In-Flight Entertainment System using BOOD

Premek Brada

28 September, 1995

This dissertation is a part requirement
for the MSc in Advanced Software Engineering

Abstract

This dissertation is concerned with the design of an In-Flight Entertainment System using Booch s Object Oriented Design methodology. The thesis project work formed a base for a subsequent evaluation of the level of software component reuse from an object-based design, and resulted in suggestions regarding a transition towards a full object oriented software development. It supports the claims that software development benefits from the use of object oriented methods.

The project was done with EASAMS Ltd. in order to provide help in the transformation of their software development process with the intention to eventually use the methodology developed by Grady Booch. Towards this goal, the company needed to gain knowledge in how to use this methodology and to investigate which components of their current object--based design can be reused in the Booch-based versions of their applications. As an additional issue, the use of CASE tools in the design process needed to be evaluated.

For the thesis project purposes a design of the passenger part of the In-Flight System was developed using Booch s methodology. This design was taken as a basis for the component reuse evaluation, and its development provided an opportunity for practical assessment of the Rational Rose and Object Domain CASE tools.

The results of this work are presented in this dissertation. It begins with a theoretical background in the form of Booch s methodology and CASE tools descriptions. The In-Flight Entertainment System application requirements are summarised in one chapter and the Booch-based design of the Passenger Application with the supporting appendices forms the main part of the dissertation. The project results are presented in the chapters discussing the component reuse and CASE tools evaluation, and synthesised in the form of recommendations regarding the methodology transition. The dissertation is concluded with a discussion of its achievements.

Note:

This is the postscript version of the dissertation with some fonts and styles changed to facilitate on-line viewing. The layout of the text has therefore changed and does not necessarily correspond to the paper version; the author disclaims any responsibility for incorrect page references caused by this. Contact the author at <brada@kiv.zcu.cz> to obtain the paper version, and Microsoft to include a sensible PostScript driver in Windows 3.1.

Acknowledgements

There are a number of people who made my studies in Sheffield possible and enjoyable. I am particularly indebted to the Excalibur Scholarship people for their idea and for making it work, and to GEC-Marconi for their sponsorship; as well as to my parents and friends at home and in Sheffield for their support.

To the LORD my God then I give thanks and praise: it was Him who called me to this studies and who has been my everpresent source of strength and joy.

Table of Contents

Chapter 1: Introduction 8

1.1 Project Goals 8
1.2 Overview of the IFES 8
1.3 Summary of Project Work 9
1.4 Encountered Problems 9
1.5 Document outline 10
1.6 Abbreviations 10
1.7 Terms 10

Chapter 2: BOOD Methodology 12

2.1 Introduction 12
2.2 Overview of BOOD 12
2.3 Analysis and Design Techniques 13
2.4 Micro Process 14
2.5 Macro Process 16

Chapter 3: CASE Tools 20

3.1 Introduction 20
3.2 Types of CASE tools 20
3.3 Role of Repository in I-CASE 21
3.4 Problems with Tools 22
3.5 Requirements on I-CASE Tools 22
3.6 Summary 23

Chapter 4: CASE Tools Evaluation 24

4.1 Introduction 24
4.2 Evaluation Criteria 24
4.3 Evaluation of Rational Rose/C++ v2.5 26
4.4 Evaluation of Object Domain v1.02 27
4.5 Application of CASE for EASAMS software development 29
4.6 Summary 29

Chapter 5: IFES Requirements 30

5.1 Introduction 30
5.2 Requirements Summary 30
5.3 Other System Characteristics 32
5.4 User Interface 32

Chapter 6: IFES Design 34

6.1 Introduction 34
6.2 IFES Analysis 34
6.3 Architectural Design 37
6.4 Detailed Design 40

6.5 Summary 50

Chapter 7: Reuse Evaluation 51

7.1 Introduction 51
7.2 Scope 51
7.3 Reuse Analysis Method 52
7.4 Examples of Reuse Tables 52
7.5 Conclusions 53

Chapter 8: Transition to BOOD 55

8.1 Introduction 55
8.2 Current Situation 55
8.3 Readily Applicable BOOD Elements 57
8.4 Methodology Transition 58
8.5 Summary 59

Chapter 9: Conclusion 61

List of References 62

Project Diary 64

Appendix A: Analysis Scenarios 65

A.1 Scanning Video Channels 65
A.2 Switch between Audio and Video 66
A.3 Payment for a Selected Game 66
A.4 Browsing Service Menus 67
A.5 Playing the Selected Video Programme 68

Appendix B: Analysis Data Dictionary 69

Appendix C: Design Scenarios 73

C.1 Playing a Video Programme 73
C.2 Switching between Video and Audio 73
C.3 Activating the Context Help 74
C.4 Going to a Previous Menu Level 75

Appendix D: Level 2 Class Specifications 76

D.1 PassApp 76
D.2 Item Collections 77
D.3 Browsers 78
D.4 ServiceItems 79
D.5 MenuSystem 81
D.6 Database 82
D.7 Devices 84

Tables of Figures

Figure 1: Macro and Micro Process 13
Figure 2: Schematic Screen Example Menu 33
Figure 3: Example Analysis Scenario 36
Figure 4: Application Layers 38
Figure 5: Module Hierarchy 38
Figure 6: Item Execution 41
Figure 7: Passenger Application Class Framework 42
Figure 8: List of Data Types 44
Figure 9: Class category PassApp 45
Figure 10: Class category ServiceItems 46
Figure 11: Class category Browsers 47
Figure 12: Class category ItemCollections 47
Figure 13: Class category MenuSystem 48
Figure 14: Architecture of the Current Design 57

Scenario 1: Scanning Video Channels 65
Scenario 2: Switch between Audio and Video Programmes 66
Scenario 3: Payment for the Selected Game 66
Scenario 4: Browsing Service Menus 67
Scenario 5: Playing the Selected Video Programme 68
Scenario 6: Playing a Video Programme 73
Scenario 7: Switching between Video and Audio Programmes 74
Scenario 8: Activating the Context Help 75
Scenario 9: Going to a Previous Menu Level 75

THIS SPACE IS INTENTIONALLY LEFT BLANK

PLAGIARISM DECLARATION HERE

Chapter 1: Introduction

This master s thesis is based on a project undertaken for EASAMS Ltd. to investigate the possibilities of a transition from an object-based to a fully object-oriented software design process. Among the issues this transition brings are the adoption of the new methodology, the learning curve of the tools that are introduced, and reuse of the already available design and code.

These issues were recognised by EASAMS Ltd. who have a long term plan to move towards the use of the object-oriented design methodology developed by Grady Booch. This plan coincided with the current software development of an In-Flight Entertainment System (IFES), a project with long duration and several scheduled software releases (different airlines, different application versions).

1.1 Project Goals

Based on this observations, the project had three main goals: to develop an example design using Booch s methodology to gather experience in using it; to indicate the opportunities for reuse from the currently developed design into the Booch-based one; and to evaluate the suitability of CASE tools for use in this design process.

The example design was necessary in order to get a practical knowledge of the methodology and its techniques, and in order to facilitate work towards the other goals. There is only a general knowledge of object oriented programming among the company staff and the know-how and results of this project can be used as an aid during the transition period.

Producing the design of at least a part of the system also helps to assess the possible reuse of code from the current development. Reuse is an important factor in the In-Flight Entertainment System project because it deals with relatively similar applications for the various airlines. Also, the cross-methodology reuse would make the transition much easier.

Lastly, modern software development benefits from the use of tools that automate the various mundane tasks and assist the creative but controlled design process. Acquiring the tools however requires knowledge of both what level of support they can offer and of the real needs of their user. The third goal of this thesis was to evaluate two tools from this point of view.

1.2 Overview of the IFES

The In-Flight Entertainment System is used by airlines to provide entertainment for passengers during the flight. It provides capabilities such as listening to audio programmes, watching films and television channels, using telephones, and purchasing duty-free and catalogue goods.

The system installed on the aircraft has two main parts: the passenger component used by the airline customers to access the services through a seat-back screen and handset, and the flight-attendant component used by the cabin crew to change the system settings and deal with the transactions and data processing. There is also a ground-based system which allows

preparation of the system data. The EASAMS Ltd. team develops the software part of the passenger and flight-attendant systems.

Currently, the company still uses a traditional object-based approach to the software design. There are two main reasons for this: firstly, the time for the application development is very limited (9 months for a fully-functional system of approx. 85 KLOC) which prevents the use of any new methodology with the associated learning curve effects. Secondly, there is a concern that the object oriented implementation would have unacceptably high run-time overheads on the hardware platform used for the system.

1.3 Summary of Project Work

The work on this master s thesis project was done in six phases that correspond to its goals. Although they to a certain extent overlapped, the sequence of these phases was as follows:-

1. As a prerequisite, a general knowledge of the In-Flight system was necessary, as well as an insight into the current IFES development. This knowledge was gained by reading the requirements specifications and associated documents, and by participating in some aspects of the team s work e.g. design reviews.

2. Another preliminary task was to become acquainted with Booch s methodology. The main reference [Boo94] was used to gain an understanding of its principles, and a small-scale pilot project was run to get an initial insight into its practical execution. Rumbaugh s book [Rum91] provided some additional helpful ideas.

3. Developing the example BOOD design formed the major part of the project work. It comprised analysing the system requirements and building the design of its representative part, using the CASE tools chosen for evaluation. Booch s book was constantly used during this work to follow the methodology as closely as possible. Three internal EASAMS Ltd. documents were produced documenting the analysis and design techniques and decisions, to be used as a reference for the transition period.

4. The direct outcome from the work was the experience with the two CASE tools, Rational Rose and Object Domain. Evaluation criteria lists were produced based on literature and the practical tool use, and the assessment as presented in this dissertation was performed.

5. The finished example design made it possible to evaluate the code reuse. This was done using the available design documentation of the current development process and resulted in an internal document produced for EASAMS Ltd. A summary of its results is also included in this dissertation.

6. All the previous steps formed a basis for understanding the issues brought by the transition from the current to an object-oriented design process. As a result, several suggestions were made regarding the use of BOOD techniques and CASE tools as well as concerning the general management of the transition process.

1.4 Encountered Problems

The work on this project had to overcome problems of various kinds. The size of the project and the amount of work it required were underestimated at the beginning and as they were realised during its course, some adjustments had to be made (the original plan was to develop a Level 2 design for the whole Passenger Application).

There was some initial uncertainty at the company side about the possible project objectives. This introduced several changes in its goals, delaying the actual design work. The major problem however was the lack of feedback from both the company managers and the team members. Although this was understandable their extremely tight project schedule left

little time to deal with any other issues it certainly had an impact on the quality of the produced design and internal documentation.

1.5 Document outline

This dissertation document is divided into seven chapters that cover the various aspects of the work, forming three logical groups.

Chapters *BOOD Methodology* and *CASE Tools* form a theoretical background for the project, describing the design methodology and presenting basic information about the tools. The *CASE Tools Evaluation* chapter follows immediately so as not to break the cohesion of the rest of the dissertation.

The following two chapters, *IFES Requirements* and *IFES Design*, contain a summary of the In-Flight Entertainment System functionality and of the developed Passenger Application design. Their text is based on the documentation developed by and for EASAMS Ltd.

The evaluation chapters form the final part of the dissertation. *Reuse Evaluation* outlines the technique and results of the code reuse investigation. The recommendations regarding the *Transition to BOOD* are presented in the last chapter.

The dissertation has four appendices related to the IFES Design chapter. They contain diagrams, tables and code extracts documenting the Passenger Application BOOD design, separated from its main text to keep it concise and more readable.

The author is aware that the length of the dissertation exceeds the recommended 60 pages. However, it already contains a substantially shortened description of the thesis project work and further truncation could only be done at the expense of leaving out important parts of its results.

The abbreviations and terms used in the dissertation are defined in the tables below.

1.6 Abbreviations

AFS	application functional requirements specification
API	application programming interface
ARS	application requirements specification document
BOOD	Object-Oriented Design methodology as defined by G.Booch in [Boo94]
FA	flight attendant
FRS	functional requirements specification
GUI	graphical user interface
IFES	In-Flight Entertainment System
LCD	liquid crystal display
PVP	personal video player
QA	quality assurance
SPM	seat processor module
TPV	third-party vendor
TPVA	third-party vendor application

1.7 Terms

Application Database	the repository of all persistent data associated with the IFES software
attribute [of a class]	variable of a data type inside the class abstraction barrier

business rules	the operating procedures pertaining to a given airline
current design	design of the IFES applications developed by EASAMS Ltd.
event	in an event-driven system, an occurrence which triggers the application response, and the data structure used to convey the associated information
feature [of a class]	refers to both attributes and methods of a class
Level 1 design	stage which identifies application components, also Architectural design
Level 2 design	design stage which established the class and component interfaces
Level 3 design	design stage which expands implementation of classes
message	transfer of control between two objects by the means of the target object s method invocation
method [of a class]	an operation defined by the class, also called member function in C++

Chapter 2: BOOD Methodology

Summary of Booch s Object Oriented Design

2.1 Introduction

This chapter contains a brief description of the object oriented design methodology developed by Grady Booch (the abbreviation BOOD will be used from now on). It should help to understand its main concepts and procedures and, Because Booch s methodology was used for the IFES design, should be also helpful in reading the IFES Design chapter of this dissertation.

This summary is based on two sources: the original description of the methodology in [Boo94] and the practical experience gained during the project work. The character of the work placed emphasis on the design stages of the development, and the sections dealing with conceptualisation and maintenance stages are therefore shortened.

The chapter is organised as follows: an overview of the methodology is presented first, introducing the notions of micro- and macro-process. Next follows a brief description of domain and use-case analysis which are the main techniques used in BOOD. The micro process is then explained, followed by the description of the macro process stages.

The notation symbols used by BOOD are not included in this methodology summary for brevity. Examples of the main ones, i.e. class and scenario diagrams, can be found in chapter IFES Design and in the appendices of this dissertation.

2.2 Overview of BOOD

Booch s methodology is an object-oriented analysis and design methodology which supports the evolutionary incremental delivery approach to software development. The BOOD development process is described at two levels: *macro process* defines the overall process framework which corresponds to the traditional waterfall lifecycle model; *micro process* defines the sequence of steps within each stage of the macro process.

The relationship between these two processes is shown in Figure 1. The solid-line boxes represent the macro process stages and correspond to the states in the development process. As indicated three of them contain the micro process in one or several spins. The micro process steps outlined in the enlargement are related to the development of the system components i.e. classes, starting with their identification. Both processes are explained in more detail later in this chapter.

The main technique that is used throughout the BOOD process is *use-case analysis* which was first described by Jacobson. It is both an analysis and a design technique based on scenarios and is described below together with another important technique, *domain analysis*.

2.3 Analysis and Design Techniques

The analysis and design techniques are used to find the candidate classes and objects that will constitute the solution for the system, and to refine the design of the existing ones. To achieve a good degree of stability of system abstractions and a correct allocation of operations to objects, BOOD uses behaviour-driven techniques to derive the design from the structure and desired behaviour of the real system.

The goal of *domain analysis* is to establish those stable abstractions. The technique identifies the classes and objects that are common to all applications within a given domain [Boo94] and that are perceived as important by the domain experts (e.g. users of similar systems). Domain analysis is done by consultations with the experts as well by as analysing the existing systems within the domain.

The product of domain analysis is a list of key abstractions which form a generic model of the domain applications. The abstractions and their relationships are captured in class diagrams (see 2.4 Micro Process below for explanation and chapter IFES Design for

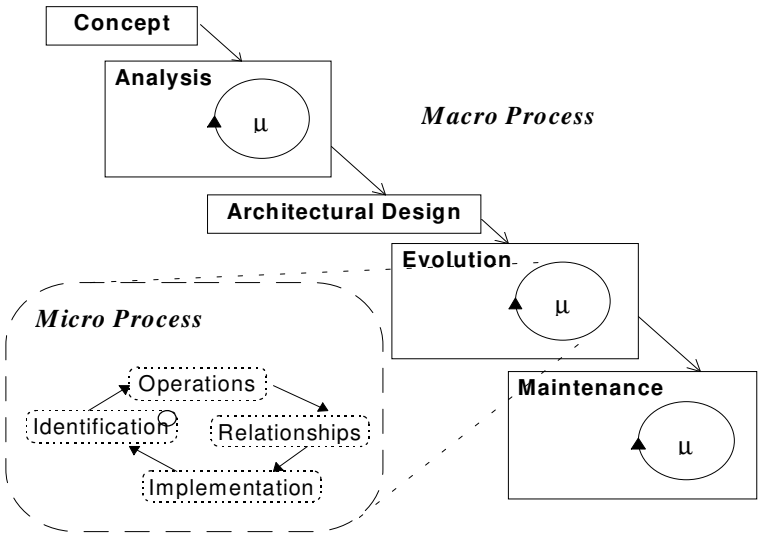


Figure 1: Macro and Micro Process

examples of these).

Use-case analysis is used to discover and clarify the responsibilities of system components, i.e. classes and objects. The idea is to explore the interactions of previously identified

abstractions during a particular example of system usage. This use-case of the system is usually initiated by a user input or a system event and can be described by a scenario of events that follow.

The analysts walk through these scenarios in order to identify the objects that participate in the scenario, the responsibilities of each object, and how those objects collaborate with other objects [Boo94]. Use-case analysis therefore leads to specification of classes and their methods at the level of detail relevant to the given functionality and design stage. The product are class specifications with clear boundaries, and object- or interaction diagrams describing the scenarios.

The scenario-driven development is a key feature of BOOD with the effect that the design of the internal parts of the system is based on its external behaviour. The main benefit is that the analysis model and the subsequent design of the software system map very closely to the real system and its environment, better meeting the user s needs.

2.4 Micro Process

The micro process describes the flow of activities in the everyday design process. This should be creative and therefore usually rather informal, and the micro- and macro process definitions attempt to put them into a controllable framework to achieve the defined level of software process maturity (described for example in the Capability Maturity Model [Pau93]).

During one spin of the micro process, the classes relevant to the functionality of the developed release are successively identified, specified and implemented. The degree of implementation detail depends on the current design level: skeleton functionality and free-form textual description at the beginning, precise definition towards the system completion. The next four subsections describe the micro process steps.

2.4.1 Identify Classes and Objects

The micro process starts by identifying the classes and other abstractions related to the functionality of the given analysis or design goal. Domain analysis and use-case analysis techniques are used to discover the relevant abstractions in the problem domain and invent new ones for the solution. Some of these abstractions may be already defined from the previous stages. The scenarios to be used in this step are identified in the corresponding stage of the macro process in order to focus the development to a particular area.

The product of this step is a list of classes added to the data dictionary of the developed system. The data dictionary is used in the following steps and stages as a centralised source of system component definitions. When CASE tools such as the Rational Rose are used, the data dictionary may have a form of class definitions held in the tool repository and accessible via the class diagrams.

2.4.2 Determine Class Methods

This step refines the semantics of the classes represented by a list of responsibilities in the analysis stage, and by methods which define the protocol of the class during detailed design. As more implementation detail is developed in subsequent iterations, the methods are given precise names and eventually defined by their full signatures (function headers in C/C++). An updated version of the data dictionary is a direct product of this process.

This step is very closely related to the previous step; in fact they are often performed at the same time. Class methods are identified during detailed scenario analysis, when the free-form textual description of messages passed between the objects is translated into a sequence of method invocations. The object and interaction diagrams which capture the semantics of

the scenarios are therefore another product of this stage. For examples of these diagrams see chapter IFES Design and the appendices.

As a complement to the scenario-driven class design, isolated class design is performed after the class roles have been established. The goal of this technique is to complete the list of class methods by adding operations that are not directly required for the system implementation but which belong to its semantics. The resulting set of class methods should follow the rules of a good class design as described in [Boo94], i.e. it should be complete, sufficient, and composed of primitive operations.

2.4.3 Identify Relationships of Classes

After the classes have been designed in isolation , they are placed in the client-supplier and inheritance hierarchies by considering their mutual relationships and structural or behavioural similarities. The aim is to achieve a clear separation of responsibilities between the abstractions.

