additionally, it will display the help information it receives from Sensitive-helper (3).
    - Sensitive-helper: it identifies the help associated with a given element. This component receives the signal (1), which alerts it to the need to show help about the specified element through (2), from the interface. If the help is not stored internally in this component, this help will be provided by another part of the system through the information flow from System (4). When it has the help data, it informs the interface through (4).
    - System: this component is optional and represents a part of the system in which the help will be stored if the Sensitive-helper is not capable of storing it internally. In this case, System will provide the help to the Sensitive-helper through (4).
- **Usability benefits:** The provision of context sensitive help can give the user guidance and will prevent errors made by the user.
- **Usability rationale:** This pattern will improve *reliability* and *efficiency*, as well as *learnability* for non-expert users.
- **Consequences:**
  - Performance will improve if the help is stored in Sensitive-helper.
- **Related patterns:** Guided-helper and Standard-helper, because both helps can be stored in the same "Help" class, furnished by special methods for properly handling each of the two help types provided by Standard-helper and Guided-helper.
- **Pattern implementation in OO:** The Interface component will generate one or more classes. The Sensitive-helper component will generate a class that will have an attribute containing the help, or a pointer to another place (class) that contains this help if it is not stored in the class itself. The

first option corresponds to the case in which the help is saved in the Sensitive-helper and the second to the case in which it is saved in another system component.

- **Example:** the system must provide the user with sensitive help when the cursor is positioned over certain elements in the interface.

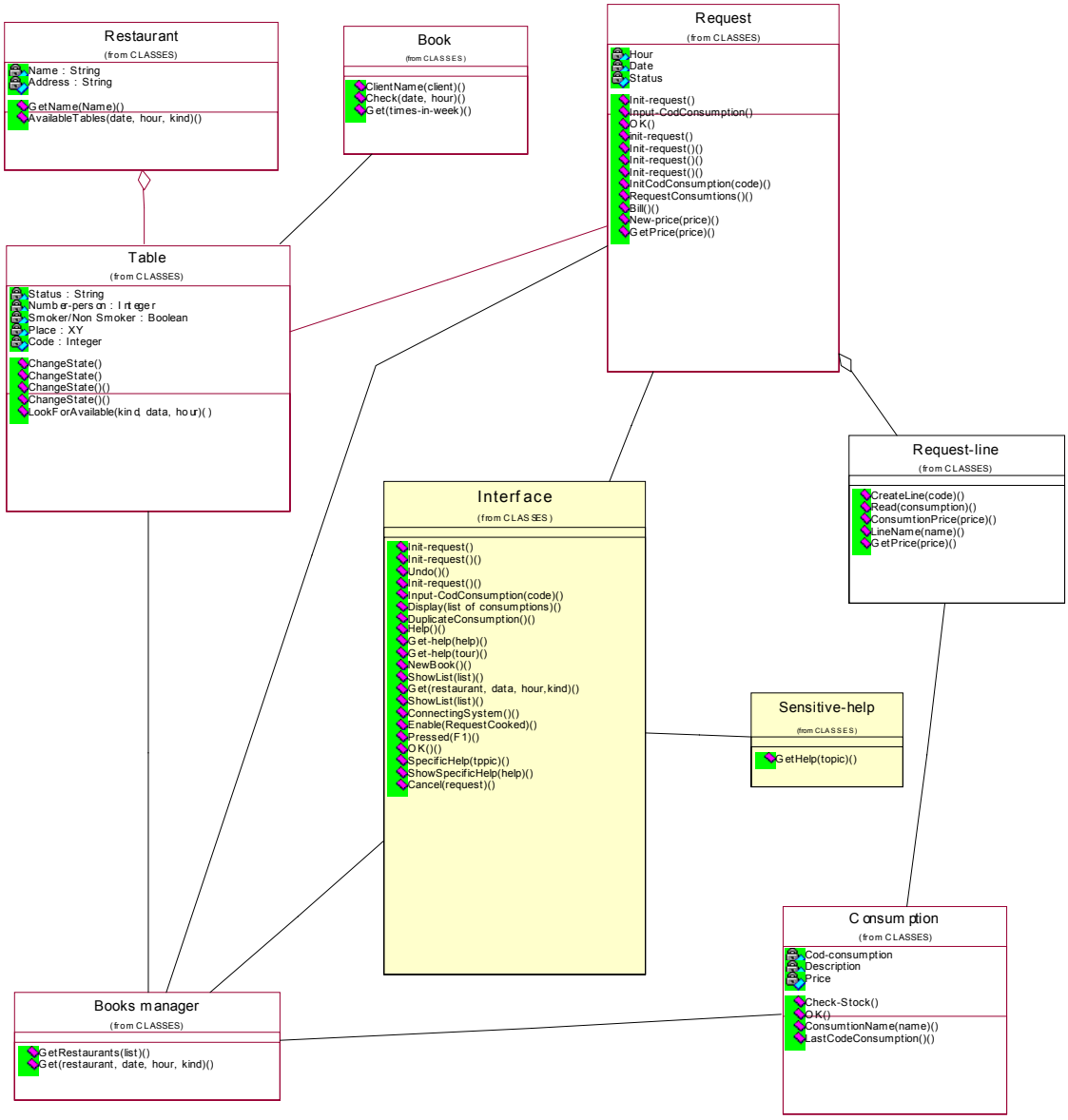


Figure F.31 Class diagram for restaurant management with Context Sensitive Help mechanism

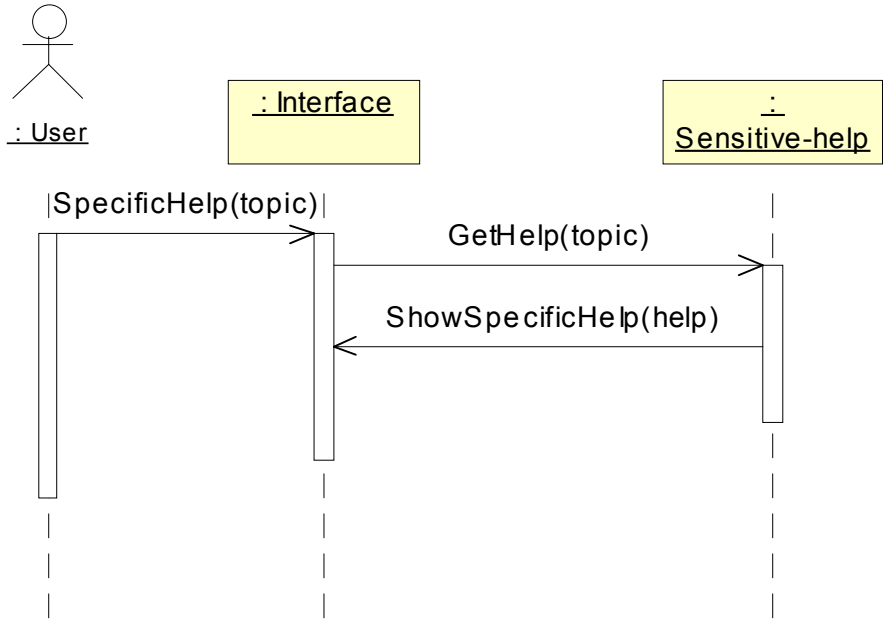
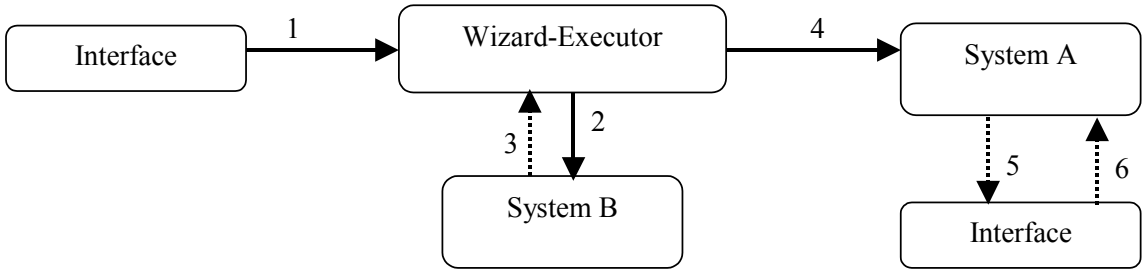