During analysis the type of relationship is often unspecified, indicating only that the classes are related in some way. As the design gets more the relationships are refined into one of inheritance, containment or client-supplier types. These relationships have the following meaning:

- *Using* denotes client-supplier relationship, where operations of the client class invoke operations of the supplier class, or have signatures whose return class or arguments are instances of the supplier class [Boo94].
- *Containment* denotes whole/part relationship (aggregation), where instances or references to instances of the part classes are attributes of the whole . However, because the methods of the contained class are usually invoked by the whole, this relationship can be considered as a special case of the client-supplier relationship.
- *Inheritance* denotes a generalisation/specialisation relationship. The concrete type of inheritance is defined by the implementation language, namely the use of single/multiple inheritance, access to the private parts of the base class, etc.

The class relationships are captured in class diagrams. In addition, these diagrams are used to structure the design into layers that correspond to the logical partitions of the system. Both inheritance and use/containment can be shown in the same diagram as required. However, if a class has many containment-type relationships it is better to specify the parts directly as attributes to keep the diagrams readable. (An example of this case is the class `CPassApp` in the IFES Passenger Application BOOD design).

2.4.4 Implement Classes

This step is an integration of the previous three steps. Class specifications are completed by defining the attributes that are needed to support their implementation, and algorithms of their methods. All methods should be specified in detail as soon as possible, mainly concerning their signatures. Optimisation of selected operations is also done in this step if required during later design stages.

There are two products of this step, which are also the products of the whole micro process. Firstly, the system data dictionary (whether implemented as an explicit database, or indirectly through details of classes in the class diagrams of a CASE tool) is updated by the completed class specifications. After this, the source code as the second product can be generated from the class specifications.

Related to the previous and this step is the development of the physical system design. The compilation units, or modules, of the implementation are specified and the classes are

allocated to them. The main goal is to structure the implementation in such a way that will facilitate independent work of the development team members.

2.5 Macro Process

The BOOD macro process defines stages in software development in a manner similar to the waterfall model (see Figure 1 on page 13). Its aim is to provide a framework for management practices to achieve control over the process, mainly by enforcing development stage boundaries. This approach is necessary in an evolutionary development where these boundaries tend to be blurred.

Software process which uses BOOD goes through the following stages: *conceptualisation* develops the basic idea of the system functions, *analysis* produces its complete model, *architectural design* establishes the logical and physical application framework, and *evolution* gradually develops the implementation in successive releases; after delivery the system evolves in *maintenance* cycles. These stages are described in more detail in the following subsections.

2.5.1 Conceptualisation

The purpose of conceptualisation is to establish the basic requirements, or problem statement [Rum91], for the system to be developed. Before the development starts, the basic idea of what is to be done needs to be clarified and its assumptions validated.

The main activity and associated product is the development of one or several throw-away prototypes which are used to validate the ideas and to make decisions about the system development. Booch observes that conceptualization is an intensely creative activity and suggests that at this stage the best thing the development organization can do to facilitate the team s effort is to stay out of its way.

2.5.2 Analysis

The goal of analysis is to provide a complete description of the system to be developed by producing a model of its external behaviour. The description is based on the knowledge gained from the prototype and captured by use-case scenarios. It shows how the problem-domain abstractions cooperate to achieve the functionalities related to the system function points which Booch in [Boo94] defines as some primary activity of a system in response to some event.

The product of analysis is a requirements document that describes the behaviour of the system, plus associated management documents like a risk analysis. The functional requirements are mainly expressed using the scenario diagrams corresponding to function points. Also, an initial form of the data dictionary is prepared which lists all key abstractions and descriptions of their responsibilities (functions) in the system.

Steps and Techniques

The analysis stage comprises a reduced micro process with two steps, components identification and scenario planning, performed in several iterations. The stop condition for the iterations is a stable data dictionary (little changes in classes and their roles and operations).

Components Identification

The purpose of this task is to find the key components of the system which are the potential classes for the implementation. However, the goal is to find classes related to the whole

problem domain, not just to the particular system under development. This helps to identify more general classes with higher potential for reuse in related applications.

The components are identified using the domain analysis technique outlined above. The consultations with domain experts reveal a number of things tangible objects, physical locations, structural elements, concepts that are important within the problem domain. These are the key abstractions used later in the scenario analysis.

Scenario Planning

The purpose of scenarios is to find out the mechanisms through which the system achieves the desired behaviour. Using scenarios, the behaviour can be jointly explored by the developers and clients which helps to precisely specify the requirements. This step implements the identification of class semantics, the second step of the micro process.

Scenario planning is based on use-case analysis described above. A scenario is storyboarded for each set of related system function points detected for the system, using the key abstractions found in the previous step. Initially this can be done in the form of a text narrative or using the CRC cards[1] to facilitate brainstorming. As the distribution of responsibilities becomes clear, object or interaction diagram representation are used to describe the scenarios.

Drawing high-level state transition diagrams is a supplementary technique which was found helpful to explore the system behaviour. It can complement scenario planning to put the scenarios into context and identify the paths in system behaviour that need to be explored in more detail.

The product of scenario planning are interaction and object diagrams which describe the system behaviour, grouped into clusters of related function points. Also, the data dictionary is updated by adding the class methods and collaborators identified during this step.

2.5.3 Architectural Design

The design of the application architecture is the third macro process stage[2]. System architecture describes both the dynamic and static aspects of the system: the layering of classes/objects that reflects their run-time dependencies, and module hierarchies that describe the compilation structure. Being concerned with large-scale strategic decisions, architectural design does not use the micro process.

This stage has two main steps and associated products: establishing the architecture of the application, and setting up the tactical policies that affect various aspects of the design. Also, a release plan is established to form a basis for the system evolution.

Logical Design: Layers

The system layers reflect how the high-level functions are achieved by splitting them into smaller tasks performed by the classes at lower levels. In Booch s description, a layer denotes the collection of class categories at the same level of abstraction and layers are split into partitions [which] denote each of the peer class categories that live at the same level of abstraction.

In BOOD, layers and partitions are described using class categories which represent logical grouping of classes that provide a set of related services. The client-supplier relationship between classes is used to determine the layers of the system: if class A needs classes B and

[1] Class-Responsibility-Collaborator cards, described in more detain in [Boo94] and other sources.

[2] Booch uses the term Design for this stage in his book. However, the more descriptive term architectural design is used in this dissertation as well as in the documents produced for EASAMS Ltd.

C to perform its functions, then A would usually be in a layer above B and C. The associations between categories are thus based on clusters of class associations, simplified to reduce high fan-outs and mesh patterns.

Physical Design: Modules

The physical design describes the compilation units into which the implementation source code will be divided, using two levels of abstraction: modules and subsystems. Modules are used in the usual meaning and contain declarations and definitions of classes. Module dependencies determine the order of compilation or file inclusion. Subsystems represent clusters of logically dependent modules which can reside in one subdirectory tree.

The physical design needs to be considered carefully as physical partitioning affects the possibilities of independent and parallel work on different parts of the system. The module hierarchy is based on class categories in general, but inheritance and containment dependencies can affect the allocation of some classes into particular modules.

Design Policies

Various aspects of the system design, called tactical policies in BOOD, need to be considered before the detailed design begins. Main issues in this area are general error detection and handling, memory management, and flow of control (e.g. client-server, event driven), as well as domain- and system-specific issues like optimisation. The purpose is to prevent these policies to develop in an uncontrolled ad-hoc manner during the design process.

Tactical policies are described in an appropriate document. Where possible, the policies should be illustrated by scenarios that capture their semantics and mechanisms. Booch also suggests to carry out a walkthrough of the policies, mainly in order to propagate the ideas throughout the team.

2.5.4 System Evolution

During the evolution stages the design is gradually refined into the implementation until the system is completed. The system evolves in separate releases horizontal or vertical slices of the system from the core towards the complete functionality. The main advantage of this approach is the early delivery of at least a skeleton system version which facilitates early feedback from the user and has important psychological benefits.

A system release, the main product of the evolution stage, is developed by performing one or more spins of the micro process during which the classes relevant to the release functionality are identified, specified and implemented. The degree of implementation detail depends on the current design level: classes are specified at the beginning, and implemented towards the release completion.

One iteration of the evolution stage consists of three main parts:

1. First, the function points which the current release should implement are identified. They are taken from the list prepared during analysis and design, and the scenarios that describe them serve as a basis for the work on the release implementation.
2. Next, one spin of the micro process is initiated and performed. During its execution, the four steps described in the previous section are performed with the effort focused on the abstractions related to the function points of the current release. Regular design reviews and other management practices should be used to ensure the quality of the development process.
3. After the implementation step of the micro process is completed, its results are integrated into the system through change management and the release is produced. Design changes need to be controlled due to their different relative cost. For example, changing a method

implementation usually affects very little of the rest of the system, whereas modifying a class interface or reorganising the inheritance hierarchy can have a substantial impact.

2.5.5 Maintenance

The last development stage represents the postdelivery evolution of the system. The changes required after the system is delivered to the customer are incorporated into the system using same methods as in the evolution stage. The only difference from the architectural design and evolution is the reduced amount of design innovation.

The corrections and/or enhancements requested by the customer are grouped to serve the same purpose as function points, but they must be prioritised so that important enhancements or bug fixes are handled first. Then the same process as in the evolution stage is started, resulting in a release which addresses the selected problems.

Chapter 3: CASE Tools

3.1 Introduction

This chapter contains a theoretical background to the principles and use of Computer Aided Software Engineering (CASE) tools. The support these tools provide for software development in all stages of the lifecycle, as well as for the associated process management tasks, is becoming more important with the growing maturity of software design methodologies and an increasing need for effective team development of high quality software. These issues were recognised in the software development at EASAMS Ltd. and resulted in a requirement to evaluate the possibilities for the use of CASE tools as a part of this thesis project.

After presenting a taxonomy of CASE tools and important aspects of their functionality, the problems with adopting these tools in wider use are discussed in this chapter. The final section presents a list of requirements that a good Integrated CASE tool should fulfil. The results of actual evaluation of two tools are presented in chapter CASE Tools Evaluation towards the end of this dissertation.

3.2 Types of CASE tools

There is a wide variety of CASE tools available on the market today, each offering a different level of functionality and support. To ease orientation in this field, researchers and practitioners have come up with categories that group the tools according to these levels of functionality. These taxonomies provide information about both the quality of the tools and their historical development as these two things come close together in this rapidly evolving area.

One common way of classifying CASE tools is described in [Kel94] and summarised below:

- *Lower CASE* are tools that support the basic procedures in one phase of software development, e.g. capturing system design in a notation of a particular methodology;
- *Upper CASE* tools cover system analysis and design as well as software production support;
- *Integrated CASE* tools provide support for the whole software lifecycle including team development and management tasks, e.g. version and configuration control, planning and progress tracking;
- *Component CASE* are based on a defined application and data interface which allows integration of tools from different vendors.

The increasing functionality of the tools in this classification is important for determining the areas in which each can be used to the best advantage. Lower- and Upper CASE tools may be suitable for an individual or a small team where the problems of shared access and communication of design ideas are not significant. For any larger-scale project however, an Integrated CASE environment is necessary to support the added tasks of project management and team development.

Integrated CASE tools are the current state of the art although many problems still need to be resolved. After the developers adjust to the changes in development culture, using these tools can bring real benefits, especially if they provide ways of tailoring to local specifics. Many of the Integrated environments are in fact on the Component CASE level, and these two categories are discussed below in more detail.

3.3 Role of Repository in I-CASE

To achieve their level of functionality, Integrated CASE tools must be able to maintain relationships between data of various kinds software component representations, design diagrams, methodology-specific rules, etc. In addition to these data-data relationships, information needs to be related to the tools that perform various transformations upon it (e.g. from design notations to source code). Both the data and their relationships must also conform to the syntax and semantics given by the design methodology.

All these requirements can be achieved only if all the data is kept in a common format and centralised. This centralised store of project data is called *repository*. It can be formally defined as the set of mechanisms and data structures that achieve data-tool and data-data integration [Pre92]. Conceptually, there are two levels of data stored in a repository: project-specific data which describe the components of the software under development, and metadata which define how these pieces of data can be related to each other and to the tools that manipulate them.

The repository is build on a standard database platform (relational or object-oriented) and adds the functionality necessary for the CASE environment. [Pre92] lists the following functions a repository provides in common with other database systems:

- centralised, non-redundant data storage of all information about the project under development;
- high-level access to the data for all tools within the integrated environment, independent of the physical representation;
- transaction control to manage data integrity during concurrent user access and prevent losses in a case of system failure;
- security mechanisms that allow the users to access only the data and tools relevant to their work;
- query and report facilities that enable users to browse the repository contents;
- means of data import and export that facilitate information exchange with other tools.

The special features of a CASE repository are then defined as

- storage of data of varying granularity and conceptual levels text, diagrams, object relationships, semantic information, metadata, etc.;
- integrity (consistency) enforcement capability which triggers cascade changes on relevant objects when an existing object is updated or a new object added, also includes explicit integrity checks;
- internal data representation independent of its external presentation so that each part of the I-CASE tool can interpret it according to its needs;
- rules governing the software development process and project management tasks to ensure that both the product *and* the process of its development are correct.

Tools that contain a full-featured functionality using a repository can thus help the software development team not only to build the product in accordance with the chosen methodology but also to ensure the traceability and quality of the process itself. This is important today when it is realised that the quality of the software process is a major factor in ensuring the quality of the resulting product.

3.4 Problems with Tools

Despite the expectations, CASE tools are still used rather sparsely and/or inefficiently in industry. One of the surveys cited in [Kem92] has found that 70 percent of CASE tools are never used, and 25 percent are used by only a group and not to their full capacity.

There are several problems with adopting CASE tools successfully. Owing to their extensive array of features and the resulting complexity, they have a relatively long learning curve. This is very often further complicated by the time needed to learn a new methodology together with the tool, and augmented by the wide variety of tasks involved during software development.

When transformed into cost estimates, the effects of the learning curve can result in a decrease of productivity for a period of two to six month [Kem92]. This can be disappointing for managers who may hope to gain a relatively quick profit from the use of the tools.

Kemerer also points out that tools often have a varying level of support for individual project tasks. As the importance of some tasks changes between projects (some may require a complicated analysis, other ones detailed testing) the benefits will be different for each of them. This makes it difficult to evaluate the tools using a fixed set of criteria.

Last but not least, the relatively high cost of I-CASE tools may put off managers who are not fully convinced about the benefits of these tools.

All these problems emphasise how important it is to know what facilities CASE tools can provide and to select the proper tool with respect to the needs of the company. Next section presents a list of general guidance rules for choosing an Integrated CASE environment.

3.5 Requirements on I-CASE Tools

As a synthesis of and corollary to the previous paragraphs, the following is a checklist of criteria or requirements on Integrated CASE tools and. It can be used to help in assessing the candidate tools (the list was compiled using information from [Pre92] and [Gra93]):

- team support: Does it support shared and exclusive access to objects plus communication among the team members?
- openness: Is integration of tools from other vendors possible, is the repository directly accessible and implemented in a standard format (i.e. using an off-the-shelf database system)?
- tailoring: To what extent can the tool be tailored to the specific project, methodology or local culture needs?
- integration: Does it integrate both technical (e.g. design) and management (e.g. planning) tools which cover the whole lifecycle and which communicate through one common repository?
- consistency: Are changes automatically propagated to all relevant objects, how is the consistency checked (e.g. on input, by explicit command, etc.) ?
- version and configuration control: Does it support version and change control of the design files, configuration and baseline management, and in what way?
- document and code generation: Can the design etc. documentation as well as source code in a required language be produced automatically from the repository contents, and how does the tool handle manual changes to them?
- reuse: Does the tool provide an access to code and design libraries so that reusable components can be both extracted and added?

- metrics collection: Can the tool automatically collect technical and management metrics during the design process and are these metrics useful for quality improvement?

All these criteria concentrate on the most labour-intensive tasks which should naturally benefit from the tool support. In general, it should be always checked on what does the vendor base his claims of improvement in productivity and quality because 100 percent automatic source-code generation ... hardly makes [any difference when] programming code actually consumes at most 15 percent of development cost [Gra93].

3.6 Summary

This chapter presented a short overview of the functions of Computer Aided Software Engineering tools mainly with respect to team development. Two lists of checklist criteria were presented which can be used for a tool evaluation. The assessment of the tools used during the work on this thesis project is given in chapter CASE Tools Evaluation.

Chapter 4: CASE Tools Evaluation

4.1 Introduction

Booch s Object-Oriented Design methodology is very rich on notation and if it is to be used in full it needs a tool to support the diagram-based analysis and design process. This was one reason behind the requirement for this thesis project to evaluate the possibility of using CASE tools. Another reason was that the current design method used at EASAMS Ltd. is almost entirely paper-based, with the design diagrams produced using the drawing capabilities of Microsoft Word for Windows.

The management of EASAMS Ltd. acknowledge that the lack of tools brings problems to the software development, mainly in the areas of communicating changes through the design team and ensuring consistency of the design during its evolution. However, no tool support was sought so far because the design methodology in use is a non-standard one and because the tight time scales for the IFES project do not allow for any delays at all.

For the IFES Passenger Application BOOD development, two tools were chosen and evaluated by practical use: Rational Rose version 2.5 by Rational (demonstration version), and Object Domain version 1.02 by Dirk Vermeersch (shareware edition). These tools were chosen for the following reasons: Rational Rose is widely used in the industry thanks to its good support for BOOD, Domain is a publicly available tool with a very low price tag. Both tools were used in the Microsoft Windows version.

The rest of this chapter lists the criteria used for the evaluation, results for the two tools in question, and suggestions regarding possible adoption of CASE tools for EASAMS Ltd. software development.

4.2 Evaluation Criteria

The criteria are drawn from two sources: the requirements on I-CASE tools which are summarised in chapter CASE Tools, and practical experiences gained during the work on the project.

It could be argued that the choice of I-CASE criteria is inappropriate as none of these tools can be regarded as integrated. In the author s opinion this approach is valid because (1) these criteria represent the full set and therefore allow to determine the category of the tool and (2) any tool (not just CASE) should be assessed with future needs in mind, especially in software industry where today s state of the art technology becomes basic requirement in near future.

4.2.1 I-CASE Criteria

These requirements are common for the tools that are made to support the team development and management of quality-centred software process:

1. **team support**
   The tool should provide locking mechanism for exclusive access to the components of design for updates while allowing unlimited copies to be viewed in read-only mode; integration with email, information systems, word processing packages is very helpful.
2. **repository**
   The design data must be held in a repository which allows consistency checking and information sharing among the tools.
3. **lifecycle support**
   This allows to evolve the system from analysis through design to the maintenance stages of its lifecycle, with smooth transitions of the information between these stages.
4. **version and configuration control**
   The tool should integrate or support control of versions of the evolving design and of its parts, as well as provide a means to manage design differences for various implementation platforms.
5. **process flow control**
   The support of a methodology is only complete if the work on the design components of one stage can begin only after all the components of the previous stage(s) have been finished.
6. **management support**
   The tool should incorporate project planning, progress monitoring, metrics collection and other management-related functions.
7. **documentation and code generation**
   It is necessary that the design documentation and source code (at least in a skeleton form) can be generated automatically from the information in the repository.
8. **integration and data exchange**
   The too should be able to export the design information in a common format which can be used by other packages.
9. **extensibility and tailoring**
   The tool should have means for tailoring its functions to the particular environment in which it is used, including the process flow control functions.

4.2.2 Experience-based Criteria

These criteria were developed during the actual use of the tools for IFES/EK design using BOOD. They reflect the needs of a developer which uses the tool on a daily basis. For everyday work, presentation capabilities and ease of use are very important because they influence the human ability to deal with the complexity of the developed software.

10. **presentation**
    The information about the system under development must be presented in a clear way that helps the designers in understanding and orientation in the system.
11. **repository-related**
    The tools should have such functions as duplication checks, queries about objects (classes, diagrams), referencing same objects in various diagrams (e.g. a method in both the class and interaction diagrams for BOOD).
12. **diagram editors**
    The diagram editors must help to make the presentation clear, for example by including text annotations and formatting the diagram elements.
13. **level of detail**
    Information about objects both for input and presentation should be relevant to the current development stage (e.g. omitting method parameters in architectural design).

14. **ease of use**
    The user interface should be designed in such a way that the tool s functions can be accessed ergonomically.
15. **stability**
    The tool must be stable to prevent losses of information and work (unfortunately, even professional programs are prone to occasional crashes in the MS-Windows environment due to its lack of memory protection).

The following two subsections present the evaluation of the two CASE tools used during the IFES/EK Passenger Application BOOD development. Each criterion is evaluated both verbally and numerically. This allows to compute a compound value of the total quality of each tool as well as include comments explaining the reason for the value.

The scale used for the individual criteria and the compound value is ordinal numbers from 0 to 10, with the equivalent meaning ranging from not supported/useless to excellent . The final compound value is computed as their arithmetic mean where all the criteria bear the same weight. This is certainly a simplification, however it is very difficult to express the importance of the criteria using numerical values without a substantial research backing such decision. The value therefore gives only an approximate evaluation and needs to be used with the accompanying verbal assessment.

4.3 Evaluation of Rational Rose/C++ v2.5

The Rational Rose tool is produced by Rational Software Corporation, Santa Clara, CA. For the evaluation purposes, a demonstration version was obtained from Rational UK, Brighton, East Sussex. This version has all the functionality of the full version but the design is limited to contain the maximum of ten classes. The full version for Microsoft Windows cost £1,900 at the time of writing.

Individual criteria:

1. **team support 8/10**
   The design can be divided into separate parts (called controlled units) that can each be modified by different developer using standard check in / check out functions, and write protection can be set for these units; however no direct support for communication is provided.
2. **repository 9/10**
   All the necessary repository functionality is provided, including queries about objects and consistency checks with detailed error reports; however the repository is not held in a standard (database) format and is not directly accessible.
3. **lifecycle support 5/10**
   Both the strength and a weakness, there are no explicit boundaries between development stages and the design can be easily gradually refined, but no specific support for the stages is supported (this is rather surprising with regard to the clear distinction of the stages in the methodology and the close link between the author of the methodology and the company that produces the tool).
4. **version and configuration control 4/10**
   No support for this functionality within the tool itself; at least the design is held in an ASCII file which can be used by external tools like RCS without major problems (this strategy is advocated in the Rational Rose user manual).
5. **process flow control 0/10**
   There is no support for sequencing the development of design components.
6. **management support 0/10**
   No functions towards this point nor any links to external tools are provided.

7. **documentation and code generation 9/10**
   The tool has a very good C++ source code generation support with a degree of customisation, in addition user changes to the produced code are possible and preserved by the tool, and also full documentation of class specifications can be generated; though the latter one is hidden under the Print specification menu command.
8. **integration and data exchange 3/10**
   The petal files used by Rational Rose for design representation and export are human-readable ASCII files but to the author s knowledge not are not standard and interchangeable with other CASE tools.
9. **extensibility and tailoring 2/10**
   The only tailorable option is the amount of information included in the source code generation; there is no support for tailoring the methodology or the information presentation.
10. **presentation 9/10**
    The diagrams are well readable and annotations, zooming and text formatting facilities are available; the only possible problem is that the presentation is purely diagram-based without any possibility to display the structure in a textual form.
11. **repository-related 10/10**
    Good support for all the required functions including detailed error reporting.
12. **diagram editors 10/10**
    Good support for both drawing and printing the various diagrams.
13. **level of detail 5/10**
    The dialogue boxes for object specifications input are the same regardless of the development stage; at least they are easy to comprehend and the amount of class specification information displayed in the diagrams can be easily changed.
14. **ease of use 8/10**
    The tool uses toolbars and an intuitive way of entering information, but navigation in large designs can be potentially difficult.
15. **stability 9/10**
    Problems with the tool s stability were accounted but were very rare.

Compound value: 6.6/10

From its list of features it is clear that Rational Rose is not an Integrated CASE tool, especially due to its lack of support for process flow, management functions and standard repository format. But it does not claim to belong to that category, and can be regarded as a good Upper-CASE tool.

For the needs of EASAMS Ltd. this tool (in the full version) may be actually better suited than a fully featured but non-customisable Integrated tool because of the specific requirements on the software process the company needs to meet. The management, version control, etc. procedures are already well established in the current management practices and therefore their lack in the Rose tool need not be seen as a disadvantage.

4.4 Evaluation of Object Domain v1.02

This tool was developed by Dirk Vermeersch, San Jose, CA. A shareware version of the tool was obtained from a mirror of the Simtel ftp archive and used for a one-month evaluation period. It differs from the full version, which is available upon registration from the author and costs US$99, by a message text included in all printed diagrams. The shareware version is otherwise fully functional.

Individual criteria:

1. **team support 0/10**
   The tool has no support for multi-user access.
2. **repository 4/10**
   There is some kind of repository but its use is very rudimentary (allows referencing classes in different class diagrams but no use of class specification in object/interaction diagrams for example).
3. **lifecycle support 6/10**
   There is no direct support to distinguish the lifecycle stages; the only feature related indirectly to this point is that the C++ source code can be generated only from module diagrams whose contents is defined by the preceding work on class specifications.
4. **version and configuration control 0/10**
   There is no built-in support for these functions or any way to control the design versions externally.
5. **process flow control 0/10**
   There is no support for controlling if prerequisites for design components have been completed.
6. **management support 0/10**
   No management-related functions are provided.
7. **documentation and code generation 5/10**
   The tool can produce both structured text descriptions and C++ source code from the class specifications; however, both functions are implemented in a rather clumsy way which makes a lot of reformatting necessary.
8. **integration and data exchange 2/10**
   There is only an option to cut&paste the diagrams in a bitmap format; however this is of little use due to the big size of the bitmaps and the consequent problems with their integration into documents, and there is no possibility to export the data in any other format.
9. **extensibility and tailoring 0/10**
   There are no options that would allow to customise the tool.
10. **presentation 7/10**
    The tool uses a window with a structured list of all diagrams which makes the orientation in the design very easy, and has a reasonable way of handling text display and zooming; however the input dialogue boxes are very complicated and no keyboard or mouse shortcuts are used.
11. **repository-related 2/10**
    The use of repository is rudimentary and the tool will even allow to use two different class specifications under the same name, although some diagram-rule checks are made on input.
12. **diagram editors: 6/10**
    Graphical presentation of the diagrams is good but the text annotations are rather basic and handling messages in object and interaction diagrams is awkward.
13. **level of detail 4/10**
    The dialogue boxes for class and association specifications contain all levels of detail without any structuring; at least most information can be safely omitted when it is not needed.
14. **ease of use 4/10**
    Despite some good ideas about information structuring, the access to the design information is often difficult and not very intuitive.
15. **stability: 5/10**
    The tool has some serious internal problems which lead to incorrect memory handling and subsequent crashes, in certain cases making the whole design file unreadable.

Compound value: **3/10**

To conclude, the Object Domain CASE tool may probably be regarded as an Upper-CASE only because it supports both design and coding stages of the software lifecycle to a certain degree. However, its shortcomings and lack of stability prevent its use for any serious project. It can be considered as an option perhaps for an individual developer because of its low price as compared to the professional tools.

4.5 Application of CASE for EASAMS software development

In the present situation it is unfortunately rather difficult to introduce an Integrated CASE tool into the EASAMS Ltd. software development mainly for two reasons: (1) the time scales for the IFES project do not leave any space to accommodate the effects of the tool adoption, (2) the diversity of projects on which the company works makes it difficult to use a single methodology which would benefit from CASE tool support for all of them.

However, the need may not be for an I-CASE specifically. As most practices of a good project management are already in place, what is most needed is a tool that would support the actual design and coding work. The Rational Rose tool could be fully used for this purpose as soon as the IFES project starts using object-oriented design methods.

In the meantime, its support for scenario analysis in the form of object and interaction diagrams would be beneficial. Also the current object-based design could be held in the tool repository since its components can be relatively well represented as classes. This would bring the benefits of a central reference which can be accessed by everyone but whose parts can be modified only by the appropriate persons. However, it may be difficult to justify the cost of the CASE tool for such a limited use.

The situation of EASAMS Ltd. is an example of the readiness for CASE issue, and the suggestion made in [Kem92] that developers are recommended to delay adopting integrated CASE tools until they are fully comfortable with the underlying methodology applies to a certain extent.

4.6 Summary

This chapter has presented an evaluation of the two CASE tools used during the work on IFES Passenger Application BOOD design. It first describes the criteria used for this evaluation, based on the checklist given in chapter CASE Tools, as well as their approximate numerical values.

The second part of the chapter contains an evaluation of the Rational Rose/C++ version 2.5 and Object Domain version 1.02 tools. This evaluation shows the advantages a professional tool can offer over an under-developed one. Also included at the end are suggestions regarding the use of tools for the EASAMS Ltd. purposes.

Chapter 5: IFES Requirements

Summary of Passenger Application Requirements

5.1 Introduction

This chapter summarises the requirements specification of the Passenger Application for IFES. The requirements are based on the relevant EASAMS Ltd. documentation and have been transformed into a narrative form.

5.1.1 System Structure

The Application Software for the whole IFES consists of four major areas:

Flight Attendant Applications

The FA Applications are hosted at the control panel (and optionally with limited functionality at the flight attendant workstation), the software provides extensive facilities for system administration, monitoring and controlling entertainment facilities, and running a duty free sales operation.

Passenger Applications

The Passenger Applications are hosted at each seat. The software supports a number of passenger services including entertainment programming review and selection, duty free and catalogue shopping, game selection and air-to-ground telephone functions. The structure of these applications may support additional applications from third-party suppliers.

Automated Applications

Hosted at the Cabin File Server, the software operates alongside the Application Database to build a comprehensive record of a given flight leg including passenger transaction details, statistics of services used by the passenger, and archiving of a flight leg for down-loading on a later stopover.

Ground Based Support System Software

The software, hosted on customer provided equipment, provides the airline with facilities for preparing the data at the ground site for loading on-board applications.

5.2 Requirements Summary

This sub-section provides a high-level summary of the software applications of the IFES. It should give a concise but good overview of its functionality relevant to the scope of this dissertation.

Video

The IFES supports up to twenty distributed video entertainment channels. Each channel would be described on a main menu with a listing of selections. The FA has the ability to program show times and monitor source operation. The passenger selects the channel of

video entertainment via the handset or touch screen and can select an alternative language soundtrack for films where available.

Audio

Audio entertainment allows customers to select one of a number of audio channels in either stereo or mono. The passenger may have the capability to view any channel description listing on the video screen while listening to another channel. FAs have the capability to monitor audio channels from an audio jack at the control panel.

Any audio channel (excluding audio associated with a pre-recorded video entertainment) may be combined with any interactive application not having an associated audio. The IFES assigns a default audio channel and retains the audio selection when a non-audio application is selected.

Games

Customers have the option of selecting games through the IFES. A screen describing the games is displayed via the main menu. From the game description screen, the passenger is able to select the game, and begin to play. The game applications are supplied by TPVs and their co-operation with IFES is maintained via a standard API. Game being executed can be paused for the duration of an announcement. Pause and restart capability will be the responsibility of the supplier.

Duty Free Sales

The duty free sales function allows passengers to place orders for duty free products at each seat. Passengers are able to display product information in an alternative language other than default English, select products for purchase, select a form of payment including payment in multiple currencies, and record credit card information at the seat place.

The system maintains a running inventory of all products and notifies passengers if a product is not available. It also maintains an audit trail of all transactions and controls access to the duty free function at the control panel and flight attendant workstation.

Catalogue Sales

The catalogue sales function allows passengers to place orders for merchandise available from a third party vendor. Passengers are provided with the capability to display product information, select products for purchase, record credit card information at the seat place, and identify the location to which the products should be delivered.

The system records all catalogue orders, including payment and delivery information. The data is handed off at the completion of the flight to the catalogue vendor. The catalogue sales vendor is responsible for any sales accounting and product delivery. In addition to using the IFES to place orders, passengers may be offered the ability to order items via a telephone call to the catalogue service provider.

Information Browser

This facility allows a Passenger to browse text and / or bitmap information on the touch screen. This information comes either from the airline or from TPVs.

Telephone

Passenger telecommunication services are initiated through menu choices or the "new call" button. The card reader at the seat is used for collecting card information. This data is to be validated by the service provider before a call is placed.

Help

Passengers may request context sensitive help on all functions of the passenger application at any time during use of the passenger services. This information guides the passenger in the use of the system, and other facilities, that may be available.

5.3 Other System Characteristics

Passenger Services

During the course of a flight, a passenger may review all transactions or purchases made via the system. This facility allows a passenger to review their account with the system, and verify each payment made. The passenger is able to select the language, from those supported, in which they wish text to be displayed.

Flight Transaction

All sales events such as catalogue sales, games payments and refunds which occur during a flight leg, are recorded as flight transactions within the applications database. This data is handed off via diskette as a part of final closeout.

Payments

The passenger is able to select to pay by credit card or cash from a list of currencies. The passenger can request a printed receipt verbally from the FA. If a service is not complementary then the passenger has to pay for that service before it is provided and is able to exit and re-enter entertainment items previously purchased without further payment.

The IFES allows for charges to be applied toward video, audio, games, and telephone services. The IFES checks the integrity of the data on a credit card and display acceptance or rejection at the seat. System records the passenger's payment and calculates the change due.

Menu structure

The facilities provided by IFES are divided into a hierarchy of services and categories within services for the purpose of menu selections. The passenger can browse the menus by changing service, selecting a category, and selecting a topic within the category.

5.4 User Interface

The passenger application interacts with the user by the means of a colour LCD display combined with a touch screen. User selections and alphanumeric data can be also entered via a handset which combines application control and telephone functionality. The application user interface uses menus and dialogue boxes for user selections, and overlays for changing system parameters like headset audio volume.

The picture below shows a schematic menu screen of the Passenger Application IFES. The horizontal menu on the bottom lists a few of the available services, the rest is accessible by scrolling its contents sideways. The menu in the centre shows the categories and topic selectable within the Business Corner service, with the Latest News highlighted. System buttons are on the top of the screen.

Figure 2: Schematic Screen Example Menu

Examples of the real user interface (screen snapshots etc.) of the IFES applications unfortunately could not be included in this dissertation because they are a property of the various airlines and under commercial confidentiality protection.

Chapter 6: IFES Design

Description of the Passenger Application Design

6.1 Introduction

This chapter presents the design of the In-Flight Entertainment System Passenger Application. In accordance with the thesis objectives, it was developed to provide an example design using Booch s methodology and to evaluate the possibility of code reuse form the current IFES design.

The contents of this chapter can be divided into three main parts, following the main stages of BOOD macro process. The first part presents the analysis of the Passenger Application requirements with a list of function points that were used in the behaviour analysis.

The second section describes the architecture of the passenger application software, outlining the main class categories, their functions and relationships. A discussion of optimisation strategies is also included.

The last section presents the Level 2 design of the Passenger Application. The attributes and methods of classes are specified, as well as their relationships. The design is concerned only with the part of the application classes related to video programmes.

Unfortunately the amount of design information had to be substantially reduced for this dissertation because the original design documentation produced for the use of EASAMS Ltd. has more than 100 pages. However all the important aspects and representative parts of the design are presented here.

6.2 IFES Analysis

The analysis of requirements on the Passenger Application behaviour preceded the actual design. Its main outcome was the familiarity with the system to be developed, and an initial description of its key components and function points.

Subsection 6.2.2 System Behaviour Analysis below lists the main functions of the system identified during analysis. The scenarios in the form of interaction diagrams that describe them are shown in Appendix A at the end of the dissertation. A sample of the other analysis product the initial form of the system data dictionary is included in Appendix B.

During the system analysis, the demonstration version of the Rational Rose tool version 2.5 was used. At this stage its restrictions were not limiting because the main work was concentrated around scenario diagrams. The tool evaluation can be found in chapter CASE Tools Evaluation.

6.2.1 Information sources

The main BOOD reference [Boo94] lists two major sources of information for the system analysis: the knowledge of the domain experts and the experiences gained from the system prototype developed in the preceding stage, conceptualisation.

This approach was not feasible for the IFES application analysis because the requirements are send to the company in the form of already finished requirements specification documents. For this reason, IFES analysis had to be done in reverse using these documents as well as requests to the analysts.

Three points can be made in this respect:

- The experience demonstrated that this is a feasible approach to analysis. However, it means effort duplication because there has to be some kind of analysis to establish the requirements. This is unavoidable unless the whole system is developed with all the participating teams using the same methodology.
- The results of analysis and the consequent design decisions may not conform to some requirements imposed on the design developed by EASAMS Ltd. (one of these requirements is a particular way of structuring the application architecture). This results from the different methodologies used by the two IFES developments.
- Scenario planning during analysis from the requirement documentation revealed a number of discrepancies in these documents mainly in terms of inconsistencies and undefined behaviour. This can be seen as an indirect argument for the superiority of BOOD over the structured method used at present, indicating that Booch s method is able to specify the system more precisely.

6.2.2 System Behaviour Analysis

This section contains the analysis of the system behaviour represented by its function points clustered by the main groups of the application s functionalities. They were identified from the IFES Passenger Application requirements documentation. In some cases more general function points that those directly indicated in the documentation were discovered, mainly in the areas of payment and information browser functionalities.

As described in chapter BOOD Methodology, the behaviour is described by scenario diagrams. During the analysis process, a scenario was developed for each of the function points. The corresponding diagrams are included in the documentation produced for EASAMS Ltd. For the purpose of this dissertation only several scenario diagrams were selected to illustrate the important aspects of the system functionality and are included in Appendix A.

The next several paragraphs list the function points of the Passenger Application identified from the system requirements documentation.

Audio

The following function points concerning the audio functionality have been identified.

- Browse audio categories and programmes
- Play a selected audio programme
- View description of a different channel than the one currently playing
- Switch to the current video channel

For an illustration of the scenarios used in the analysis stage, the following diagram shows the analysis of the first function point in the list above. See Appendix A for more scenario diagrams.

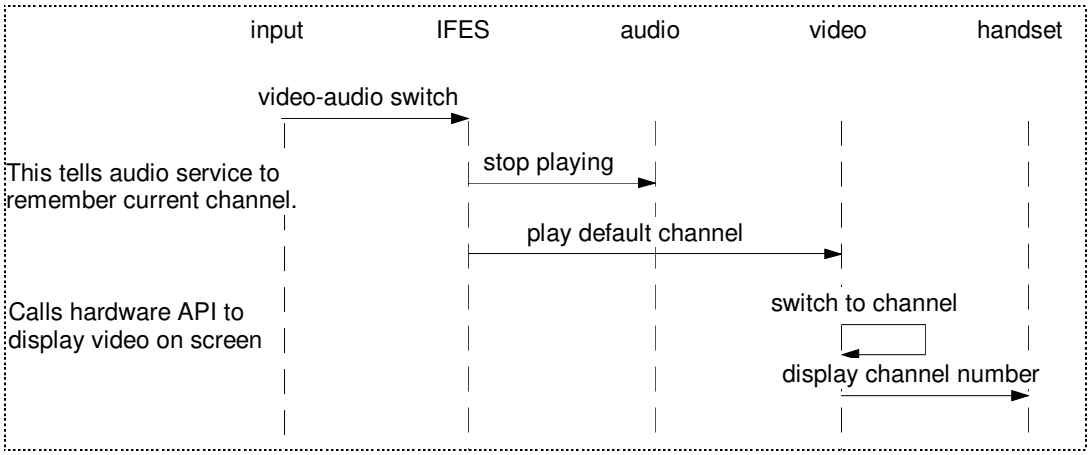


Figure 3: Example Analysis Scenario

Catalogue shopping

The following function points concerning the shopping functionality have been identified.

- Browse graphical screens with item descriptions
- Order an item
- Change the order
- Delete item from the order
- Cancel the order
- Review the order record
- Submit the order
- Switch to the current video programme

Games

The following function points concerning the games functionality have been identified.

- View description of available games
- View game instructions
- Pay for the selected game
- Pause a game during an audio/video announcement
- Change the game audio soundtrack
- Register for a multi-user game
- Start a multi-user game

Information Browser

The following function points concerning the text browsing functionality have been identified.

- Display information browser pages
- Page forward and backward in the pages

System

The following function points concerning the overall functionality of the system have been identified.

- System startup
- System close-down
- Browse menus up to an item selection
- Display help pages for the current context

- Change the screen and headphones attributes
- Activities during a passenger audio announcement

Video

The following function points concerning the video functionality have been identified.