Figure F.32 Sequence diagram for restaurant management with Context Sensitive Help mechanism

F.16 Wizard Architectural Usability Pattern

- **Pattern Name:** Wizard
- **Usability Mechanism**: The wizard pattern presents users with a structured sequence of steps to carry out an operation, which it guides them through one by one. The task as a whole is separated into a series of more manageable subtasks. The user can go back and change earlier steps in the process at any time.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends the functionality to be assisted (1) to Wizard-executor. Additionally, for every step in wizard execution for which the user needs to enter information or make a decision, System A sends this notification to the interface through (5). Once the interface has the required information, it sends it to System A through (6).
    - Wizard-executor: it receives the request to execute a given wizard (1) from the interface. The information related to the wizard can be stored in the Wizard-executor or another system component. If Wizard-executor does not store the different steps of the wizard internally, it consults System B through (2), and, receives the information on the function to be executed to perform the different steps of the wizard from System B through (3). For each step to be taken, Wizard-executor asks the System to execute the functionality associated with each step through (4).
    - SystemA: it represents the part of the system that executes each step of the wizard. It receives the different functions to be executed from the Wizard-executor through (4) and, if user intervention is required, System A will inform the interface through (5) and will receive the information entered by the user through the interface by means of (6).
    - SystemB: This module is optional and will only be necessary if the Wizard-executor does not store the steps for each wizard that can be executed in the system internally. It receives the request for the name of the next step in wizard execution from the wizard-executor (2) and returns the information on the name of the function to be executed through (3).
- **Usability benefits:** The wizard helps with guidance, showing the user what each consecutive step in the process is.
- **Usability rationale:** The task sequence informs the user at once which steps will need to be taken and where the user currently is. The *learnability* and *memorability* of the task are improved, but it may have a negative impact on the *efficiency* of users forced to follow the sequence.

- **Consequences:**
  - Wizard execution might affect system *performance* of specific tasks.
  - Storing the functions to be executed outside the Wizard may reduce system performance but, in exchange, improve system modularity and the encapsulation principle.
- **Related patterns:**
- **Pattern implementation in OO:** This pattern will generate a "wizard-manager" class, which is responsible for knowing what wizard it has to execute depending on the request received from the interface. Additionally, the "wizard" class is responsible for ascertaining one by one the different steps to be taken using the "wizard-system-function" for this purpose, which will have the name of the system function to be executed and the order number by means of which it can identify the order of the different wizard steps. In this case, the Wizard-manager and wizard correspond to the Wizard-executor module, and the wizard-system-function class represents the System module, as it stores the information outside the Wizard-executor module. Additionally, the Waiter-device, represents the Interface module.
- **Example:** the waiter creates a rapid access for the functionality "Create new order" by pressing F2.