- Select and play a video programme
- Change the video channel
- View the selected programme s description
- Switch to the audio service
- Scan video channels

6.2.3 Analysis Data Dictionary

An important product of the analysis stage is the initial data dictionary. Its entries are the classes that constitute the application implementation, and during analysis they are discovered as the key abstractions found within the system. A part of the analysis version of the data dictionary is shown in Appendix B.

The techniques used for the initial identification of classes were:

- identifying hardware components of the system these are the tangible things within it;
- domain analysis on the ARS and FRS documents domain experts were not available but their knowledge is contained in the specification of the system which identifies its main parts, services and mechanisms;
- scenario analysis as a mechanism for a function is developed, the missing components are discovered or invented.

6.3 Architectural Design

This section presents the architectural design of the In-Flight Entertainment System Passenger Application. The architecture is composed of the logical design of the system (partitioning into layers of operation), its physical structuring into modules, and includes a description of various tactical design policies.

An evaluation copy of the shareware CASE tool Domain version 1.02 was used to develop the architectural and level 2 design. The class diagrams and source code extracts found in the next section were produced in Domain and later formatted. See chapter CASE Tools Evaluation for the evaluation of the tool.

6.3.1 Design Assumptions

The Passenger Application design was developed with the assumption that a good hardware platform (e.g. an Intel i486/33MHz platform) will be used, without much need for software optimisation. This assumption was necessary to facilitate the development of an example BOOD design even though it is a significant difference to the platform available at present (this uses i386sx and 4MB of memory). However, the need for optimisation that is important for the present system was kept in mind in order to enhance reuse possibilities.

There is a potential for some inaccuracy of this design with respect to the customer requirements. This is caused by the need to work from draft requirement documentation during the analysis and design because the issued versions of these documents were not yet available. However, the design should be sufficiently open to provide an easy way to accommodate the eventual changes.

6.3.2 Logical Architecture

The logical architecture of the system was developed by analysing the client-server relationship between the emerging classes. As a result, the IFES Passenger Application is structured into the following hierarchy of class categories and sub-categories:

```
PassApp  (classes CPassApp, CContext)
AppServices
      ServiceItems  (classes
CItem,CPayItem,CAudio,CVideo,CGame,...)
      Browsers  (classes CInfoBrowser,CHelp)
      ItemCollections  (classes
CItemCollection,CPaymentSystem,CVideoSystem,COrder)
UserInterface
      Primitives  (classes from MFC: CWindow,CMenu,...)
      MenuSystem  (classes
CMenuIFES,CListMenu,CGraphicMenu,CCatalogueMenu,...)
Database      (classes CDatabase,CCacheManager)
Devices  (classes
CHandset,CNetwork,CDisplayCtrl,CHeadphonesCtrl,...)
GenericLists  (classes CList,CRing,...)
```

Figure 4: Application Layers

There are four main groups of class categories:

- `PassApp` which contains the root of the application;
- `AppServices` contains classes directly involved in providing the IFES functionality;
- `UserInterface` provides the screen GUI objects;
- `Database` and `Devices` provide the low-level support.

Each of these categories is expanded into more detail in the 6.4.5 Class Specifications section. If the category contains subcategories (not directly classes) the contents of the subcategories is presented in the order given in Figure 4.

6.3.3 Physical Architecture

This section describes the source code modules that constitute the IFES Passenger Application implementation. No module BOOD diagrams are included into this documents as they do not convey more information than the module hierarchy shown in Figure 5.

```
PassApp  (class CPassApp,CContext)
GUI
      Primit  (classes derived from MFC)
      Menus  (classes
CMenuIFES,CListMenu,CGraphicMenu,CCatalogueMenu,...)
Services
      BaseItems  (classes CItem,CPayItem,CProgramme,...)
      Items  (classes
CAudio,CVideo,CGame,CCatalGoods,CVendorApplication,...)
      Collections  (classes
CItemCollection,CPaymentSystem,CVideoSystem,COrder,...)
      Browsers  (classes CInfoBrowser,CHelp)
Devices  (classes
CHandset,CNetwork,CDisplayCtrl,CHeadphonesCtrl,...)
Database  (classes CDatabase,CCacheManager)
Lists  (classes CList,CRing,...)
```

Figure 5: Module Hierarchy

The figure shows the directory structure of the implementation. Classes will each reside in a separate file to facilitate parallel work on their specifications.

6.3.4 Tactical Policies

This section describes various policies that should be followed during the design and implementation evolution. The policies listed below are largely specific to the IFES development, few of them are general for the domain.

Business Rules Distribution

Provided the application model as described above is correct, the business rules information can be largely stored in the Application Database in the form of initialisation files and menu structure definitions. The information can be downloaded into the passenger application (namely the application driver and service objects) at start-up and during execution.

Class categories affected by different business rules for different airlines are the `AppServices` category and `CPassApp`, and partly also `CDatabase` which has to provide the appropriate methods. Although the affected parts will probably form bulk of the software, the application services model and class frameworks should ease code reuse (see section 6.4 Detailed Design for their description).

Control Flow

The distribution and flow of control between the application objects will be based on the MS-Windows event handling mechanism. This will allow greater flexibility for possible future extensions and should also ease integration of the IFES Passenger Application with the underlying software platform.

Depending on the hardware platform available the run-time overheads of the event driven system may be unacceptable and there may be need for optimisation of various degrees. The optimisation issues and their impact on design are considered in the next paragraphs.

Optimisation Strategy

As noted above, the application is designed with the assumption that there will not be much need for optimisation. However, the object oriented design makes optimisation easier than classical methods because it allows to localise it to the most critical parts. Within IFES the greatest need for optimisation is in the operations that provide on-screen display functionality. Also, keeping the same class interfaces prevents the changes to affect the implementation of functional mechanisms.

If the system was designed for the current platform, the performance and size optimisation methods would be (in order of importance):

1. Using direct method calls whenever possible and event handling in as few places as possible. This results in a reduced number of event handlers that are called each time an event needs to be processed. Use inline methods for attribute access and simple computations.
2. Caching often used information (e.g. menu service classes keep full lists the of categories and items they contain to reduce database access).
3. Design for a flat inheritance lattice to avoid too much dynamic binding. Implement objects with similar behaviour that differ only in some minor aspect as one class with a rich interface and behaviour modified by flags rather than use inheritance and virtual methods.

6.4 Detailed Design

This section describes the class interfaces of the IFES Passenger Application classes. It corresponds to the Level 2 design as used in the EASAMS Ltd. design process, denoting the stage which adds component interface specifications (function signatures) to an architecture established during Level 1 design. See the chapter Transition to BOOD and the list of terms in Introduction for more information.

In terms of the In-Flight system BOOD development it can be seen as one iteration of the Evolution stage. It builds on analysis and the architectural design and adds class interface details (i.e. the attributes and signatures of methods) to the general class architecture.

6.4.1 Scope

This design release of the Passenger Application implements only the functionality for playing a video programmes with the supporting system functions. This area was chosen for these reasons:

- The video service functions are very representative of the rest of the system both with respect to the flow of control, the logical information structure and its physical representation;
- It is a vertical slice of the system representing the first release, as corresponds to the evolutionary incremental strategy of BOOD;
- Building the design of the whole system would not be possible with the given resources, namely one person effort during a period of approximately one month.

In the dissertation document, the need for brevity further limited the specifications to the video-related classes only; the original design documentation produced for EASAMS Ltd. specifies all classes. Where appropriate, scenarios were developed as part of the design process to find out the class methods necessary for implementing the function points. In the documentation they explain the use of these classes. Several example design scenarios are presented in Appendix C.

6.4.2 Design Principles

There are two main ideas behind the IFES Passenger Application design: an application model based on generic service items, and a white-box framework design for reuse inspired by [Joh88]. Both were chosen in an attempt to achieve an open and easily extensible design. They are described in the following sections.

Application Service Items Model

The design attempts to provide a general way of providing the entertainment services in which the items in a particular menu can be of mixed types as identified in the airline requirements. An example is the business centre service whose options include information browser items, a video programme, audio programmes and TPVAs.

Based on this observation, the whole design is build around various kinds of items which are grouped into the menu service categories as required. On the implementation side the corresponding objects are very simple during menu selections, and expanded to provide the whole associated service after the item selection (e.g. to play an audio programme).

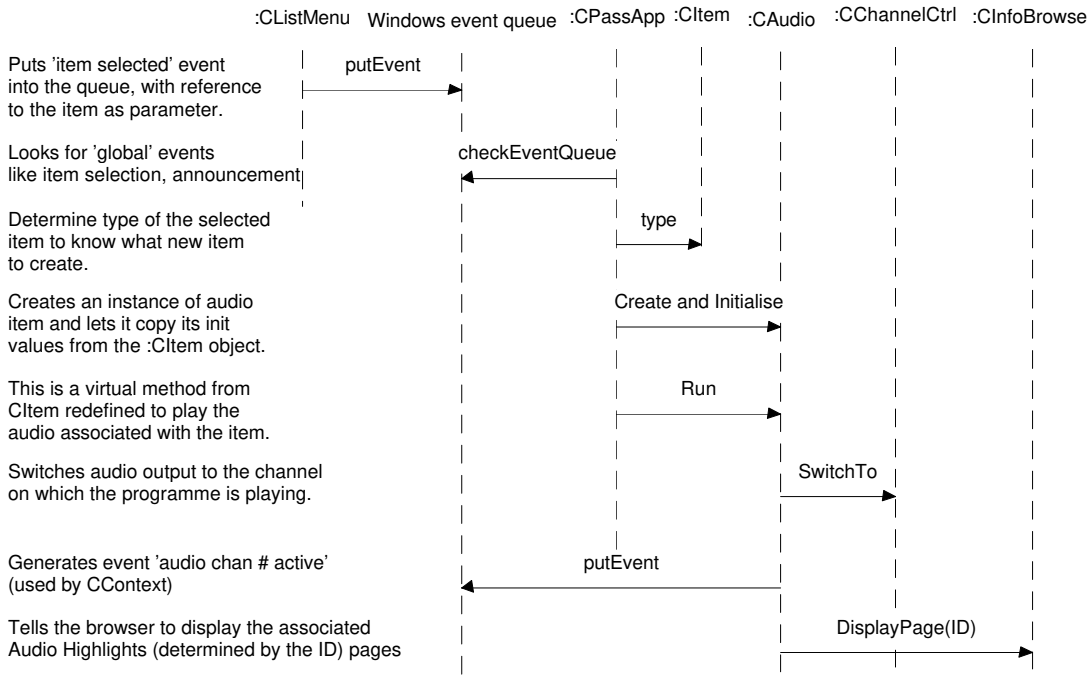


Figure 6: Item Execution

This mechanism is illustrated in the interaction diagram in Figure 6 which shows the object interactions when a passenger selects to play an audio programme.

In case this model of the Passenger Application is reasonably general (i.e. can be used to implement the In-Flight Entertainment System for any airline) it should lead to a very high degree of design and code reuse. Changes between the airlines would be restricted to the following areas:

- adapting the existing or adding new service classes to provide the required functionality (the class interface should remain the same);
- creating new specific item collections if the current payment and video systems don t offer the necessary functions; see also the next subsection for a discussion;
- restructuring the data in the application database (contents of menu and service items) and changing the bitmaps and shapes of the user interface objects;

Application Class Framework

The application design is also influenced by the idea of white-box frameworks for reuse described in [Joh88]. This is an approach to component reuse taken a step further: instead of a disjoint collection of reusable classes, the whole framework for the applications in the given domain is provided using abstract classes with polymorphic interface. A new application is then built by adding its specific behaviour through derived classes with appropriately redefined [virtual] methods.

Applied to the IFES Passenger Application BOOD design, this approach is used mainly in two areas: communication between the application driver and the menus using lists of generic items, and maintaining lists of items grouped for various reasons (e.g. to perform operations on both the whole list and its individual items). This is illustrated in Figure 7 with these areas shaded.

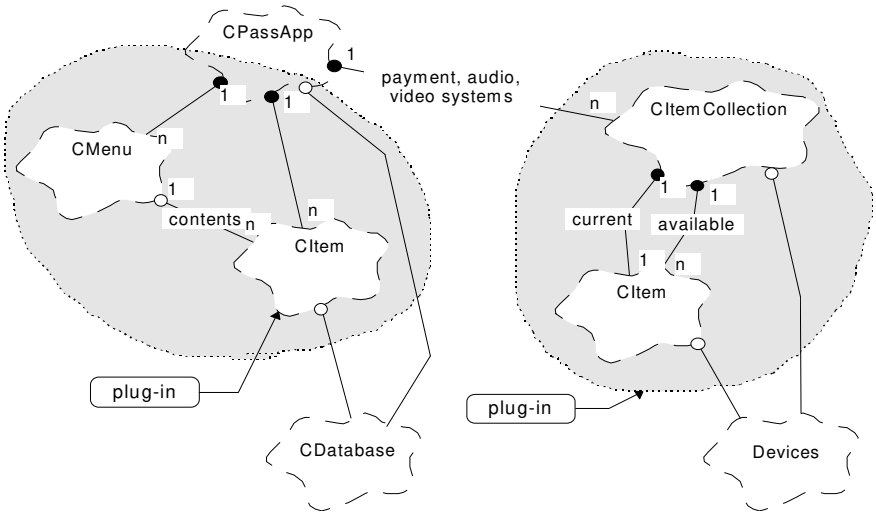


Figure 7: Passenger Application Class Framework

The picture shows the base classes that are used in the framework, although in the implementation the appropriate derived classes are often used. For example the systems are implemented by `CPaymentSystem`, `CVideoSystem`, etc. as necessary.

The plug in boxes point to the components that bring the framework genericity. In the menu area, new items of any kind can be added and will be handled by the user interface classes in the same way as the current ones as long as they are derived from `CItem` class. No specific information about the type of the items (audio programme, game, etc.) is needed for the menuing purposes.

After an item is selected, the object of the proper class is either created or looked up in the relevant item collection by the application driver. All the item objects can be then manipulated initialised and executed using methods defined in the `CItem` specification. Thus a uniform interface to the item functionality is achieved independently of its type.

Concerning the second area, any item collection derived from `CItemCollection` contains a list of references to `CItem` objects. The skeleton functionality provides access to the last item used, looking up items by their ID, and updating their list. The derived collections add the specific functionality, e.g. purchase sequence for the payment system or scanning video channels for the video system.

A new collection can be defined and accessed using the polymorphic interface of `CItemCollection`. The Microsoft Windows event mechanism makes it easy to activate its specific functions, for example purchase of an item, with only little code changes in the application driver.

6.4.3 Relevant Function Points

For this design release, only the function points related to the video functionality were used. The associated primary scenarios are:

1. System start-up;
2. Select a video programme in a menu;
3. Play selected video programme;
4. View video information and help;
5. Switch from video to audio.

During the analysis of these scenarios, secondary scenarios were necessary to explore the mechanisms for the following functionalities:

1. Payment for an item (outline);
2. Return to the previous menu level;

3. Stop a TPVA when switching to a video programme.

It should be mentioned that in several cases the mechanism identified during analysis could not be used for the design purposes because it was incorrect. The main reason was that the incomplete knowledge of the system during analysis and limited feedback from the company team members. If used, quality assurance practices like reviews should ensure correctness of the analysis.

6.4.4 Issues

Several areas of the IFES design remain to a certain extent unresolved and would require more discussion and investigation. Among the important ones are:

- validity of the service-item application model, mainly with respect to its generality and ability to cover future airline requirements;
- clearer identification of high risk areas within the application (need for optimisation, liability to substantial changes among airlines and associated amount of expected rework);
- the relatively high distribution of the business rules across the Passenger Application which could potentially pose problems in rework for other airlines.

Foreseeable Benefits

Despite these open issues, the presented design should offer the following benefits as compared with the current development:-

1. The service application classes have a relatively uniform behaviour and interface based in `CItem`. This makes it easier to add new kinds of services and/or extend the present ones if necessary, e.g. plug in payment into all services when its functions and implementation are clarified. As a result, design and code reuse should be fairly high. (In the current system these functionalities are designed in a rather rigid way.)

2. Provided the application model based on executable service items is correct, these items can be arranged in almost any way to suit very different kinds of entertainment systems. This should help to overcome the limitations of hard-coded business rules as used in the current system.

3. The device wrap-around classes will ease both host-based and target platform testing as they allow to use hardware stubs and real platform access under the same API. In addition they should be easily reused for other airline IFES applications as long as the hardware platform does not radically change.

6.4.5 Class Specifications

This section contains the specifications of the IFES Passenger Application software components. In particular, the classes related to the video service functionality are described. For class declarations see Appendix D.

With accordance to the Level 2 design definition, the design specifies class method signatures and describes their functions. Scenarios of the major functionalities describe how the high-level functions could be implemented; selected scenarios are presented in Appendix C. At this level of design, no or very little information is provided about the method implementation details, and in some cases the representation of data types is not fixed.

Form of Specifications

The design information is expressed in two forms. The class diagrams describe the relationships of classes in a category plus classes from other categories which are related to them (the former are shown as full-size class icons, the latter as flat icons).

The class specifications are in the form of C++ header files extracts. Specifications are grouped by the class categories as described in the IFES (BOOD) Passenger Architectural Design document. See Appendix D for declaration of classes related to the video functionality.

Both the class diagrams and the source code were produced by the Object Domain CASE tool in which the design was developed. However, its limited capabilities for both diagram and source code production made it necessary to edit the output a lot.

6.4.6 Types

The following table lists the descriptions of the basic data types used within the IFES Passenger Application. They are used either for class attributes, or as a means of data exchange between the high-level classes and the database.

Name	Description
TAudioChanMap	function same as TVideoChanMap
TCacheItem	used by the cache manager and database to exchange data for caching
TCreditCardData	credit card data as read by the handset reader
TFacilityType	type of facility associated with an item, also used for requesting help pages
TItemID	unique identifier for each item within the system
TItemLevel	position of the item in the menu forest: services are roots and topics are leaves
TLanguage	list of languages supported by the in-flight system
TOrderID	unique identifier of each order, for reference
TPaymentID	unique identifier of each purchase, for reference
TResult	return value of methods, used for error indication and checking
TSeatData	contains seat identification information
TSoundtrack	determines the logical video channel associated with the given language.
TStatusPVP	used by the CPVPCtrl device
TVideoChanMap	maps logical video channels to the physical channels within the Boeing distribution core; the array has an entry for each logical channel
TVideoData	data defining a video programme as read from the database

Figure 8: List of Data Types

6.4.7 Classes

This section contains the specifications of the IFES Passenger Application classes. Each subsection corresponds to one class category as shown in Figure 4 on page 36. See Appendix C for examples of scenarios that illustrate the class behaviour, and Appendix D for declaration source code extracts.

A. Application Driver

This category represents the top level of the application. Classes in this category control the running of the Passenger Application and maintain its state.

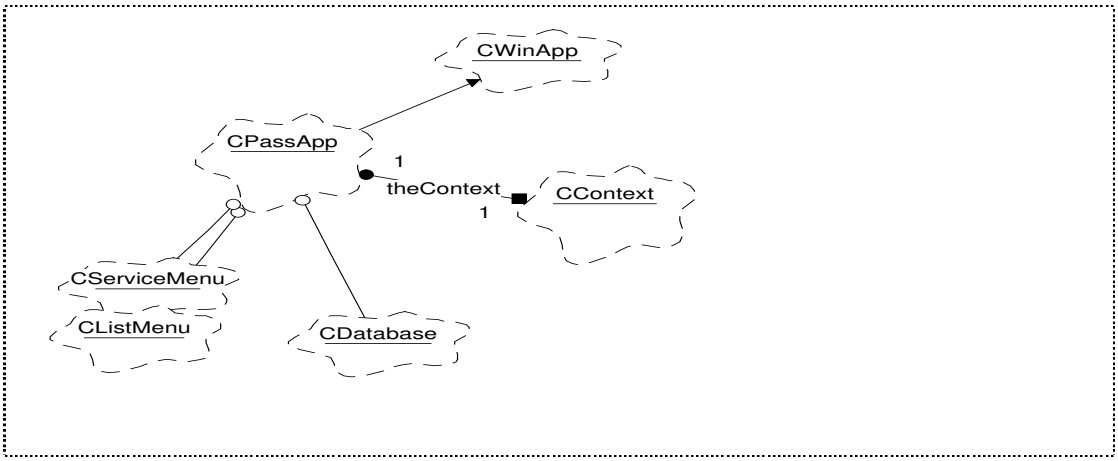


Figure 9: Class category `PassApp`

Description of Classes

The `CPassApp` class is the root of the application. It launches the various services, performs the start-up and closedown functions, instantiates all the necessary run-time classes, and handles responses to system events such as announcements and the Off button. This class owns most objects that constitute the application although this fact is not shown in the class diagram.