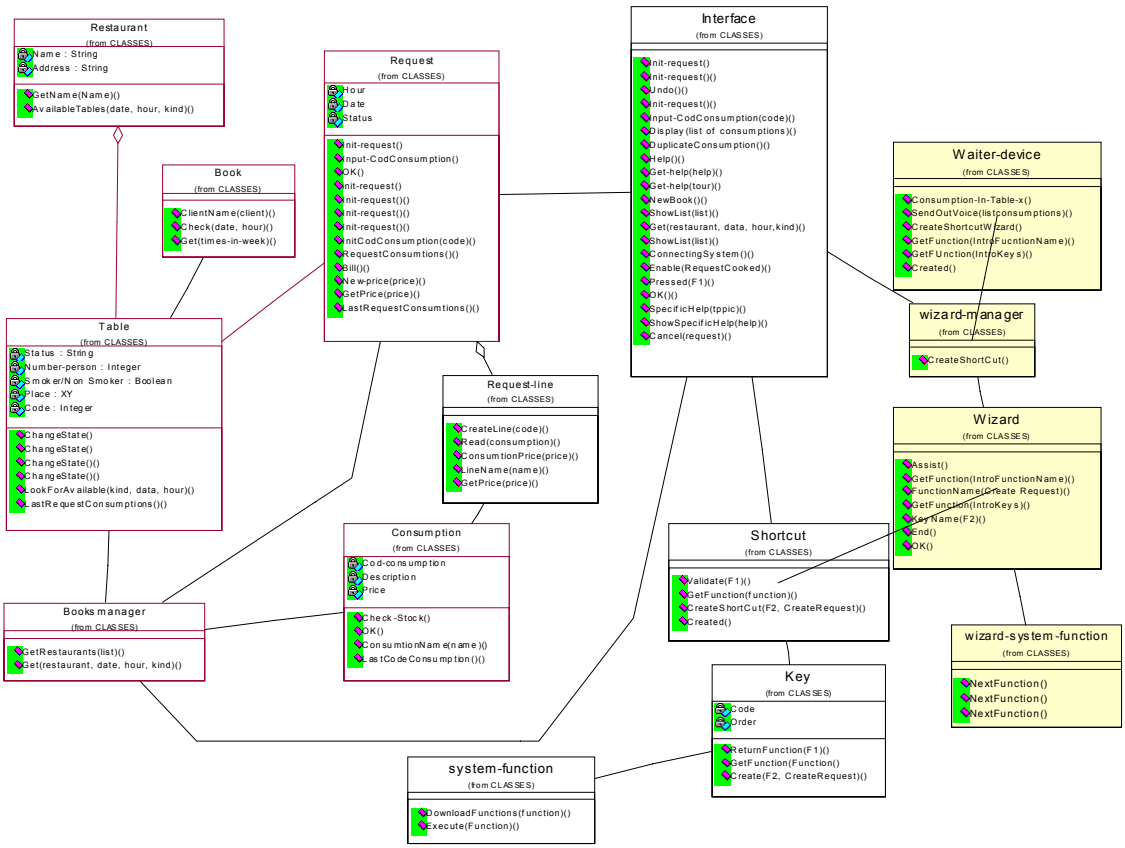


Figure F.33 Class diagram for restaurant management with Wizard mechanism

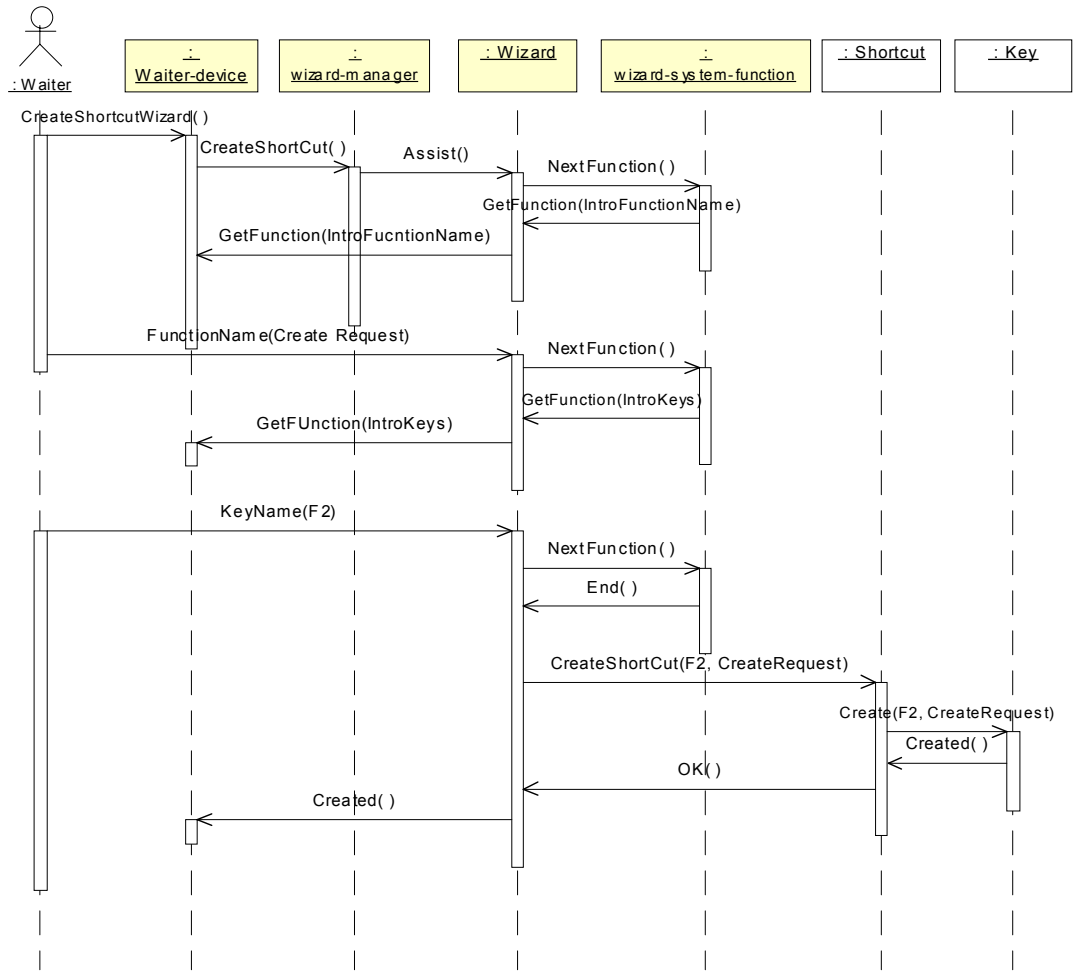
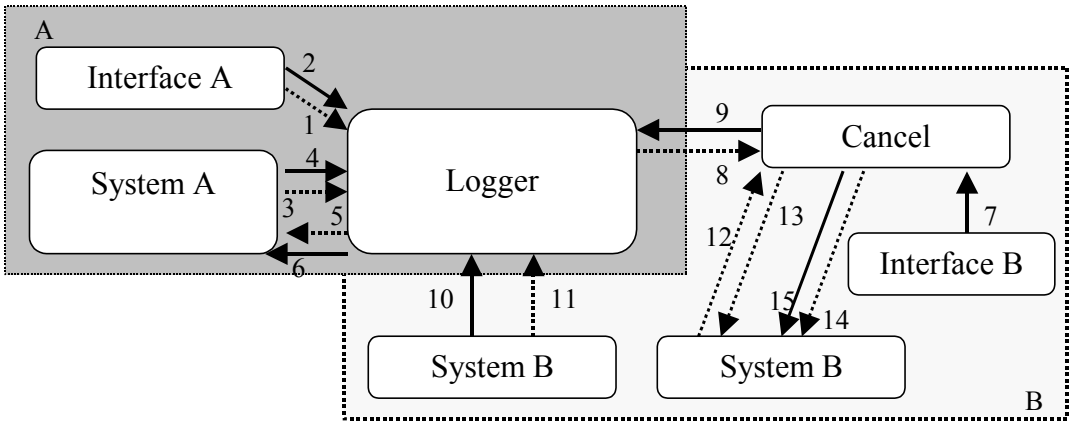


Figure F.34 Sequence diagram for restaurant management with Wizard mechanism

F.17 Cancel Architectural Usability Pattern

- **Pattern Name:** Cancel.
- **Usability Mechanism**: Users should be allowed to cancel a command that has been issued if they realise that they have done the wrong thing before an error state is reached. This is different from being able to undo an action after it has finished to return to the previous state.
- **Solution:**
  - Diagram:



  - Participants:
    - InterfaceA: it receives the request to execute an operation in the system, which may contain both the operation and the data (1) (2). As we will see later, this execution request can also come from the system (3) (4).
    - SystemA: this module sends the functions and data to be executed in the system (3) (4) to the logger, and also, optionally, if the logger does not store the logged actions internally, sends the information to the part of the system that manages these actions (5) (6).
    - Logger: this module receives the actions and the data requested by the user or from another part of the system (1) (2) (3) (4) and stores the logged actions and data either internally or in another part of the system, in which case it will have to send this action and the data to be processed by the respective part of the system to the system (5) (6). Logger receives the cancel request (9) from Canceler, then, if the logged actions are stored internally, it sends them one by one to Canceler in (8), provided that the all the operations stored by the logger have been performed. If the operations have only be stored but not executed, then nothing is sent in (8) and if they are stored externally, all the logger will have to do is receive them in (10) and (11) and delete them. If they are not stored internally, it will receive both the data and the operation to be cancelled from another part of the system, which we have called System B, by means of (11) and (10), respectively.
    - Interface B: it receives the cancel request and sends it to Canceler in (7). Additionally, it will search the system for both the action performed and the data associated with this operation (10) (11), provided that the logger does not store the data internally (esto lo hace el sistema B?).
    - System B: it searches the system for both the action performed and the data associated with this operation (10) (11), unless the logger stores the data internally. It receives the actions to be undone (or cancelled?). It receives the actions to be undone (or cancelled?) (13) and provides the opposite action (12)

(for which purpose, it will have to store the opposite for each action, see implementation section for example). The opposite action and the respective data will be sent to the respective part of the system (15) and (14) for execution.