The system state information is concentrated in a specialised class `CContext` rather than being invisibly distributed among system components. There is one instance of this class in the system which is accessible by most of its objects. (The class specification is not included in this document.)

B. Application Services

This category contains classes that form the core of the IFES Passenger Application functionality by providing system s specific passenger services. It has several subcategories as shown Figure 4 on page 38. Details of these categories are described in the following text.

Service Items

The `ServiceItems` category contains the classes that provide the functionality of the Passenger Application. Instances of these classes will be created or looked up by the `CPassApp` object upon menu topic selection which will then ask them to activate their service.

None of these classes has direct access to the screen in any way. User interface access is managed by the `UserInterface` and `Browser` classes even in the cases when the functional relationship to the service item classes is very tight (namely for the information browser item class). This separates the internal mechanisms from the presentation, making the class responsibility boundaries clearer and implementation for other airlines easier.

Description of Classes

`CItem` is a universal base class that provides the basic information about any kind of service item for use by menus. This information is also used by `CPassApp` to create an object of the appropriate class to provide the associated service.

Abstract classes `CPayItem` and `COrderItem` provide payment and goods order methods to be inherited by the classes that need them. These functions are basically the same across the IFES services and it is therefore beneficial to extract them to separate classes.

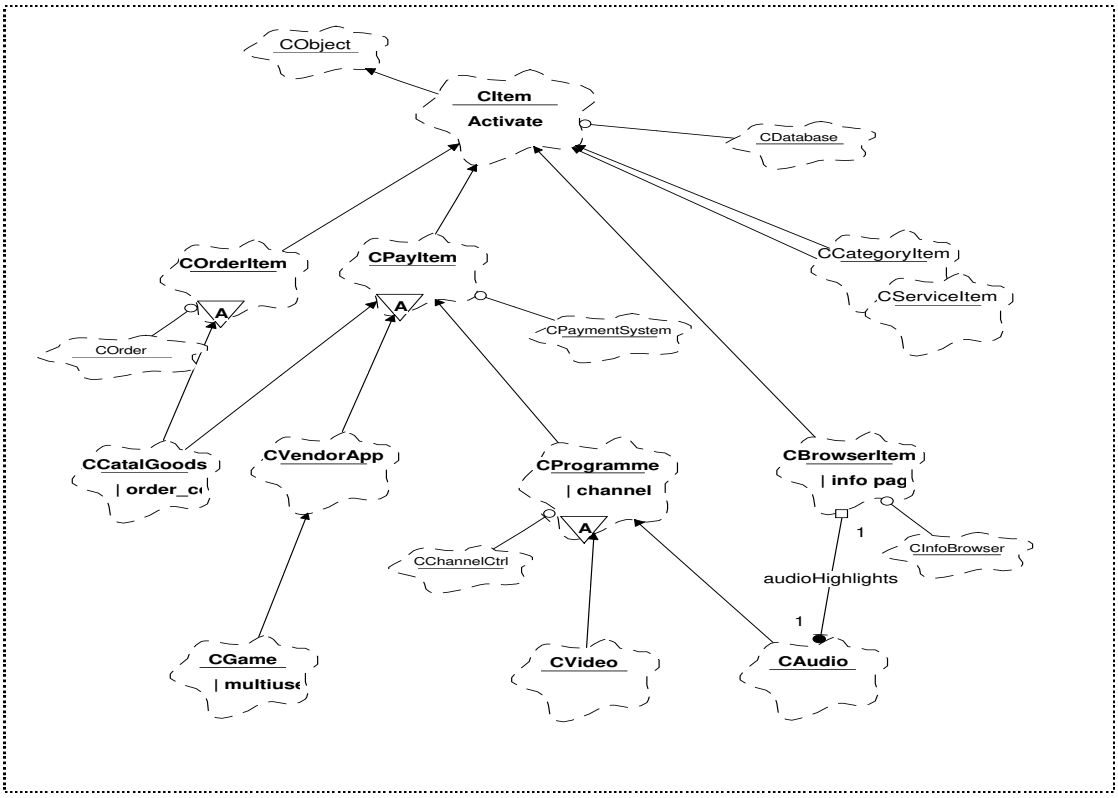


Figure 10: Class category `ServiceItems`

The seven main classes in the lower half of the diagram (`CCatalGoods`, `CBrowserItem`, `CAudio`, etc.) implement the various IFES passenger application services. They all have a redefined version of the virtual method `Activate()` inherited from `CItem` which provides an polymorphic entry point to their service.

Two scenarios related to `ServiceItem` classes are included in Appendix C. One illustrates the mechanism of activating a video programme selected from the menu, the other one the operations performed when a switch from audio to video is triggered by the handset key press.

Text Browsers

This category contains classes that provide the text browsing services in the form of an information browser engine and a help system class which uses an instance of the browser.

Description of Classes

The `CInfoBrowser` class provides functions of the information browser: displaying text described by the RTF tokens and its paging by the user. It is used by `CBrowserItem` and `CAudioProgramme` (category `ServiceItems`) to display their associated text and graphic information. This functionality is separated from the service item classes to make them independent of the display presentation that will differ among the airlines.

The context sensitive help is implemented by the class `CHelp` which contains an instance of the `CInfoBrowser`. Despite the apparent similarity of behaviour with the information browser, aggregation was preferred to inheritance. There are two main reasons for this decision: the two classes have rather different functional semantics (text viewer vs. help), and if required another screen representation can be given to the help pages simply by providing a different text pager.

The help object uses a referenced instance of `CContext` to determine the current application state and gets the corresponding help pages directly from the database. Thus it has no direct

connection with any other system component which should make it available in any state of the Passenger Application execution.

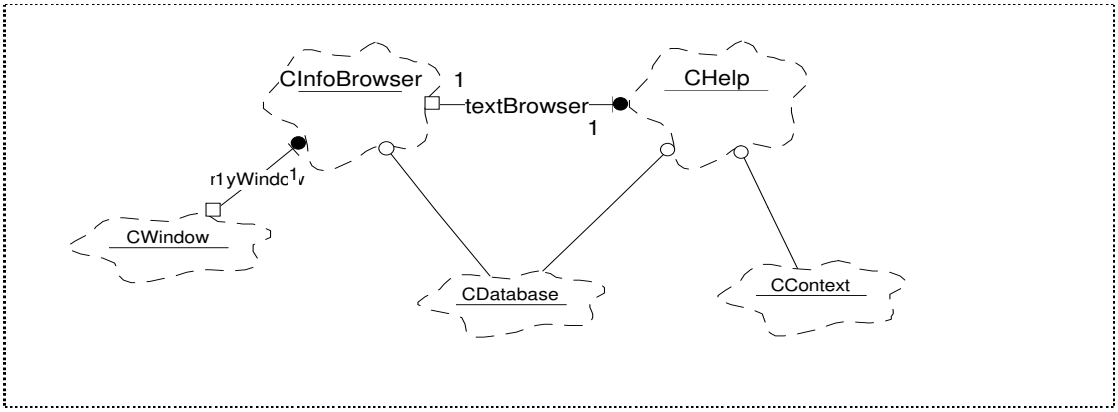


Figure 11: Class category Browsers

One scenario is included in Appendix C which illustrates the activation of the context-sensitive help.

Item Collections

This category contains classes that supply the payment and order functionality for the service item classes. These functions are basically the same for all Passenger Application services. It is therefore useful to implement these shared characteristics of the service classes in a separate layer of the inheritance lattice.

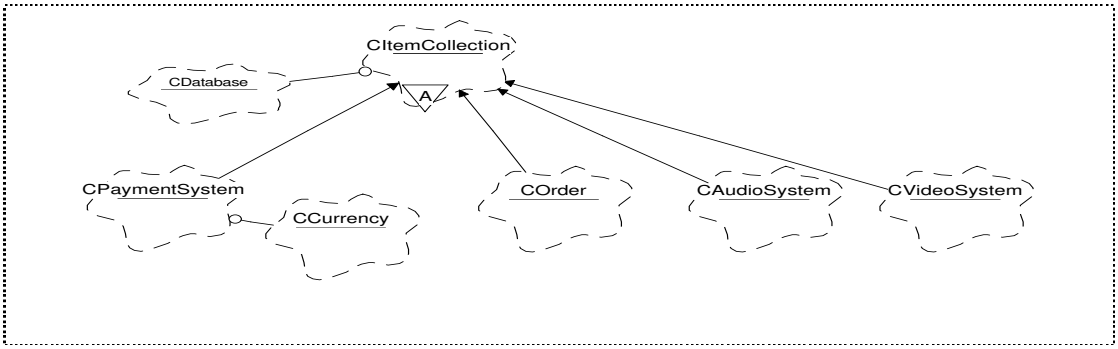


Figure 12: Class category `ItemCollections`

Description of Classes

As described in Application Class Framework on page 41, the classes in this category share the functionality of maintaining a list of item objects. This functionality is provided by the abstract class `CItemCollection`.

Class `CVideoSystem` maintains an up-to-date list of video programmes currently available at all the video channels. It is consulted by the `CPassApp` object when a video item is selected from the menu, and handles other operations like channel scanning.

Class `CPaymentSystem` provides operations for checking credit card information, payment by cash, and maintaining account information. All these functions are linked to the database and/or IFES Flight Attendant application for back-end calculations. Class `CCurrency` maintains a representation of currencies available for payments and their exchange rate against a selected base currency.

C. User Interface

This category contains classes that form the various elements of the user interface. It has two subcategories: `Primitives` which contains the basic user interface building blocks (views, buttons, windows, etc.), and `MenuSystem` which uses the basic classes to build menus needed in the IFES Passenger Application.

It is expected that the classes in this categories will be reused between different airline applications except for the shapes and bitmaps used for their on-screen representation. Building these classes by inheritance from the basic ones should also speed up the user interface development.

Primitives

The classes contained in this category are the basic user interface building blocks, and as such readily available from class libraries (e.g. Microsoft Foundation Classes). Reuse of the library classes is the preferred strategy for the IFES Passenger Application BOOD development because it increases programming efficiency and reduces development time.

Developing these basic classes afresh would be justified only if the use of library classes resulted in a provably inefficient and/or overly large runtime code. The `Primitives` category is included in the design to take account for this possibility.

Description of Classes

The meaning of and functions provided by the classes are that of the Microsoft Foundation Classes, i.e. `CWindow`, `CButton`, `CMenu` etc. A specialised class `CStringRTF` implements the display engine for text strings in the Rich Text Format.

Menu System

This category contains classes that define the functionality of the menus found within the IFES Passenger Application. Their instances use lists of `CItem` objects to obtain the items text and graphic information to be displayed on the particular menu, and return the item selected by pressing the OK button to the `CPassApp` object. Menu classes build on the interface primitives and process the messages generated by instances of these classes, namely the button objects.

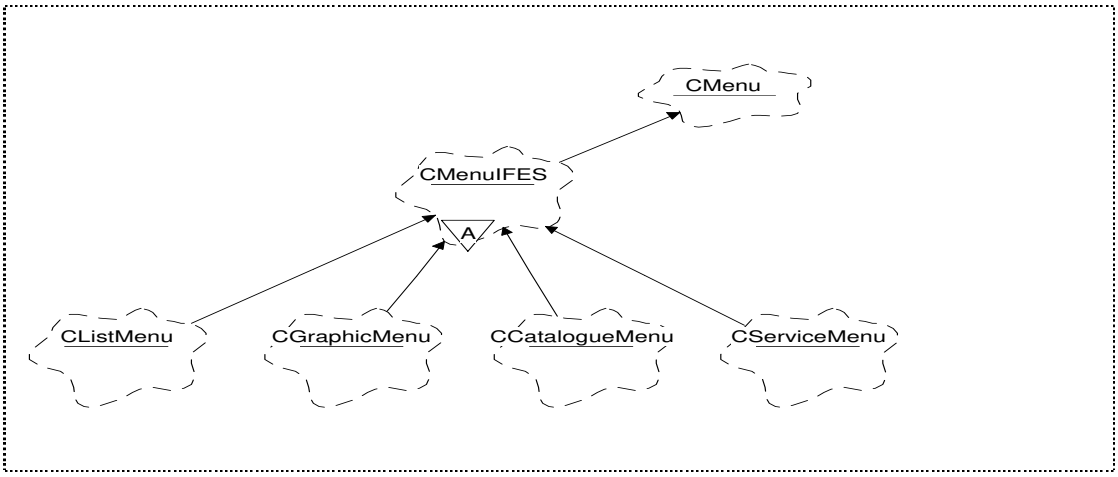


Figure 13: Class category `MenuSystem`

Description of Classes

The four leaf classes in the category represent the different variants of menus found in the requirements documents: `CServiceMenu` is the one on the bottom of the screen and displays the service icons and titles. It uses the left and right arrow button objects.

CListMenu is the common menu which scrolls item title text strings according to the up and down arrow presses. It displays the categories and topics that represent the available selections within the given application service. CGraphicMenu co-operates with CAudio and CVideo objects when the passenger browses the graphic previews. It displays one window of text and graphics at a time.

CCatalogueMenu is used by the shopping service class CCatalGoods during the purchase sequence. It mixes static text and graphics with scrollable menu items within one window.

From all the classes in this category, only the specification of the base class of all Passenger Application menus is included in Appendix D. The derived menu classes use the same class interface, overriding the pure virtual methods to provide their specific functionality. The scenario included in Appendix C illustrates how tracing back to higher menu levels can be done. Refer also to Scenario 4: Browsing Service Menus in Appendix A for additional information.

D. Database

Class CDatabase serves as an access point to the Application Database: it converts the support operations for the higher-level classes into series of database queries and processes the received or dispatched data. The CCacheManager provides low-level caching of raw data; only this cache should be used within the application in order to reduce duplication of data.

Description of Classes

The CDatabase is a class utility (a collection of relatively unrelated functions enclosed behind a class abstraction barrier) with a relatively large number of methods as needed by the higher level classes. Its functions can be more precisely defined only after these classes have been designed in detail.

E. Devices

This category contains classes that provide a controlled access to the devices of the hardware platform. Similarly to CDatabase they translate the function calls from the higher-level classes into the appropriate hardware API calls, and conversely the hardware-generated events into high-level ones.

It is difficult at this stage to determine any patterns of their behaviour or structure and therefore the invention of the potential base classes and the building any kind of inheritance hierarchy is left to the subsequent design stages.

Description of Classes

There are two logical groups of classes distinguished by the behaviour of the devices: those which provide input of some form (interactive) and those which only control the appropriate device without producing active output (control). However, only the specification of classes relevant to the video functions is presented in this dissertation for brevity.

F. Generic Lists

This category contains classes that represent basic abstract data types based on linked list which are used within the Passenger Application. Similarly to the basic user interface category, class reuse from a library is strongly encouraged. Alternatively, it should be relatively easy to transform the linked list functions developed for the current system into the appropriate classes.

Description of Classes

The category contains classes like CListItem and CLinkedList which have the usual semantics and are therefore not specified here in more detail. A CRing class which defines a

list with the head and tail items connected will also be needed for the wrap-around lists in menus.

6.5 Summary

This chapter presents the design of the IFES Passenger Application developed for EASAMS Ltd. in order to illustrate how to perform the BOOD development process. An analysis of the system was done first, using domain and behaviour analysis techniques. Subsequently the application architectural design was produced, followed by the design of class interfaces and relationships.

An associated issue of the use of CASE tools was investigated during the design development. The results of the investigation are presented in the form of the tools evaluation in chapter CASE Tools Evaluation of this dissertation. The produced design was also used to assess the possibilities of design and code reuse between the current design and the design based on BOOD methodology; the results are summarised in chapter Reuse Evaluation.

Chapter 7: Reuse Evaluation

7.1 Introduction

The evaluation of possible code reuse between the current design and the BOOD design was one of the project goals. The main motivation for this investigation was to find the level of similarity between the two design approaches and the likelihood of code transfer between them. This chapter presents a summary of the method used to determine the level of reuse and of its results. It is to a large extent based on the reuse analysis document produced for EASAMS Ltd.

Note

The results of the actual reuse analysis produced for EASAMS Ltd. are based on and include references to their source code. As such they are confidential and therefore not included in this dissertation; all code extracts and document references in this chapter have no correspondence to the documentation of EASAMS Ltd.

7.2 Scope

The reuse analysis covered all layers of the Passenger Application in accordance with the project goals. The BOOD design differs quite substantially from the current one, both in terms of the application architecture and its building blocks. Despite this, the same functional requirements and common low-level API on which both are based should result in a number of similar functions and data types, especially in the lower layers.

The resulting document identifies code mapping from the current airline Passenger Application Level 2 design to the Passenger Application BOOD Level 2 design. This is not an ideal basis for reuse evaluation because both designs are concerned with applications for different airlines (referred to as Airline A and Airline B in the following text) and the respective requirements differ. However it was the only possibility at the time of writing for the following reasons:

1. the BOOD design covers only the Airline B Passenger Application up to Level 2 design;
2. only the Airline A Level 2 documents were available for the analysis; Airline B documents were not yet in an issued (i.e. baseline) version.

In addition, no Level 3 documents were available for the Airline A Passenger Application and therefore the descriptions of module private (implementation) functions were not available. The reuse analysis is consequently based only on public module functions which complicated the assessment of the degree of reuse better estimates require the knowledge of implementation details for both the data and operations.

7.2.1 Documents

The following documents were used for the reuse analysis:

- Airline A Level 2 design document;
- IFES Level 1 (framework) document;

- Airline B Level 2 Design Alpha Release (internal memo, covers `Services` layer only).
- Airline B Passenger BOOD Architectural Design document;
- Airline B Passenger BOOD Level 2 Design document;

7.3 Reuse Analysis Method

The equivalent section of the reuse analysis document lists the current development Passenger Application functions and data types and indicates to what level they can be reused within the BOOD design. The target classes in the BOOD design are also indicated where possible. Several already available Level 2 design functions from the current Airline B development were also analysed.

Each layer of the IFES application framework was analysed in turn. For each layer, two separate lists of reusable and of non-reusable components were produced in the form of separate tables for function reuse and data types reuse.

The reuse analysis was done by comparing the description of functions and types in the design documents. Where additional information was available (e.g. in the form of source code or scenarios) it was used to determine the similarities in function s implementation. Using the knowledge of the target application design and professional judgement, the character and level of reuse was estimated for each component.

7.3.1 Estimates of Reuse

As noted above, target class(es) in the BOOD design are indicated for each component of the current design. Here, target class means the class into whose attributes or methods the component would be incorporated. In some cases the target could not be clearly identified; level 3 design information for both applications would be needed to facilitate better evaluation.

After each table, an approximate percentage of code reuse estimated for each layer was included. The format of this estimate is *Overall: x% from y% of current code* with the following meaning:

x estimate of reuse in the given layer; the conversions from the notes in the table are approx.:

partially	50%
maybe / depending on implementation	0% 50%
parts	60%
most or all except data representation	70%
probably all	90%

y percentage of reusable functions/types out of total number in given layer.

It should be noted however that the percentages are *rough* estimates only see the 7.5 Conclusions for a discussion about reasons. For practical use, they are to be taken as optimistic.

7.4 Examples of Reuse Tables

The two tables below show an example of the tables used in the original reuse documentation produced for EASAMS Ltd. A separate table was produced for each layer of the current application design. The fields in the rows have the following meaning:

Function/Type Name of the function or data type used in the EASAMS Ltd. design documentation;

Source Section in the relevant document included for traceability;
Destination The class or data type within the BOOD design which would use the original function/type;
Notes Indicates (verbally) the degree of reuse expected for the given component.

Because the original EASAMS Ltd. code is covered by commercial confidentiality, the contents of the tables has no relationship to the actual source code of the current IFES application.

Function	Source in design docs	Destination class in BOOD design	Notes what and how much reused
`WinMain`	section 1.1.4.2.1	`CPassApp`	partially, startup code and message pump

Overall: 40% from 60% of current code.