- Canceler: it sends the cancel request (9) to logger and also sends each of the actions to be undone (or cancelled?) that it receives from logger to System B (13) and receives the opposite operation to the one performed (13 (12)) from system B. When it knows what opposite operation to be performed is, it sends it to System B along with the data associated with this operation through (14) and (15). Alternatively, if all the operations are stored in the system and performed together when the user presses accept, then Canceler will simply read through (10) and (11) and delete the accumulated operations, in which case (14) and (15) will not be used at all.

- **Usability benefits:** Being able to cancel commands helps with error management, as if users realise that they have done the wrong thing then they can interrupt and cancel an action before the error state is reached. It also gives users the feeling that they are in control of the interaction.
- **Usability rationale:** This pattern improves *reliability*, as it prevents users from making errors, at the same time as it improves user *efficiency*, enabling them to go back when they have taken an incorrect action.
- **Consequences:**
- **Related patterns:** History Logging, Undo.
- **Pattern implementation in OO:** This pattern generates a Canceler class responsible for triggering the whole cancel process. Additionally, the listener and action-done classes appear, which are used to store the actions that are either performed as the system operates or are stored and then all executed together. The “system-action” class is used to establish what the opposite is for any action that can be cancelled and has been executed by the system.
- **Example:** The waiter can cancel an order even if it has not be sent to the kitchen.

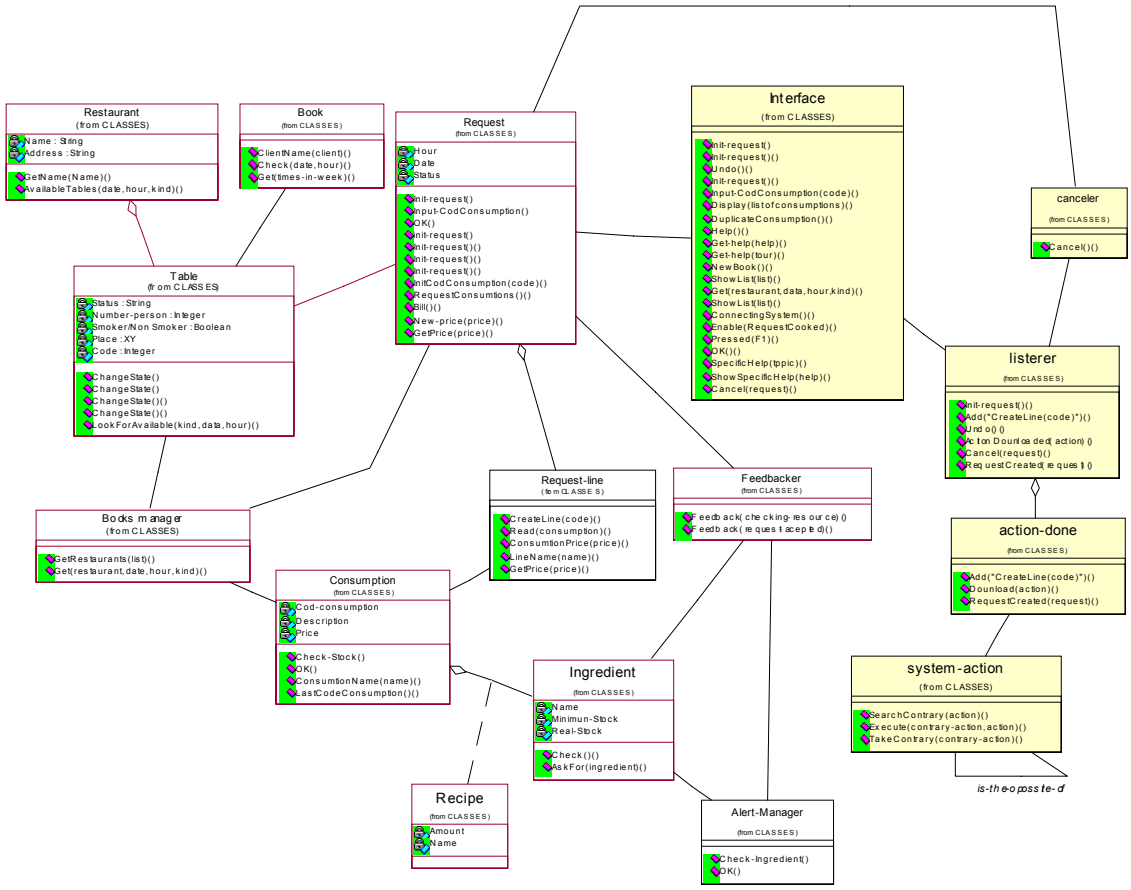


Figure F.35 Class diagram for restaurant management with Cancel mechanism

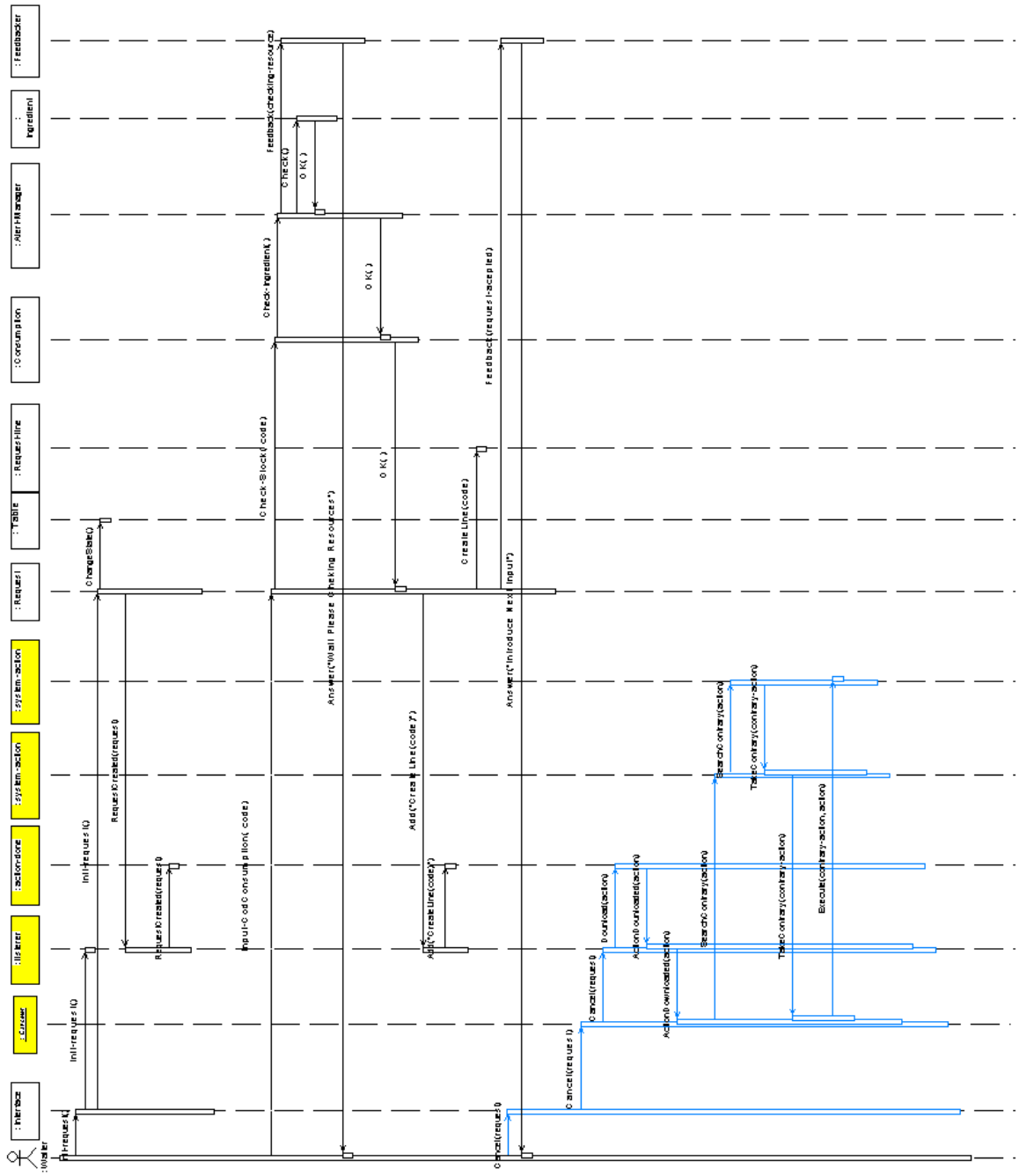
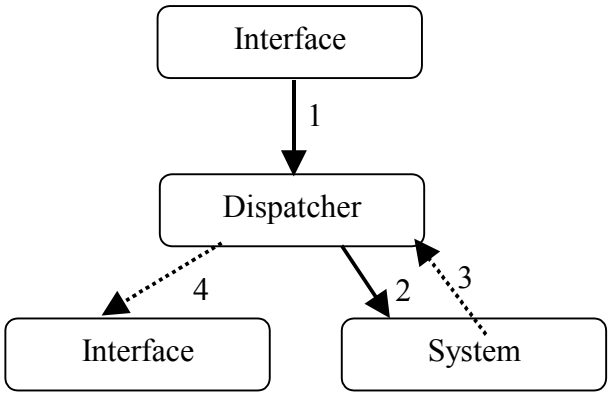