Table 1: Reuse analysis functions

Type	Source in design doc	Destination type in BOOD design	Notes what and how much reused
`GLOBALS`	section 1.3.5.1.1	`event types`	probably all

Overall: 90% from 100% of current code.

Table 2: Reuse analysis data types

7.5 Conclusions

This section presents the conclusions drawn from the reuse analysis and a summary of code reuse. The problems surrounding it are indicated, followed by comments about the reuse patterns and possible approaches to its exploitation.

In a summary, the reuse analysis showed that although there should be some level of code reuse, the really useful level of reusing whole functions and mechanisms will be very difficult to achieve. This conclusion is an argument in favour of the suggestion made in the Transition to BOOD chapter to develop the object-oriented IFES design afresh rather than by re-engineering the current application.

7.5.1 Reuse Evaluation

The greatest level of reuse should be from the low-level support modules, with target classes mainly in the `Database` and `Devices` categories. This was expected due to the proximity of these components to the seat box hardware API which partially dictates the available data and operations.

Higher layers show decreased levels of reuse, contributing mainly to `CPassApp` implementation. Although the relative reuse from these layers is approximately the same as from the low-level modules, they amount less to the total figure because of the smaller number of functions they provide.

Including Level 3 details (module implementations) into the analysis may increase this ratio; however this is uncertain as the high-level mechanisms differ between the two designs.

Classes derived from `CItem` and `CItemCollection` can probably reuse only a small part of the current business rules implementation code.

7.5.2 Problems

There are two main problems in reusing the current code, resulting in the small figures. Firstly, the higher layers of the two designs are rather incompatible both in terms of data representation and the mechanisms used to implement the system functionality. This makes the current functions as well as data types of limited use, both the public and the implementation ones.

The second problem lies in the low-level data representation. While the basic attributes are same or very similar (channel numbers, programme titles, etc.) they are grouped in different ways in many cases. This means that the elements of the current structured types would need to be moved into the target classes and data structures on a one-by-one basis, involving an amount of manual work.

7.5.3 Reuse Patterns

The analysis showed that the similarities of the two designs are on a rather abstract level. There are a number of similar mechanisms both in the current and in the BOOD design, mainly in the areas of device control, item information retrieval, and payment for items (e.g. check credit card, then record payment). Most of these represent communication between higher level components and the Services layer or `CDatabase` and `Devices` respectively. However, the differences in implementation details described above will probably prevent substantial reuse of these functions.

Most types used for communication with the database should be reusable in the BOOD design. As long as the attributes of service items remain similar there will be no need to change their low-level data representation. This fact is interesting with respect to the claims that object-oriented design is easy to change and maintain because it is be based on data (usually relatively stable) and not system functions which change very often.

Chapter 8: Transition to BOOD

8.1 Introduction

This chapter describes the steps and techniques that were suggested to EASAMS Ltd. as a possible way of migrating from their current design process to an object-oriented one based on Booch s methodology. The ideas presented in the following paragraphs form a synthesis of the work on this master thesis project, bringing together an analysis of the process used at present and its products, the experiences with techniques and tools gained by developing the BOOD-based design, and their possible application during and beyond a software process transformation period.

The text of this chapter is divided in three parts. The current situation is analysed in the first section. This includes an overview of the development process used by EASAMS Ltd., observations about the current design of the IFES applications, and of the use of object-oriented techniques within it.

The second section presents a short list of ideas regarding the features of BOOD that could be readily used within the process and how this could be done. Lastly, a discussion is included concerning the steps in migration to a full BOOD process, and the transformation and reuse of the design and code already developed for the application. The benefits of such migration are also indicated.

8.2 Current Situation

EASAMS Ltd. are using an TickIT certified software development process. It has clearly identified stages and the design methodology in use must be certified by the company process quality engineers. Also, management-based quality assurance practices are in place, namely project planning with risk assessment and baseline management, design and code reviews, and documentation standards.

With respect to the In-Flight Entertainment System development, EASAMS Ltd. act as a subcontractor of GEC-Marconi In-Flight Systems Ltd. responsible for the production of the application software. This means the functional requirements are delivered to EASAMS Ltd. as issued documents causing some undesirable effects, namely requirements being imposed on the application s design (as opposed to its functionality) and communication problems.

8.2.1 Process

The IFES development process is divided into the following stages:

1. **Architectural, or Level 1 design**
   High-level application architecture is developed, modules and their functionality are identified.
2. **Level 2 design**
   The module interfaces are identified in the form of function signatures and data types.
3. **Level 3 design**

Implementation details (algorithms and private functions) are developed so that coding can start.

4. **Coding and Module testing**
   The application modules are implemented in the selected programming language(s). Functions are tested using stubs.
5. **Host-based integration testing**
   The whole system is tested on a host platform to ensure its correct functionality, using stubs for specific hardware API calls.
6. **Target platform testing**
   The system is tested on the target hardware to ensure correct hardware-software integration.

The design is developed using a modular decomposition methodology with some influences of object-oriented methods. It originates in the Hierarchical Object-Oriented Design (HOOD) developed in 1987 1989 for the European Space Agency but has been to a large extent invented to suit the needs and specifics of the IFES development process, mainly by adopting an incremental delivery approach. The established QA framework helps to ensure the stability and quality of the development process.

This approach was implemented mainly due to the tight constraints on resources available for the system development. For each of the airlines, a working system of approximately 85 KLOC (Flight Attendant and Passenger Applications plus the Application Database) has to be delivered by a team of twenty engineers in nine months. Moreover, the work on the different airline systems overlaps to a certain extent. Due to these constraints it was not possible to start using a methodology new to the team members whose background is in structured programming and the C language.

8.2.2 Products

The current design of the IFES applications has a clear structure, based on a hierarchy of function layers. These are composed of modules which cooperate on a client-supplier basis. The three main application layers define in turn user interface components and functions, application control which implements the airline s business rules, and low-level support operations i.e. database and device access (see Figure 14 below). This architectural design is generic to all IFES applications developed by EASAMS Ltd.

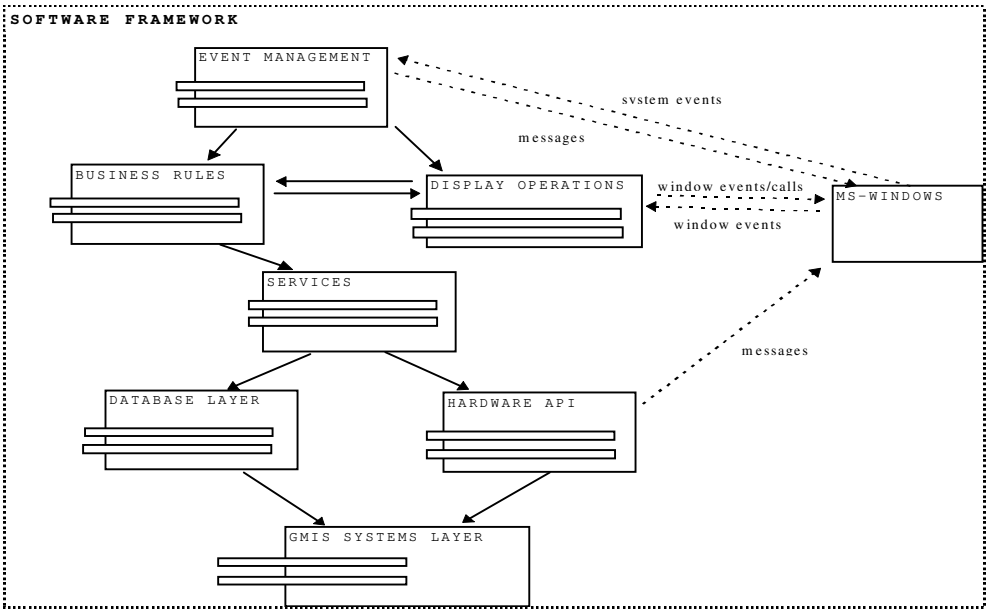


Figure 14: Architecture of the Current Design[3]

The main design goals of the current system are clear module interfaces to facilitate their independent design and coding, and localisation of airline-specific features (mainly business rules and user interface elements) into separate components to ease reuse between different airline versions.

Similarly to a good object-oriented design (or, indeed, any good design) the modules have well identified and separated responsibilities. However, the design resembles more a modular structured one rather than an object-oriented one: the modules merely define sets of functions which operate on data passed between the modules. For example there are functions to create a menu but there is no notion of the menu as a separate entity within the IFES application.

The current design is reasonably well structured and robust. Its main weakness is the overly rigid implementation tied to the particular airline specifics without much effort towards generalisation. This may result in decreased code reuse between the applications for different airlines. It can also cause major problems in perfective maintenance, i.e. implementing major functionality enhancements.

The design documentation uses Hierarchical OOD object diagram notation, plus state transition and object interaction diagrams. The state transition and object diagrams are used in a way relatively similar to BOOD, the Hierarchical OOD object diagrams are used to specify the logical structure of the application.

8.3 Readily Applicable BOOD Elements

Based on the knowledge of both the process used by EASAMS Ltd. and Booch s methodology applied to IFES, several suggestions can be made how to incorporate selected BOOD ideas into the process immediately, without any need to change the current methodology. However, the benefits of the methodology cannot be reached unless object-oriented design is used to a full extent.

The *use-case analysis* techniques using scenarios can be adopted during the architectural and Level 2 design stages, using modules and the functions they contain instead of objects and

[3] Diagram taken from the EASAMS Ltd. IFES Software Development Framework Document with permission.

their methods. Scenarios would be beneficial first to develop the mechanisms that implement the required functionalities, and later (during design reviews) to check that the invented functions work in the scope of the mechanism definition.

As was experienced in the Passenger Application BOOD development, scenario analysis during early design has also the added benefit of identifying discrepancies in the requirements specification. This early problem detection would be very useful in the current situation where delayed response to some queries about unclear requirements cause design problems or even plan slippage. Also, as Booch suggests in [Boo94], scenarios help to identify test cases for the integration testing stage.

A good object-oriented design leads usually to a high level of *component abstraction* with the benefit of a substantial code reuse. The emphasis BOOD places on pattern scavenging should certainly be an inspiration for the current process an effort to develop a design based on generalised components means an initial increase of work but can bring large effort savings in later developments, especially in a system like the IFES.

Lastly, more Booch s *notation* can be used for the current design description. The main area would again be the scenario diagrams, which can capture the use-case analysis as outlined above. Both interaction and object diagrams can be used for this purpose. It should also be possible to use a modified class diagram notation for the application architectural design.

8.4 Methodology Transition

To experience its benefits, the BOOD methodology has to be used in its full form. However, it is not easy to change the development process in an industrial environment where the associated cost of staff training and possible temporary decrease of productivity must be carefully considered. At EASAMS Ltd., the transition towards Booch s Object Oriented Design could be done in the following steps.

The pre-requisite for its use is a familiarity of the designers and programmers with object oriented concepts in general. This requires staff training and skill building before any serious work can start because the paradigm shift from structured methods is usually not easy. Also there is a need for both management and team willingness to undertake this step.

8.4.1 Steps

The transition would be best started by a relatively small-scale pilot project to gain initial experience with the techniques and development process in general. Within the IFES development for example it could be used to design one isolated part of the system (the display operations layer could be a suitable choice) which could communicate with the rest of the application via a defined API. The main issues to consider are choosing a low-risk area and allocating the appropriate personnel for the pilot project.

After the initial experiences have been gained, a full-scale switch to BOOD would be possible. Within the IFES framework it can be done by starting a new airline system development using the methodology or by gradually transforming the current design to an object-oriented one. The next subsection deals with these issues.

8.4.2 Re-engineering Strategies

With respect to the transformation of the current design into an object-oriented one, there are two basic strategies that can be adopted:

1. design the system from the scratch using an object-oriented methodology, which is the approach used in this thesis project;
2. re-engineer the current system into an OO-based, where the first approximation may be that the classes are based on current modules.

There may be compelling reasons for either of these: while the first approach should result in a clearer and more maintainable design, the second would help to keep the system functionality while gradually changing its implementation.

The chosen strategy influences the kind of reuse that would be possible, and consequently the level of component reuse:

1. only intersections of functionality can be reused, resulting in rather low yields;
2. current code would be transformed, resulting in potentially high yields.

Whichever re-engineering strategy is used, a degree of experience with object-oriented design methods is required. The first method (new design running in parallel with the current one) should actually be preferred because it does not require the reverse engineering step and is easier for newcomers to the object oriented methods.

8.4.3 Tools

As is discussed at the end of the CASE Tools Evaluation chapter, it is rather difficult at present to introduce CASE tools into the EASAMS Ltd. software development process. With the management framework in place, what is needed is a tool that supports the design and coding work. During and after the full methodology change, the Rational Rose tool can be suggested for this purpose.

In the meantime, the diagrams which are used to describe design and the underlying decisions can be produced using a good *graphic editor* such as Visio by Shapeware Corporation. Its main advantage is a seamless integration into the Microsoft Office suite currently used for design documentation production and good customisation facilities. However, the repository-based functionality (central point of reference, design checking, etc.) would be still missing.

To partially remedy this it can be recommended (as Booch suggests) that a *database* is developed that would held the system data dictionary and support team development. This would still leave a need for manual checking of design correctness but at least the communication of ideas and design parts among the team members would be easier.

As noted elsewhere, the situation of EASAMS Ltd. is an example of the readiness for CASE issue, and the suggestion made in [Kem92] that developers are recommended to delay adopting integrated CASE tools until they are fully comfortable with the underlying methodology can be endorsed.

8.4.4 Benefits

The advantages which the transition to BOOD as design methodology should bring can be summarised in the following points:

- more stable design, both the process and its product, because the centre of gravity of the development lies in the earlier design stages;
- increased degree of reuse between different airline applications, achieved by a more open and general design;
- good tool support as Booch s methodology is gaining strong support in the industry;
- the marketing advantage of using a modern and prospective methodology, with the associated benefits of possible know-how transfer to other projects (BOOD should be suitable for a variety of different application domains).

8.5 Summary

This chapter presents the end results of the master thesis project work in the form of steps proposed for adopting Booch s Object Oriented Design as the main methodology at EASAMS Ltd. The proposal is based on understanding the current development process and

its constraints, knowledge of BOOD gained during the example design development, and on experience with CASE tools.

The proposed changes fall into two categories. Some features of the methodology can be used within the current process without much need for its change; the scenario analysis technique is probably the most useful of these features. Secondly, the actual methodology switch should be preceded by a pilot project to gather the necessary experience and allow smooth change in the development culture.

Chapter 9: Conclusion

Changing the software development methodology is not an easy task in any large organisation. Apart from the psychological issues it brings the paradigm shift in the minds of designers and programmers, and the change of development culture there are a number of associated technical problems.

The goal of this master s thesis project was to look at several of these issues and produce a series of documents that would help the company, EASAMS Ltd. to deal with them. In particular, there was a need to gain an initial know-how in Booch s Object Oriented Design methodology which was the development methodology chosen for the company s future software projects. The associated issues of evaluating code reuse and use of CASE tools were also considered.

The design of the IFES Passenger Application produced as part of the thesis project proved that the object oriented approach makes it easier to comprehend a complex system: it was possible to develop by one person a design, albeit only on the level of class declarations, corresponding to approximately 15-20 thousand lines of code. The other recognised benefits of the BOOD approach are a better understanding of system functionalities and a more general application design with increased possibilities of code reuse.

There are two main results of this project that should be beneficial for EASAMS Ltd. Firstly, the example design documentation aids in understanding the BOOD development process. It complements the main reference book which lacks information about the practical aspects of the method s application and use. If desired, the design can also serve as an inspiration for the real IFES development. Secondly, the evaluation of the CASE tools gives guidelines which can be used to choose a tool to support the software development.

Personally the work on the thesis project was quite enjoyable and will certainly bring benefits to its author as well. The main ones are the knowledge of Booch s progressive methodology which is gaining industry-wide support, and a practical insight into a well managed, quality oriented software development process. It is felt that these benefits would be very valuable for any seriously minded software engineer.

Considering the work done within this project and its outcomes from a wider perspective, there are several outstanding issues which could be pursued further. The reuse evaluation could be made more precise if a more detailed BOOD design was developed, implementing the full functionality of both Passenger and Cabin Crew applications.

The Rational Rose CASE tool was used only in its demonstration version and in a single-user development environment. Its capabilities would be better evaluated using a full version for a team project. It would be helpful to assess other tools as well to achieve a more objective view on their benefits.

To conclude, the project met the objectives set at its beginning. The question that remains open is to what extent its results will be used by the EASAMS Ltd. management and team members. The transition towards a full object oriented design development is certainly possible and it is believed that this project would provide help in its implementation.

List of References

Bibliography

The following table gives an alphabetically sorted list of the books and articles referenced in the dissertation. Most of the references were directly used during the thesis project work.

[Boo94]	Booch G. (1994), Object-oriented analysis and design with applications (2nd edition): Benjamin/Cummings. ISBN 0-8053-5340-2
[BS5750]	BS 5750 part 13 (ISO 9000-3): British Standards Institute 1991
[Gra93]	McGrath, F. (1993), Checklist for Buyers of ICASE Products , IEEE Software, November 1993, pp.108 110
[Hum89]	Humprey, W. (1989), Managing the Software Process : Addison-Wesley 1989
[Joh88]	R.E.Johnson, B.Foote (1988), Designing Reusable Classes , Journal of Object Oriented Programming, June/July 1988, pp.22 35
[Kel94]	Kelly, P. (1994), CASE Workbook : lecture notes, Department of Computing, University of Northumbria
[Kem92]	Kemerer, C.F. (1992), How the Learning Curve Affects CASE Tool Adoption , IEEE Software, May 1992, pp.23 28
[Mey88]	Meyer, B. (1988), Object-oriented Software Construction : Prentice Hall International
[Pau93]	Paulk, M, et al. (1993), Capability Maturity Model for Software, Version 1.1 : Software Engineering Institute, Carnegie Mellon University
[Pre92]	Pressman, R. S. (1992), Software Engineering: A Practitioner s Approach (3rd edition): McGraw-Hill. ISBN 0-07-707936-1
[Rum91]	Rumbaugh, J. et al. (1991), Object-oriented analysis and design : Prentice-Hall. ISBN 0-13-629841-9

Applicable Technical Documentation

The table below lists all relevant technical documents produced by EASAMS Ltd. or GEC-Marconi In-Flight Systems Ltd. which were used for the thesis project work. The documents are copyrighted by the respective companies and are not publicly available.

[Airline A] Marketing Specification for Passenger Application
Application Requirement Specification For The GMIS 2700IK Inflight System
Video Channel Map Description (technical note)
GMIS Network API for the Cabin System Control Panel and Attendant Workstations
Software Design and Control Procedures
System Software Specification Annex F: SPM Application Programming Interface
[Airline A] In-Flight Entertainment Applications: Level 2 Design
In-Flight Entertainment Applications: Software Development Framework Document

The following internal documentation was produced for EASAMS Ltd. during the project work.

[Airline B] IFE Applications: Passenger Application BOOD Analysis
[Airline B] IFE Applications: Passenger Application BOOD Architectural Design
[Airline B] IFE Applications: Passenger Application Level 2 BOOD Design
IFE Applications: Reuse in Passenger Application BOOD Design

Project Diary

The table below shows the major milestones in the master s thesis project, the dates for which their completion was planned, and the dates when they were completed in reality.

Stage	**Date planned**	**Date completed**
Background reading	14. 7.	14. 7.
IFES analysis	31. 7.	4. 8.
Architectural design	10. 8.	18. 8.
Detailed design	30. 8.	5. 9.
Tools and Reuse evaluation	8. 9.	15. 9.
Dissertation document	22. 9.	27. 9.

Appendix A: Analysis Scenarios

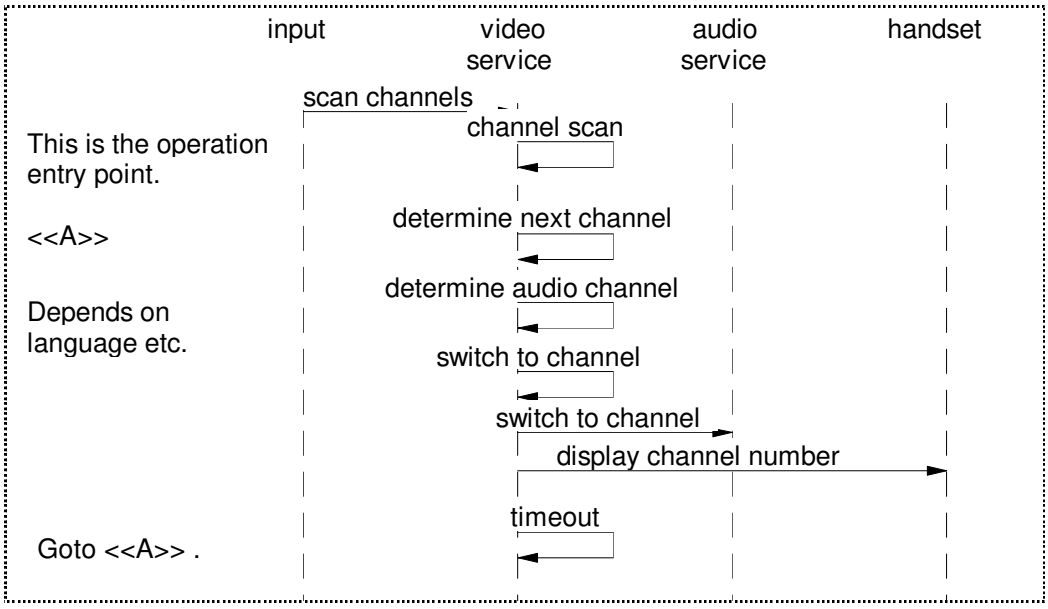
This appendix contains the scenario diagrams produced by the IFES Passenger Application system analysis which is described in chapter IFES Design. As is noted there, it was not feasible to include all the scenarios from the original design documentation; the diagrams presented here are selected examples. Refer to chapters BOOD Methodology and IFES Design for more information about the scenario diagrams.

The diagrams show how the key components of the system would interact to achieve the behaviour defined by the respective function points. At this stage (analysis) the names of the classes, objects and their operations used in the diagrams are in a free text form, as the main concern are the mechanisms rather than precise class interfaces.

The scenarios show logical rather than physical sources and handlers of events: for example in Scenario 4: Browsing Service Menus the Left , OK , etc. messages can come from the touch screen or handset key presses, and would be transformed by the appropriate button objects to these high-level events. Also for this reason the handset and the touch screen classes have been grouped into a single input object in most of the diagrams.

A.1 Scanning Video Channels

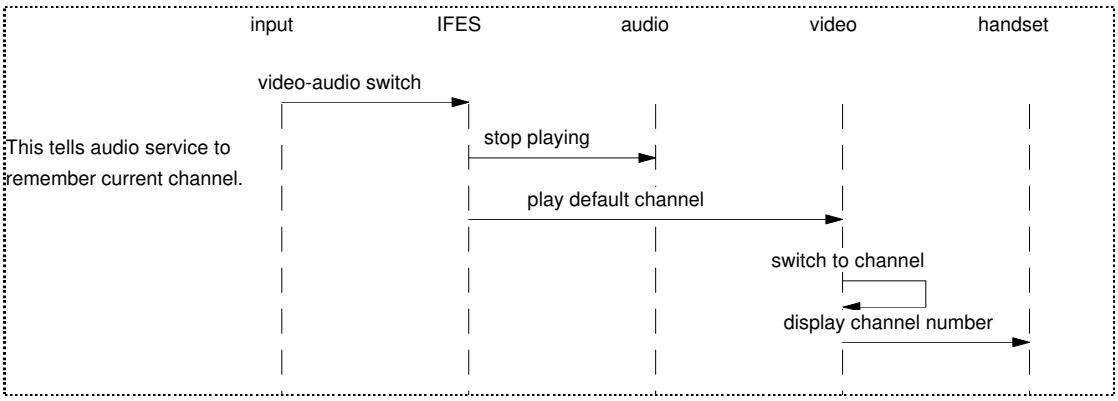
This scenario shows how the channel scanning after the handset Scan button press would be done.



Scenario 1: Scanning Video Channels

A.2 Switch between Audio and Video

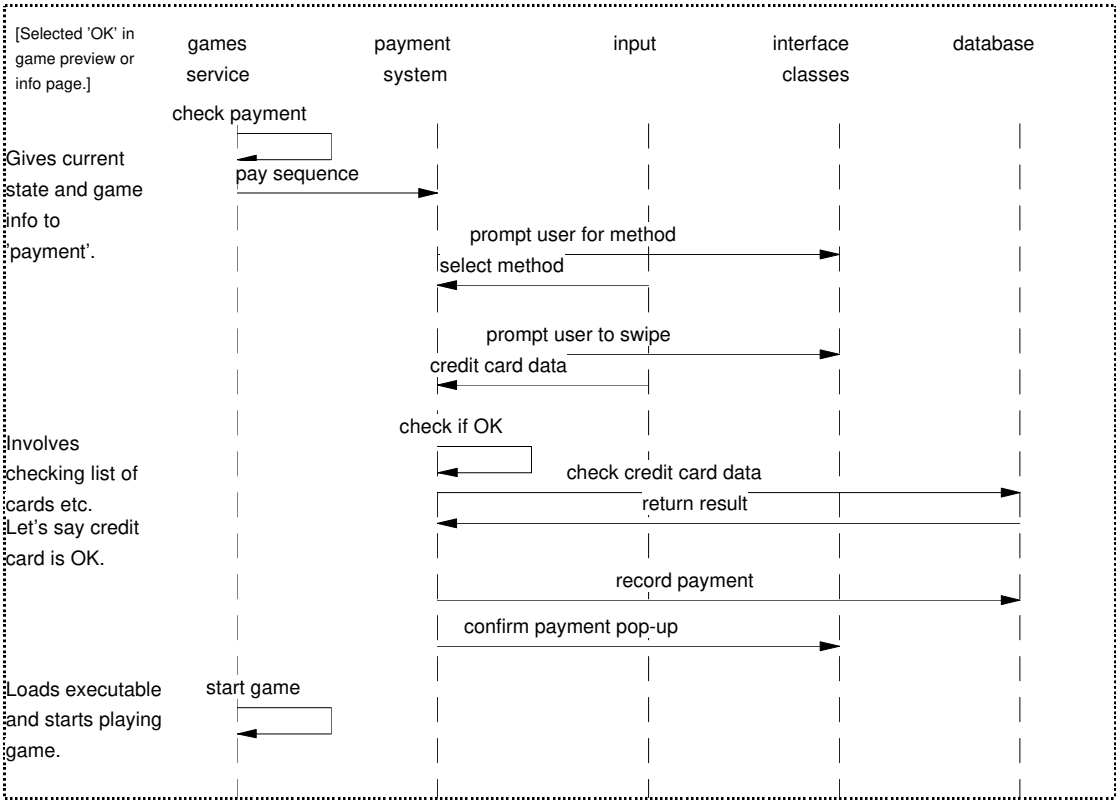
This scenario shows how the IFES would switch from the currently playing audio programme to the video programme played last.



Scenario 2: Switch between Audio and Video Programmes

A.3 Payment for a Selected Game

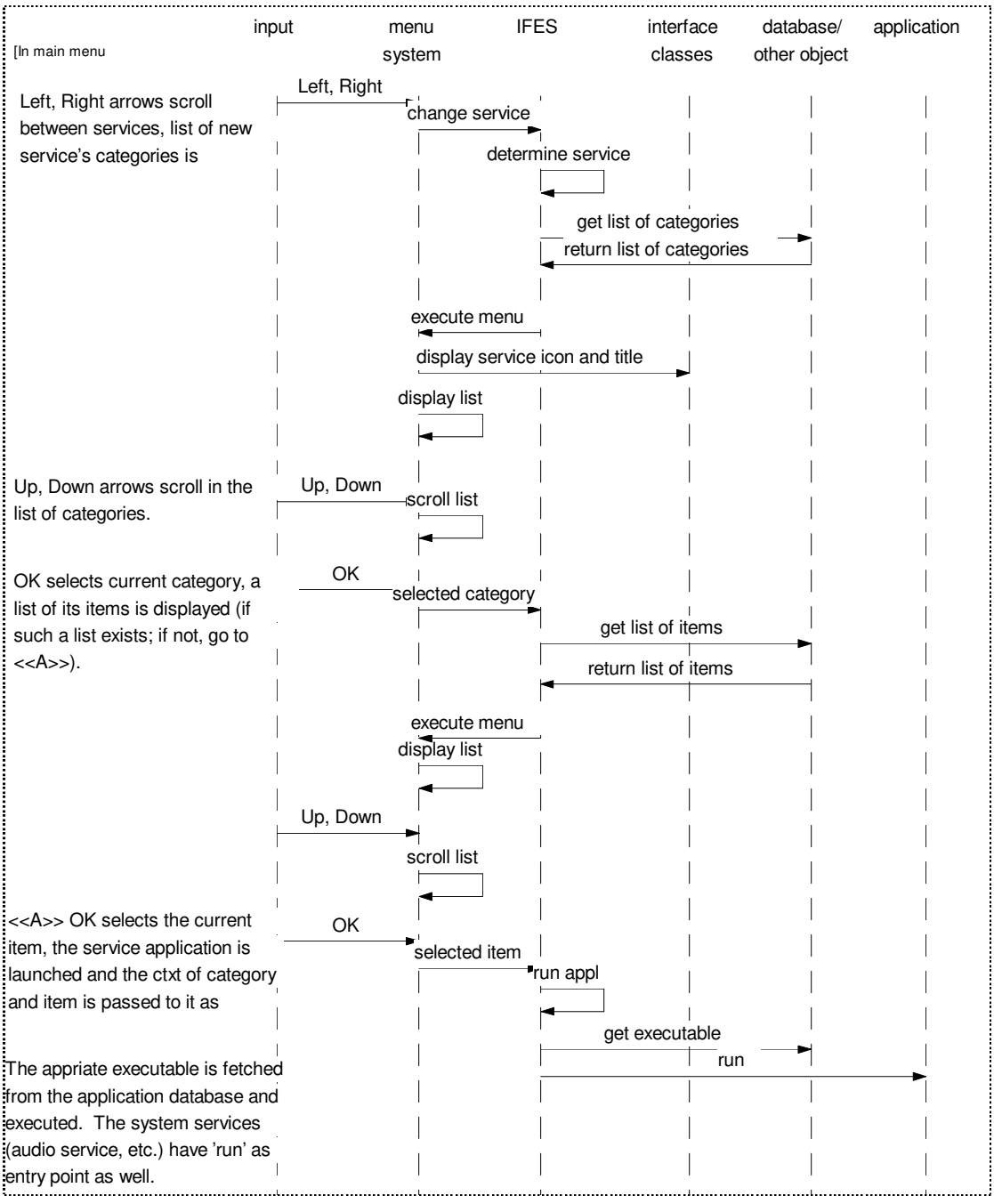
This scenario shows an outline of the payment actions after a purchasable game has been selected.



Scenario 3: Payment for the Selected Game

A.4 Browsing Service Menus

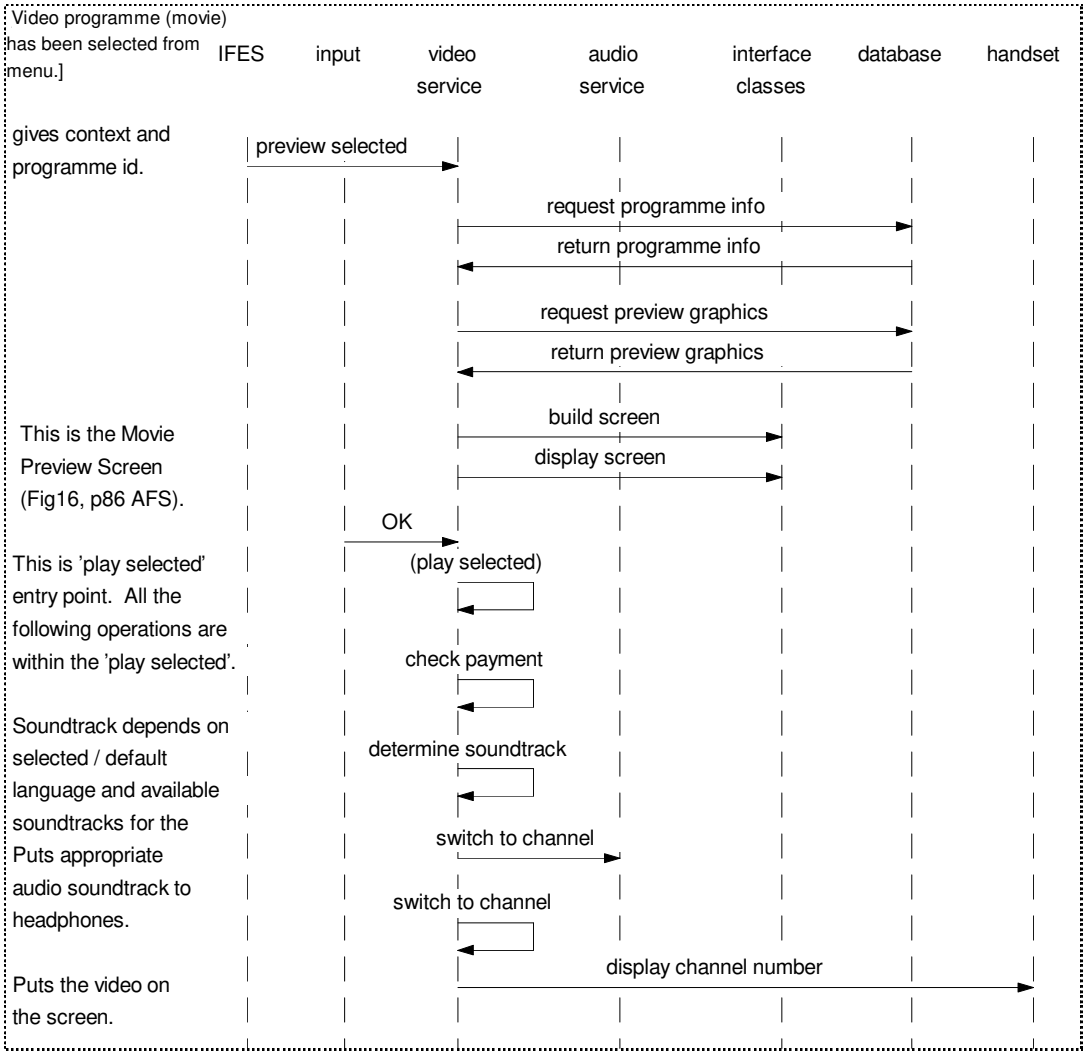
This scenario shows how the user interface and `CPassApp` objects cooperate when the user browses through the application menu contents.



Scenario 4: Browsing Service Menus

A.5 Playing the Selected Video Programme

This scenario shows the sequence of actions within the application after a video programme has been selected for watching from the menu.



Scenario 5: Playing the Selected Video Programme

Appendix B: Analysis Data Dictionary

This appendix contains a part of the IFES Passenger Application analysis data dictionary. It was not possible to include the whole data dictionary developed during the project work into this dissertation and the following table presents a sample of its analysis version. The classes which it includes are in some ways characteristic of the rest of the classes, and should be a good illustration of both the data dictionary format and contents, and of the system key abstractions.

The dictionary contains the following characteristics of each potential class:

- its name, at the analysis stage in a rather free-text form;
- a general characteristic of the class and a description of its responsibilities;
- a list of its emerging attributes and methods (they are not distinguished at this stage because that is largely an implementation decision made during class design);
- a lists of its collaborators which will be used during design to place the classes in the inheritance lattice and to clarify their run-time relationships;
- the method by which the class was identified within the system;
- general comments as appropriate.

Class	Description, Responsibilities	Operations, Attributes	Collaborators	Method	Notes
audio programme	kind of programme with associated channel	[programme +] channel			needs elaboration; analysis phase.
audio service	responsible for playing audio programmes, and video soundtracks mapped on audio channels.	[service application +] play [selected] audio (and display its screen) switch to [given] audio channel play default channel stop playing (store state) channel scan display programme graphics	audio controller, input, menu, graphic menu,	domain	
button	interface class, in dialogue boxes etc.	?	?	domain	

catalogue	maintains list of item descriptions and associated info	provide list of items provide first item provide last item provide number of items provide item of [given] number	catalogue shopping, menu, graphic menu, database,	domain	
catalogue item	provides information about the piece of goods	name description text price promotional graphics number on stock available options in given category (e.g. colours). shipping address read data	catalogue, graphic menu, menu system	discover y	
catalogue shopping	service running shopping choice and purchasing, dealing with payments	purchase sequence run (=application entry point) review purchases modify purchases delete purchases stop shopping (=pause for announcement)	input (?), catalogue, order, database, interface classes	domain	
context	identifies present state of the system	current service current category current item	IFES, help system	scenario	maybe invisib le in IFES.
currency	used in cash transactions, item price display	name symbol exchange rate	catalogue item, programme, payment system	domain	

database	provides necessary database access at the seat platform (configuration and application database blended together with file access)	return list of services return list of categories for a service return list of items in a category return item information (title, play time, ...) return item graphics return help pages text check credit card data record catalogue order record PVP tape order return list of ordered goods for a passenger record payment cancel order return application/game executable	order, catalogue, audio service, video service, information browser, IFES, payment	domain	
display control	controls LCD/CRT display output quailty	change brightness, ... get brightness, ... get attributes change attributes	overlay	discover y	
game application	the game executable with its service points	run (=entry point) demo pause resume	games service, menu, IFES	discover	
graphic menu	handles picture+text based item information browsing for audio, catal. etc.	determine next/previous item display menu item browse [with start item]	audio service, video service, games service, catalogue shopping	inventio n	
help system	ctxt sensitive text oriented pages ? kind of info browser ?	determine prev/next page show help pages [for current context]	audio service, video service, games service, catalogue shopping, menu,	domain	

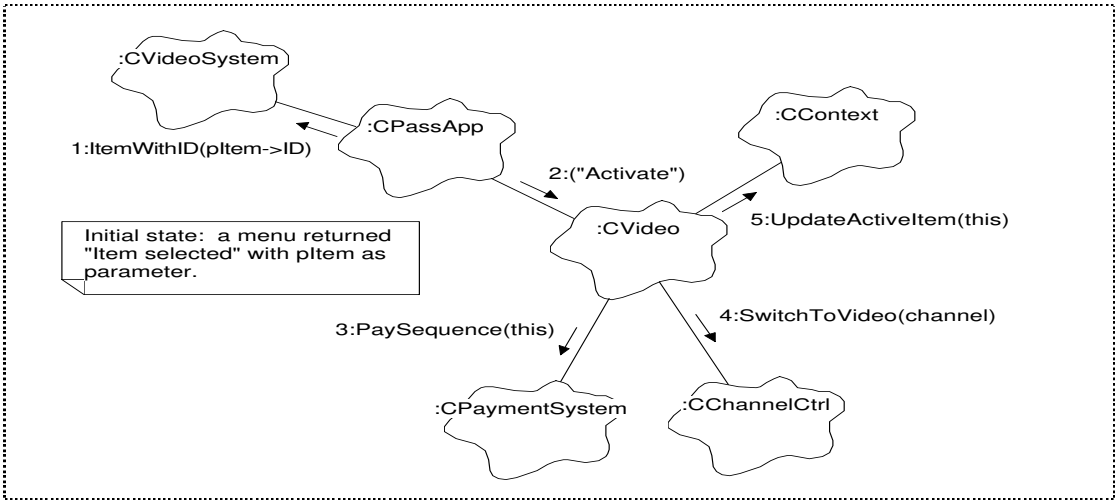
IFES	The whole system. runs appropriate services depending on menu choices, manages announcements and config.	make announcement determine next/prev service run application refresh screen startup (=entry point) shutdown	audio service,video service, catalogue shopping, games service, network messages, interface classes, database, menu, overlay	domain	
info browser	(hypertext?) browsing service	show first page show next page show previous page	IFES, menu, help system	domain	
payment system	deals with user payments, shared among services	pay sequence check credit card OK	database, audio service, video service, catalogue shopping, games service, interface classes	system	
programme	one piece of audio or video entertainment of specified duration; provides info about it. maybe kind of item	title description duration artists price (for watching etc.) graphics	<no direct collaborators, will be generic class>	invent	
video programme	kind of programme with associated soundtracks and video channel	[generic programme +] video channel number list of soundtrack languages+channels	video service, database		
video service	selection, viewing (mapping from title to channel), and payment for video programmes	[service application+] preview selected movie play selected movie play default channel switch to [given] channel stop playing channel scan determine audio channel for soundtrack	IFES, menu, audio service, help	domain	

Appendix C: Design Scenarios

This appendix contains the scenarios used during the IFES Passenger Application design to explore the functional mechanisms and class implementations. Refer to the chapter IFES Design for more information and for the context of these scenarios.