Figure F.36 Sequence diagram for restaurant management with Cancel mechanism

F.18 Multi-tasking Architectural Usability Pattern

- **Pattern Name:** Multi-tasking
- **Usability Mechanism :** Multi-tasking describes the situation where the system (and the user) can manage several tasks at the same time, allowing switching from one task to another as is most conducive to efficiently and effectively doing the work..
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends the function to be executed to dispatcher in (1). Additionally, if the user is to be informed of anything that is happening, he receives information from the dispatcher in (5).
    - Dispatcher: this component knows what resources are needed for each function that it has to execute in the system. It receives the function to be executed from the interface in (1). It sends the function to be executed to the system component in question in (2) after having checked that all the resources required to execute this function exist. Additionally, it receives the result of performing this operation from the system in (3). This result may specify either error or OK if everything went according to plan. If user has to be informed of the result of the operation performed, it sends this information to the interface (4).
    - System: this component refers to the part of the system responsible for executing the function specified by dispatcher in (3).
- **Usability benefits:** Providing a multi-tasking environment gives users the feeling that they are in control of the system, as at any point they can switch to the task that is of most interest to them.
- **Usability rationale:** This pattern might improve *efficiency* in system use by expert users, but it might also provoke more mistakes, thereby having a negative impact on *reliability*.
- **Consequences:**
  - System *performance* might be negatively affected, as resources have to be shared by different actions or applications.
- **Related patterns:**
- **Pattern implementation in OO:** The Dispatcher component generates a “dispatcher“ class responsible for distributing the different user requests depending on the resources available at the time.

- **Example:** the clock advises that any reservations made should be cancelled 20 minutes after reservation time if the diners have not arrived.