C.1 Playing a Video Programme

This scenario shows how the video programme selected from the menu is played. The scenario is sufficiently simple to be described by an object diagram.



Scenario 6: Playing a Video Programme

Comments: `thePassApp` checks with `theVideoSystem` if the selected item is currently available on one of the video channels. The scenario assumes this is true, and proceeds by activating the video programme object returned. The `PaySequence()` method is not invoked in case the programme is free, complimentary, or has already been paid for; the `CVideo` object carries all the necessary information to check this (see the class specification).

C.2 Switching between Video and Audio

This scenario is based on the co-operation with the hardware platform which performs the actual switch from video to audio channel output. A message is then sent to the application informing it about the change; the following pseudocode shows how the necessary consequent checks could be performed within `thePassApp` object.

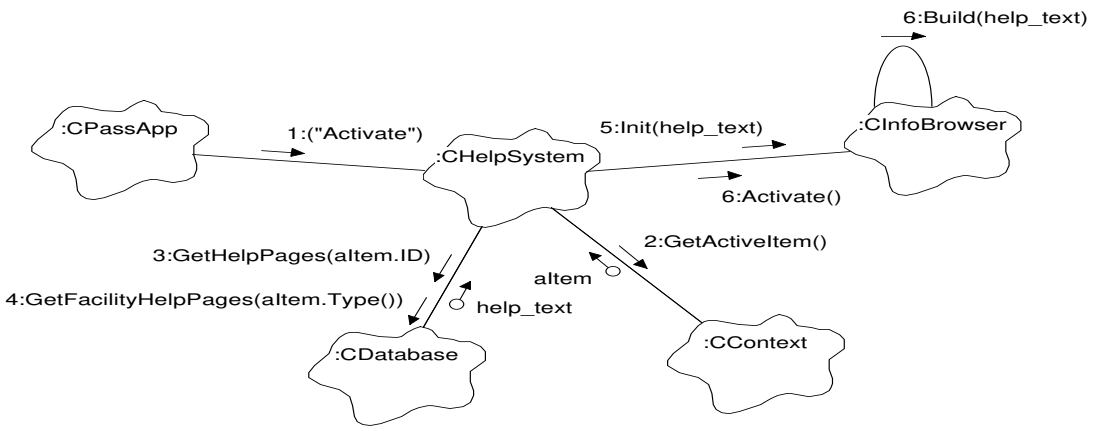
```
result = theContext.ActiveItem(programmePlayingNow);
// 'result  should be SUCCESS and the parameter return value valid
because the
// switch can only occur when in audio or video mode.  the returned
programme
// playing now is in fact the one before the switch occured.
```

```
// we need to get the programme item that is to be played, it will be
// of  the other  type than the one before the a/v switch.
if ( progPlayingNow.Type() == video )
then
  aPayItem = (CPayItem&) theAudioSystem.CurrentItem();
  // this is the audio that has been switched to because
  // it was the last one playing;  theAudioSystem can either take
  // the value from its attributes directly or ask the CChannelCtrl
about the
  // current channel and [safely] return the associated programme
item.
else
  aPaytItem = (CPayItem&) theVideoSystem.CurrentItem();
end if;

// The checks should be the same for both audio and video items,
therefore
// we can use CPayItem and assign it the proper value.
// Purchase is the main reason why playing the programme could be
abandoned.

if ( aPayItem.PurchaseNeeded() )
then
  // item has to be purchased before it can be activated
  result = aPayItem.Purchase();
else
  result = SUCCESS;
end if;

// The programme is either free/complimentary/purchased, or cannot be
played
if ( result == SUCCESS )
then
  generate  Stop  event for the  programmePlayingNow
  // the now abandoned programme needs to be told that the hardware
does
  // not play it any more.
  generate  Activate  event for the  aPayItem  object
  // processing will update both context and appropriate CXxxSystem
objects.
else
  generate  Activate  event for the  programmePlayingNow  object
  // this will switch back to the previous programme because the new
one
  // cannot be played, effectively overriding the hardware action.
end if;
```

Scenario 7: Switching between Video and Audio Programmes

C.3 Activating the Context Help

The starting state is in a list or graphic menu with a video item selected. The outline of actions during help activation are shown in the following object diagram:



Scenario 8: Activating the Context Help

C.4 Going to a Previous Menu Level

This is a secondary scenario that complements the exploration of the menu functions set up in the appropriate scenario (not included in this thesis).

1. The Back button is selected; the appropriate `CButton` object generates a Back button pressed event for `thePassApp`.
2. `thePassApp` processes the event

```
if ( menuSelections.Size() > 1 ) then

  // the current menu is not the top level so backtracking is
possible
  menuSelections.Pop();           // removes the current level from
the stack
  pItem = menuSelections.Top();   // gets the current item for
further checks

  if (pItem->Level() does not match the active menu type) then
    // e.g. the new item is a category which has a list menu
associated,
    // but the current menu is the graphic menu
    generate  Close  event for the current menu;
    generate  Activate  event for the menu to become the current
one;
  end if

  theMenuContents.Free();         // destroys the current list of
menu items
  BuildMenu( pItem->ID() ); // and builds the one for the level up
  generate  Update  event for the current menu with refence to
theMenuContents
    as parameter;

else
  // nothing happens
end if
```

3. the appropriate `CMenu` object processes the Update event

```
contents = (CRing<CItem&>&) aEvent.param;
current = contents.First();
Draw();
```

Scenario 9: Going to a Previous Menu Level

Appendix D: Level 2 Class Specifications

This appendix lists the source code of the IFES Passenger Application class declarations developed in the Level 2 desing. The project work involved specification of all classes related to the video functionality; only the important ones were selected for the purpose of this thesis.

The base source code was produced by the Domain CASE tool from the design specification. It was then formatted and completed by adding vital information not included by the automatic generation, e.g. specification of inheritance.

The class declarations that follow are grouped by the class categories in the order in which these are given in the IFES Design. See this chapter for more descriptions and explanations of the class functionality.

D.1 PassApp

```
//
//
// CLASS  : CPassApp
//
//
/*
RESPONSIBILITIES :
Passenger (seat-box) application driver
*/

class CPassApp{
public:

    /* power up initialisations, among others network and event
registration and
       displaying the introductory screens.
       returns Success if initialisation is OK, Failure if any
problem occurs*/
    TResult StartUp();
    /* performs clean up actions on application shut down*/
    TResult ShutDown();
    /* processes MS-Windows event messages sent to the object*/
    EventHandler();
    /* the main programme, starts the passenger application*/
    WinMain();
    /* system context*/
    Ccontext theContext;

protected:
    /* builds the list of service menu items from database values*/
    TResult BuildServiceMenu();
    /* builds the list of list/graphic menu items that form the
contents
       of the menu with ID given by the parameter.  This mechanism
allows
       building nested menu structures.*/
    TResult BuildMenu(TItemID);
```

```
    /* sequence of items selected via menus during the passenger
interactions,
       is used for backtracking by the Back button.*/
    CStack<CItem&> menuSelections;
    /* list of items that are contents of the service menu; is built
by the
       BuildServiceMenu method.*/
    CRing<CServiceItem&>;
    /* list of items of the current list or graphic menu; is built by
the
       BuildMenu method.*/
    CRing<CItem&>;

private:
    /* creates and inits the given topic s item.  the basic
information is taken
       from the parameter values, and the proper class instance is
created
       based on the  type  value.  the created item is told to
initialise the
       rest of its data from the database.
       returns a reference to the created item.*/
    CItem& CreateItem(TItemData*);
    /* creates and initialises all the screen objects it owns (menus,
buttons)*/
    TResult StartUpGUI();

    /* passenger application window; the class will also contain
other user inter-
       face objects like the Back and Off buttons.*/
    CWindow myWindow;

};
```

D.2 Item Collections

```
//
//
// CLASS  : CItemCollection
//
//
/*
RESPONSIBILITIES :
base class for classes collecting purchased items or listing all
available channel programs;
an abstract class
*/

class CItemCollection{
public:
    /* initialisation, redefined in subclasses for specific
functionality*/
    virtual TResult Init() = 0;
    /* return next item in collection, 'next' determined from
previous
       manipulations*/
    CItem& NextItem();
    /* returns previous item from collection, 'previous' determined
from
       manipulations before this call*/
    CItem& PrevItem();
    /* returns reference to the list of items held*/
    CList<CItem&>& ItemsList();
    /* returns TRUE if item with given ID is in the system's
collection, FALSE
       otherwise*/
    Boolean IsItemAvailable(TItemID);
    /* returns item of the given ID from the collection*/
    CItem& ItemWithID(TItemID);
    /* returns reference to the item which was last
```

```
accessed/manipulated*/
    CItem& CurrentItem();
    /* class interface template for updating the collection
contents*/
    virtual TResult Update() = 0;
protected:
    CList<CItem&> itemList;
    CItem& current;
};

//
//
// CLASS  : CVideoSystem
//
//
/*
RESPONSIBILITIES :
maintains list of currently played video programmes on all channels
*/

class CVideoSystem : public CItemCollection {
public:
    /* redefined to build the list of video programmes from database
information*/
    virtual TResult Init();
    /* rebuilds the list based on a new enquiry to the database*/
    virtual TResult Update();
    /* starts video channel scan*/
    void Scan();
private:
    /* switches to next channel in scanning*/
    void ScanNext();
};


//
//
// CLASS  : CPaymentSystem
//
//
/*
RESPONSIBILITIES :
provides a uniform interface for purchase support operations to all
classes that need it.
*/

class CPaymentSystem : CItemCollection {
public:
    /* runs purchase sequence for the given item.  interacts with
user interface
       objects and devices as necessary.*/
    TResult PurchaseSeq(CPayItem&);
    /* redefined, initialises the list*/
    virtual TResult Init();
    /* redefined update to ???*/
    virtual TResult Update();
};
```

D.3 Browsers

```
//
//
// CLASS  : CInfoBrowser
//
//
/*
RESPONSIBILITIES :
defines data and operations that allow browsing info pages which are
described by RTF token lists.  With respect to information browser
```

```
items it is basically in the position of a device that carries the
item s function.
*/

class CInfoBrowser{

public:
    /* displays the next page of text; returns Fail if on it
already*/
    TResult PageNext();
    /* displays previous page of text, returns Fail if on it
already*/
    TResult PagePrev();
    /* called at the end of browsing session, releases the list of
RTF strings*/
    TResult Close();
    /* displays the info pages text*/
    TResult Activate();
    /* displays current info page*/
    void Draw();
    /* redefined; reads text blob of RTF tokens (determined by the
given ID) from
       database and converts it to RTF strings.  sets first string as
current*/
    TResult Init(TItemID);
    /* specific to CInfoBrowser; blob of text which contains RTF
tokens
       is given as parameter.  sets first string as current.*/
    TResult Init(char*);
protected:
    Boolean fullscreen; // DEFAULT VALUE : FALSE
    CList<CTokenRTF> tokens;
private:
    int current;
    /* extracts the RTF token strings from the raw text*/
    CList<CTokenRTF> BuildRTF(char*);
    /* the window which displays the text*/
    CWindow* myWindow;
};
```

D.4 ServiceItems

```
//
//
// CLASS  : CItem
//
//
/*
RESPONSIBILITIES :
contains basic information about a service topic, and defines the
interface for all items to perform the function associated with them.
*/

class CItem : public CListItem {

public:
    /* activates the item s function;
       empty skeleton to be redefined in each child class as needed,
provides
       common interface that can be used in CPassApp for polymorphic
calls*/
    virtual TResult Activate() = {};
    /* reads item's data from the database; to be redefined in
subclasses
       to read all the data required.  in order to reduce network
traffic and
       database access, this method should be only redefined in the
leaf classes
       to read the complete data structure for the item*/
```

```
    virtual TResult Init();
    /* indicates whether the item is performing its function at the
moment;
       returns value of the attribute*/
    Boolean Active();

protected:
    char[] name;
    char* picture;
    char* description;
    TItemType type;
    TLevel level;
    TItemID ID;
    CContext* theContext;
    Boolean active;
};

//
//
// CLASS  : CPayItem
//
//
/*
RESPONSIBILITIES :
defines the data and operations for all items that can be purchased.
an abstract class.
*/

class CPayItem : public CItem {

public:
    /* perform the item purchase via CPaymentSystem*/
    virtual TResult Purchase() = 0;
    /* returns TRUE if a purchase is needed before it can be
activated,
       FALSE if the item is free, made complimentary, or has been
paid for*/
    Boolean PurchaseNeeded();
    /* Registers that it has been purchased*/
    void RegisterPurchase(TPaymentID);

protected:
    float price;
    Bool charged; // DEFAULT VALUE : FALSE
    Bool paid; // DEFAULT VALUE : FALSE
    TPaymentID paymentID;
};

//
//
// CLASS  : CProgramme
//
//
/*
RESPONSIBILITIES :
defines data and operations common for audio and video programmes;
an abstract class.
*/

class CProgramme : public CPayItem{
protected:
    int channel;  // logical channel number
    TTime startTime;
    TTime duration;
};

//
//
// CLASS  : CVideo
//
```

```
//
/*
RESPONSIBILITIES :
defines data and operations of a video programme item that can show
itself on the screen
*/

class CVideo : public CProgramme {
public:
    /* redefined method, starts playing the video and informs the
context*/
    TResult Activate();
    /* processes event messages sent to the object*/
     EventHandler();
    /* gets its data from the database */
    virtual TResult Init();
protected:
    TSoundtrack[] soundtracks;      // array of all available
language versions of
                                // the video;  channel  is the current
one
};

//
//
// CLASS  : CAudio
//
//
/*
RESPONSIBILITIES :
defines data and functionality of an audio programme with the
associated audio highlights
*/

class CAudio : public CProgramme {
public:
    /* redefined to start playing the programme and display the audio
highlights*/
    virtual TResult Activate();
    /* gets its data from the database, instantiates and/or
initialises its info
       browser object*/
    virtual TResult Init();
protected:
    TItemID infoPages;  // ID of the associated info page sequence;
used by
    audioHighlights* CBrowserItem;  // this item for initialisation
};
```

D.5 MenuSystem

```
//
//
// CLASS  : CMenu
//
//
/*
RESPONSIBILITIES :
establishes generic  menu functionality,
the services, list, graphic and purchase menu are all derived from
this class.
an abstract class
*/

class CMenu : public CView {
public:
    /* initialises the value of list of items from the parameter*/
    TResult Init(CList<CItem> *);
```

```
    /* draws the menu on screen; pure virtual*/
    virtual void Draw() = 0;
protected:
    CList<CItem>* contents;
    CItem& current;
    CButton* buttonPreviousItem;
    CButton* buttonNextItem;
};
```

D.6 Database

```
//
//
// CLASS  : CDatabase
//
//
/*
RESPONSIBILITIES :
provides interface to the application database and access operations
that support higher-level class operations.
*/

class CDatabase{
public:
    /* checks with the application database if given credit card data
are valid
       for purchase and order operations
       returns Success if the data is valid, Failure otherwise*/
    TResult CheckCreditCard(TCreditCardData*);
    /* returns (via parameter) all data pertinent to a video service
item
       determined by the given ID.  clients are responsible for
freeing the memory
       allocated to the data.
       returns Failure if the given ID has no data associated in the
database*/
    TResult GetVideoData(TItemID,TVideoData*);
    /* returns (via parameter) data pertinent to the audio service
item
      determined by the parameter ID.  it is caller's responsibility
to free the
      memory allocated for the data.
      returns Failure if the given ID has no data associated in the
database */
    TResult GetAudioData(TItemID,TAudioData*);
    /* commits the purchase of the given item to the database.
       if successful, the parameter value is set to the transaction
ID and the
       return value is Success,
       otherwise returns Failure and the parameter value is
undefined*/
    TResult RegisterPurchase(CPayItem&);
    /* returns (via parameter) list of data structures with
information about
       all services as used with the service menu.  clients are
responsible for
       freeing the list.*/
    TResult GetServiceData(CList<TServiceData>* );
    /* returns (via parameter) list of items that constitute the menu
of given ID.
       the first item is the default one.  clients are responsible
for freeing the
       list.*/
    TResult GetMenuContents(TItemID,CList<CItem> *);
    /* returns (via parameter) the list of audio data for audio
programmes
       currently playing on all channels. clients are responsible for
freeing
       the list.*/
```

```
    TResult GetCurrentAudios(CList<TAudioData> *);
    /* returns (via parameter) list of video data structures for all
video
       programmes currently playing on all channels. clients are
responsible for
       freeing the list.*/
    TResult GetCurrentVideos(CList<TVideoData> *);
    /* initialises the class instance--connects to the application
database (?)
       returns Failure if the initialisation was unsuccessful */
    TResult Init();
    /* returns (via parameter) text of help pages for item of the
given ID.
       clients are responsible for freeing the memory allocated for
the text*/
    TResult GetHelpText(TItemID,char*);
    /* returns (via parameter) text of help pages for the IFES
Passenger
       application, top level.
       clients are responsible for freeing the memory allocated for
the text */
    TResult GetSystemHelpText(char*);
    /* returns via parameter the help text of the given facility
(video etc).
       clients are responsible for freeing the memory allocated for
the text */
    TResult GetFacilityHelpText(TFacility, char*);

private:
    /* the database cache is contained here*/
    CCacheManager cache;
};


//
//
// CLASS  : CCacheManager
//
//
/*
RESPONSIBILITIES :
caches raw data for the database.  is not used by or visible to any
other part of the system.
*/

class CCacheManager{

public:
    /* returns TRUE if data with the given ID are stored in the
cache,
       FALSE otherwise*/
    Boolean CheckCache(TItemID);
    /* stores the given data (ID, size, data) in the cache*/
    TResult AddData(TItemID,unsigned int,BYTE*);
    /* removes data for the given item from the cache pool and frees
the memory
       allocated to the data structure
       returns Success if operation was OK,
       Failure if the item of given ID was not in the cache*/
    TResult RemoveItem(TItemID);
    /* removes some items from the pool using a LRU algorithm, frees
the memory
       allocated for them.  will need an age threshold from
somewhere*/
    TResult Sweep();
    /* initialises the cache pool, allocates memory for it (parameter
gives pool
       size, should have a default value TBD)*/
    TResult Init(int);
    /* closes the cache manager, frees the cached data and the memory
pool*/
```

```
    TResult Close();

private:
    CArray<TCacheItem> contents;
};
```

D.7 Devices

```
//
//
// CLASS  : CHandset
//
//
/*
RESPONSIBILITIES :
reads credit card data, displays number on the handset display
*/

class CHandset{
public:
    /* displays given [channel] number on the handset 7 segment
panel*/
    TResult DisplayNumber(int);
    /* enables reading credit card data*/
    void EnableCardSwipe();
    /* disables reading credit card data*/
    void DisableCardSwipe();
    /* returns value of the attribute*/
    Boolean IsSwipeEnabled();
    /* initialises the handset data*/
    TResult Init();
protected:
    int numberShown;
    Boolean swipeEnabled;
};

//
//
// CLASS  : CChannelCtrl
//
//
/*
RESPONSIBILITIES :
controls switching to given audio and video channels
*/

class CChannelCtrl{
public:
    /* switches to given channel*/
     SwitchToAudio(number);
    /* switches video output to given channel*/
     SwitchToVideo(number, window_coordinates);
    /* cycles through video channels*/
     ScanVideo();
    /* returns default logical audio channel (taken from HW)*/
    int DefaultAudio();
    /* returns default logical video channel number (taken from HW)*/
    int DefaultVideo();
protected:
    TVideoChanMap mapVideo;
    TAudioChanMap mapAudio;
};
```