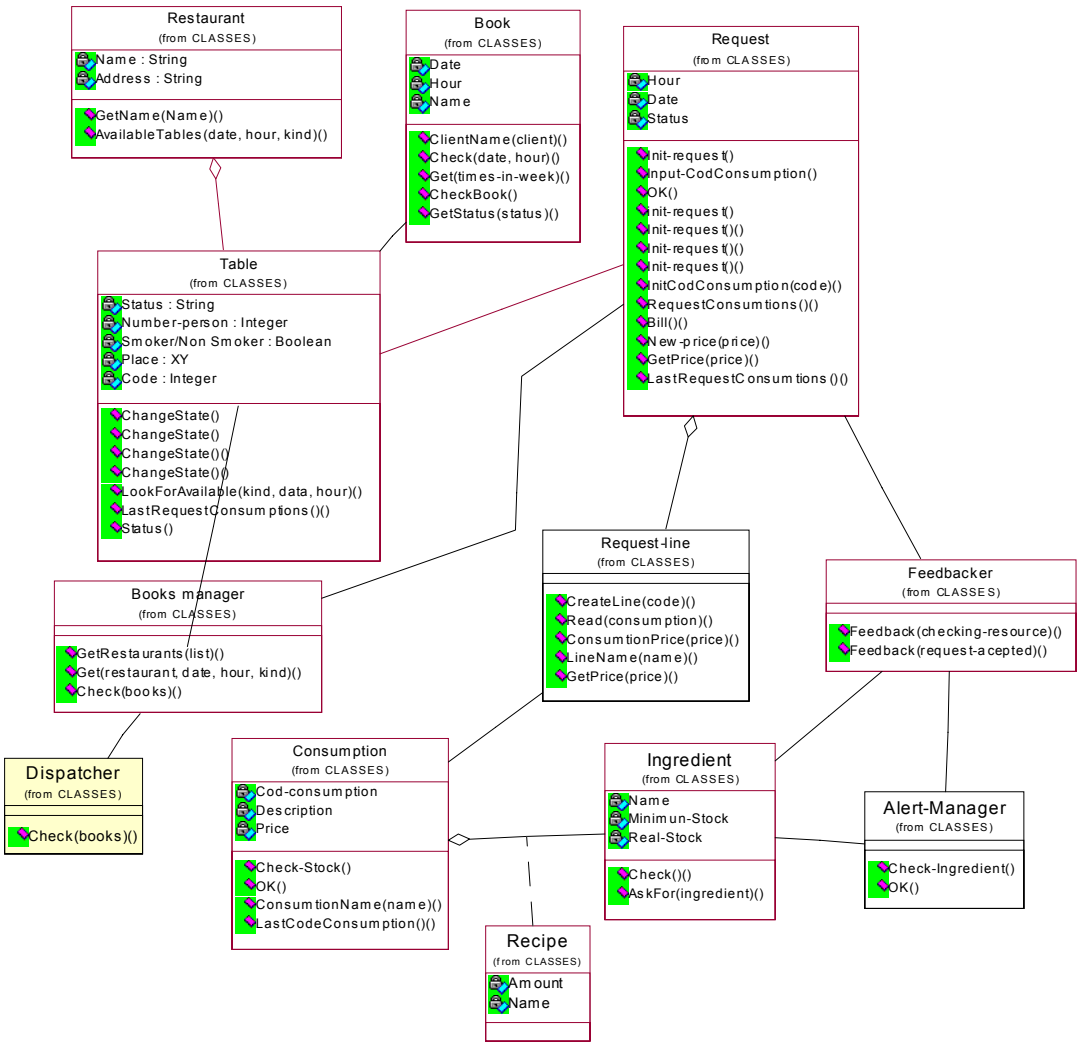


Figure F.37 Class diagram for restaurant management with Multi-tasking mechanism

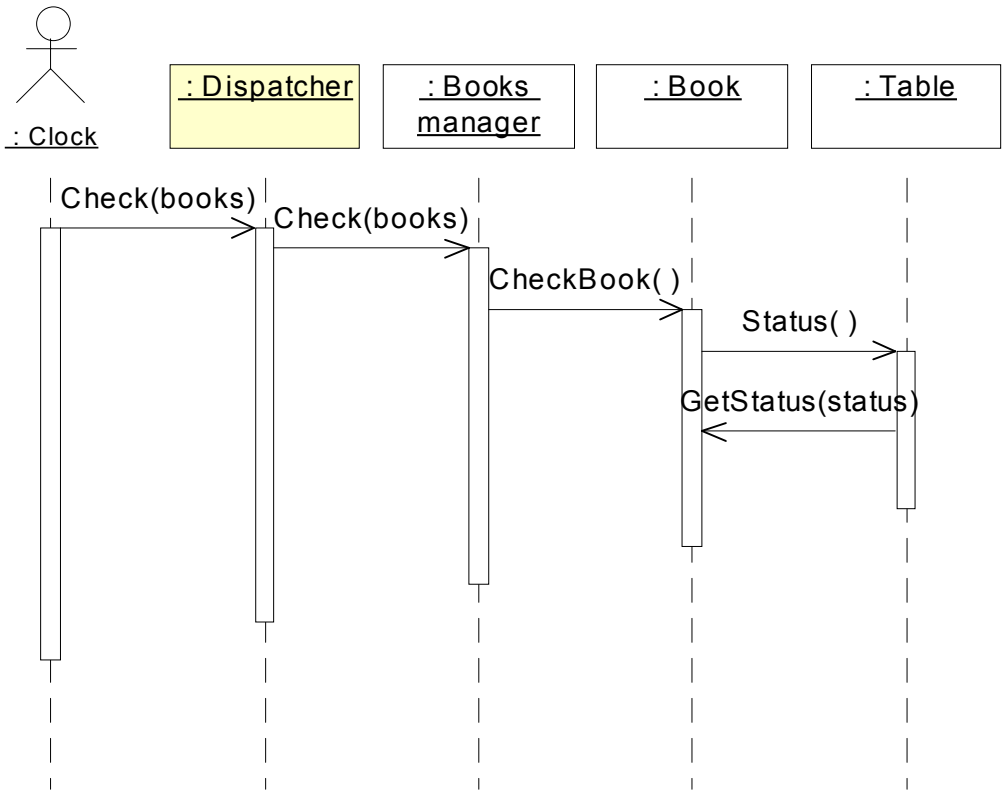


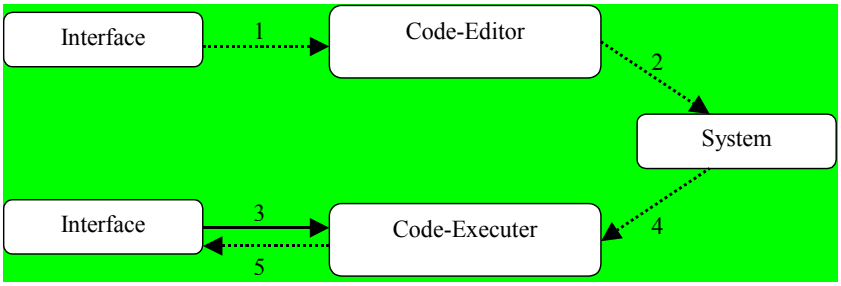
Figure F.38 Sequence diagram for restaurant management with Multi-tasking mechanism

F.19 Command Aggregation Architectural Usability Pattern

- **Pattern Name:** Command Aggregation
- **Usability Mechanism:** The system should provide the capability to allow users to perform different actions by means of a single command. Macro creation would be an example of this pattern.
- **Solution:**

STEP 3. Abstraction of the design solution for Command Aggregation

- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends a data set (1) corresponding to a given command, as well as the program code associated with the command to be created, to Code-Editor. Additionally, if a command is to be executed, it sends the name of the previously created command (3) to the Code-executer. When the command has been executed, it will receive the result of the command or error, if it cannot be executed, (5) from Code-Executer.
    - Code-Editor: it receives the name of the command to be created and the program code to be associated with the command (1) from the interface, which it sends to system for storage (2).
    - Code-Executer: it receives a previously created command (3) from the interface. It asks the system for the program code to be executed (4) and executes this code. Also it sends the result of executing this command to the interface through (5).
    - System: it receives the name of the command as well as the associated program code (3). It also sends the program code associated with a command to the Code-executer when it is requested for execution (4).
- **Usability benefits:** Providing the ability to group a set of commands into one higher level command reduces the users' cognitive load, as they do not need to remember how to execute the individual steps of the process once they have created a macro, they just need to remember how to trigger the command.
- **Usability rationale:** This pattern improves user *efficiency* and prevents any errors that may be made during the individual actions grouped in the aggregate command, which means that it improves *reliability*. On the other hand, it can have a negative impact on long-term *learnability*, that is, on user *memorability*.
- **Consequences:**
  - Increases system *performance* in executing the aggregated commands.

- **Related patterns:** The wizard pattern should be used combined with commands aggregation to facilitate and guide the process to create or execute a command.
- **Pattern implementation in OO:** In this case, the class editor includes the name of the command to be created and the code lines of the created program. Code lines must be stored to be reloaded in the future in the class command.

Example: the system must have the capability to create macros, for instance, to create a macro that would permit a maître d'hôtel to change the permitted period for arrival at the restaurant before the booking time, taking into account the number of bookings for each day.

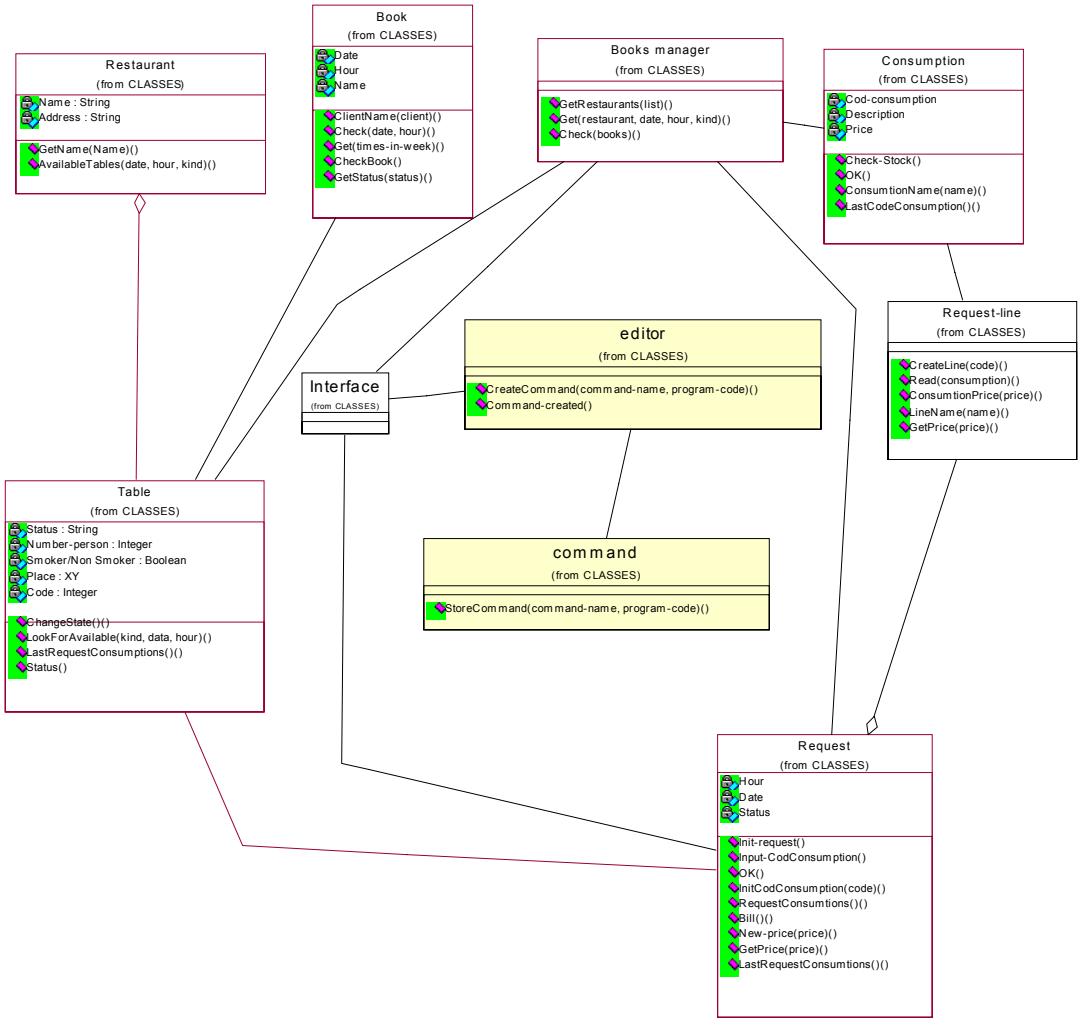


Figure F.39 Class diagram for the first application with Command Aggregation mechanism

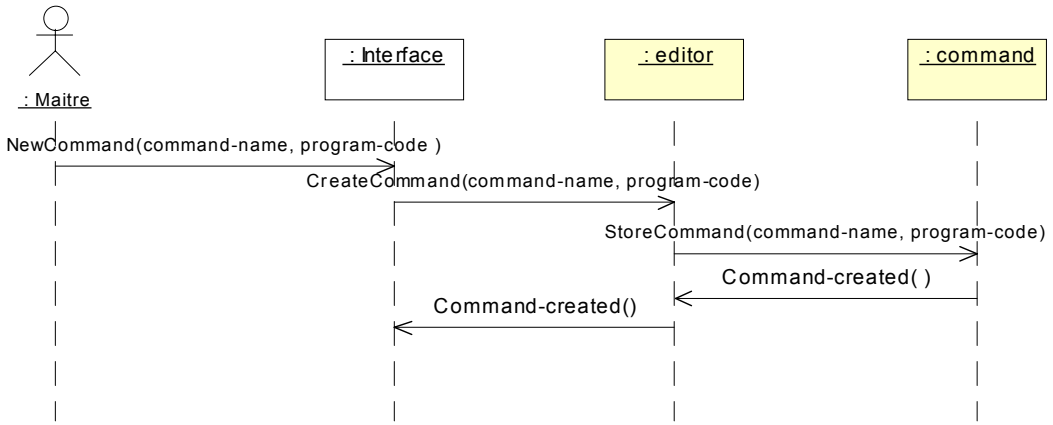
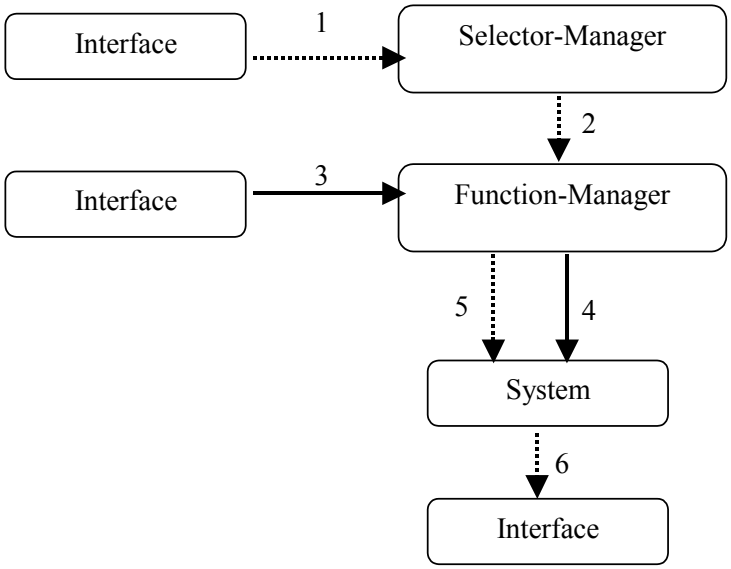


Figure F.40 Sequence diagram for the first application with Command Aggregation mechanism

F.20 Actions for Multiple Objects Architectural Usability Pattern

- **Pattern Name:** Actions for Multiple Objects.
- **Usability Mechanisms:** The same action often needs to be applied to a number of different objects. Providing the user with the possibility of grouping the objects and applying one action to them all "in parallel" will be of help in completing such a task more quickly and accurately. Errors are more likely to be made if each object has to be dealt with separately.
- **Solution:**
  - Diagram:



  - Participants:
    - Interface: it sends the set of objects selected by the user from the interface to Selector-manager in (1). Additionally, it sends the function to be executed in (3). If, after the requested operation has been executed, the user is to be informed of the result of the operation, the respective data are sent to the interface in (6).
    - Selector-manager: this component receives the set of elements on which to operate in (1). Additionally, it sends the set of objects on which the system is to operate to function-manager in (2).
    - Function-manager: it receives the operation to be executed in (3) and receives the set of objects on which the system is to operate in (2).
    - System: it receives the function to be executed (4) and the list of objects on which the specified function is to be executed in (5). Additionally, it sends the result of the executed function to the interface in (6).
- **Usability benefits:** Providing the ability to perform the same action on a number of objects at once reduces the time that it will take the user to complete a task, as the system should be much faster in repeating actions than the human user. The number of clicks (or equivalent actions) that the user has to make to complete the task is reduced.
- **Usability rationale:** This pattern improves user *efficiency* because users do not have to repeat the same action several times on different objects, and it also improves *reliability* through error prevention.

- **Consequences:**
- **Related patterns:**
- **Pattern implementation in OO:** This pattern generates a "selector-manager" class, responsible for receiving any user requests referring to a set of resources, and a "function-manager" class, which is responsible for collecting the list of elements to which the specified function is to be applied and requests that this function be applied to each element in the list.
- **Example:** the cook selects several ingredients and requests restocking.

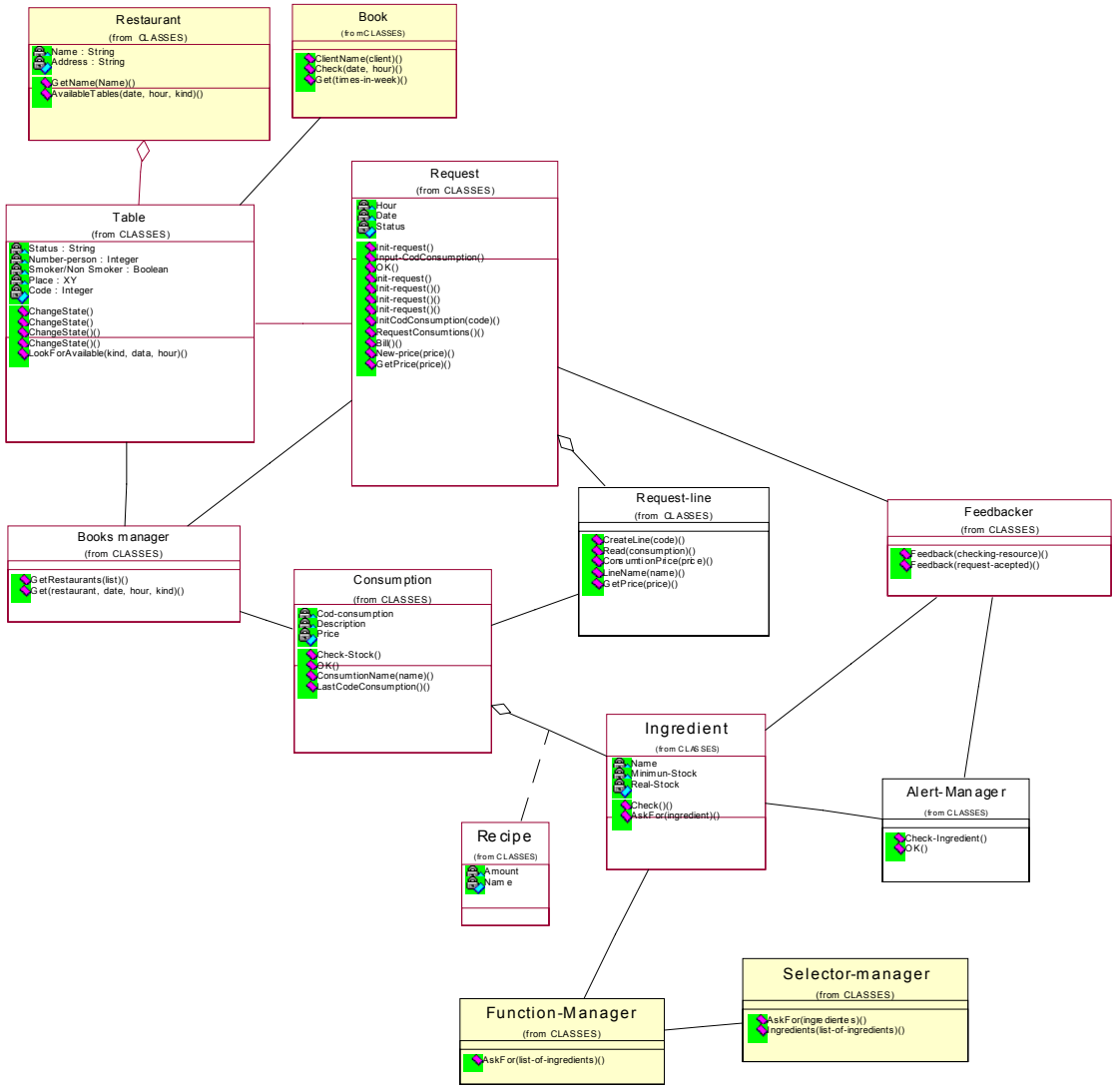


Figure F.39 Class diagram for restaurant management with Actions for Multiple Objects mechanism

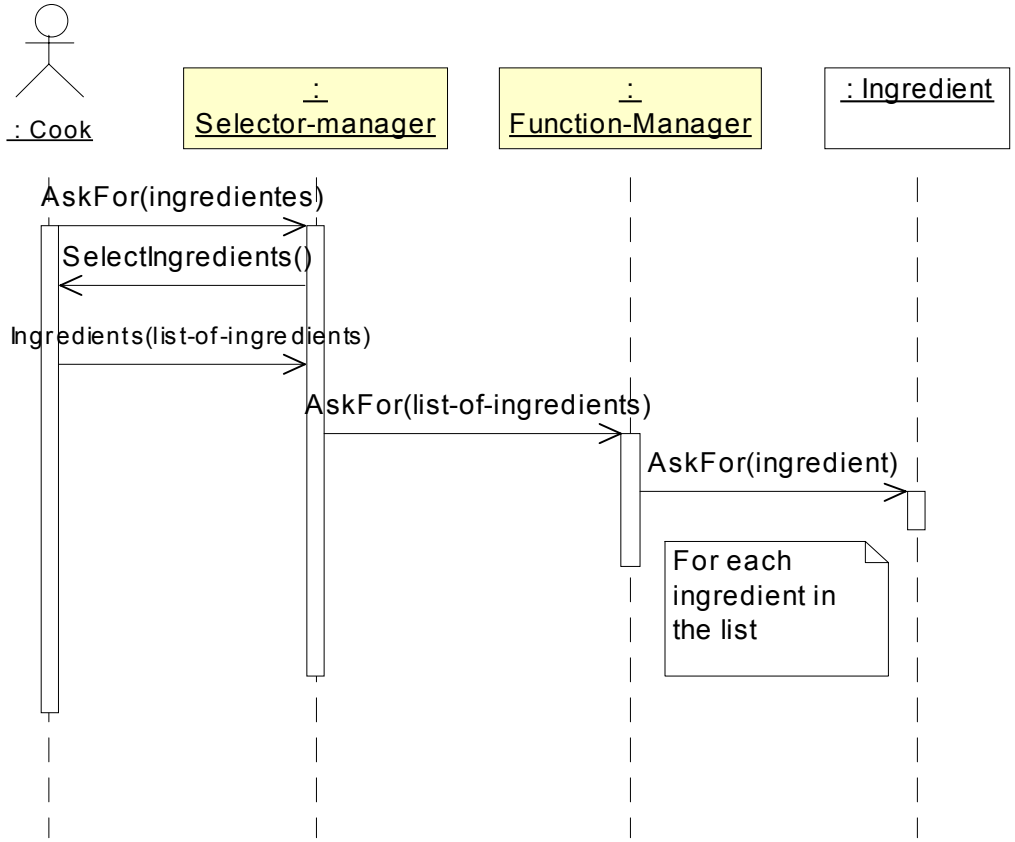


Figure F.40 Sequence diagram for restaurant management with Actions for Multiple Objects mechanism