Inspection of OO Software with Incomplete Documentation Using A Document Driven Approach

By
Hongying Shi, B.Eng.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2004) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
Inspection of OO Software with Incomplete Documentation Using A Document Driven Approach

AUTHOR: Hongying Shi, B.Eng.
(Huazhong University of Science and Technology, China)

SUPERVISORS: Dr. David L. Parnas and Dr. Alan Wassyng

NUMBER OF PAGES: xi, 156

Abstract

Critical software requires formal and rigorous inspection to achieve the required quality, and good documentation provides a solid basis for inspection. The *Document Driven Inspection* approach takes advantage of precise and complete documentation to serve as a mathematically rigorous and effective technique to review software in a disciplined way. However, it is often the case that precise and complete documents are not available to the inspectors. In these cases, the *Document Driven Inspection* approach is still useful as illustrated by our case study. As far as we are aware, this is the first application of this approach to an object-oriented critical software system.

In this thesis, we investigate several state-of-the-art techniques in software inspection, and also some new techniques that focus on inspection of object-oriented design and code. The *Document Driven Inspection* approach proposed by Parnas is introduced and further analyzed as applied to an object-oriented design, especially those with incomplete documentation. We illustrate the application of this approach as applied to an object-oriented case study, by producing the complete and rigorous documents that can then act as a basis for further inspection. These documents also illustrate how to document an object-oriented design using tabular expressions.

Acknowledgements

First of all, I would like to express my sincere gratitude to Dr. David Lorge Parnas, and Dr. Alan Wassyng, my co-supervisors, for their support and encouragement, invaluable guidance and suggestions throughout my graduate study in McMaster. Without them, this work would not have become possible. I am grateful to Dr. Wassyng, for him being so nice to accept me half way during my thesis writing, and for him putting so much time, efforts and wonderful ideas to help me finish this work.

I would like to thank Dr. Emil Sekerinski and Dr. Spencer Smith for being on my thesis committee and for their valuable comments on my thesis.

Thanks to Materials and Manufacturing Ontario and MD Robotics for the partial financial support of this work. Sincere appreciation to Michael Schmidt and Joshua Richmond from MD Robotics, for them putting so much time and advice on the case study. I enjoyed the enlightening communication with them.

This thesis is a special present to my lovely son Nicholas (dangdang) and my dear husband Zhiyong. Nicholas was born during my graduate study period, and he has brought me and my family endless joyfulness and happiness. My husband Zhiyong is the most important person in my life who is always there with me, with his love, supports, understanding, and strengths.

Last but not least, I would like to thank my parents. They are always my backbone. They also took good care of me and dangdang during his early months.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Computer systems are playing an increasing role in applications where a failure or malfunction could lead to significant financial losses or even serious injury. Software in such applications, e.g. banking transactions, aircraft control or nuclear power station shutdown systems. is called *Critical Software*.

Safety-critical software is that subset of critical software in which safety is the key, and an error in the safety-critical software system could result in loss of life. *Mission-critical software* usually costs a large amount of money to build, and functions in a critical role to fulfill the ultimate mission of a system. Any failure in that software may cause a disaster, for instance, the crash of a banking system.

Unfortunately, we continue to hear about serious accidents caused by software failure. As an example, in June 2004, Canada's largest bank, Royal Bank of Canada (RBC), had to cope with one of its worst computer nightmares. Ten Million RBC customers were not sure of their account balances and had no proof of payroll deposits.

Customers' cash, in the form of electronic pulses, had been trapped for days in RBC's computer system, which was hit by a programming error and was late in allocating transactions to individual accounts. As explained by RBC staff, fixing the error cost a day of transaction processing time, which resulted in the incident [23].

There is no doubt that the quality of critical software is extremely important. Redundancy is a classical engineering solution used in the design of critical software systems to safeguard the systems against random hardware faults. The use of formal methods in software on the other hand, is a rigorous approach to building reliable software applications. It is used all too rarely in practice. More work needs to be done in this field.

Software inspection has been regarded as an efficient and effective way to control the quality of software. For a critical software system, inspection serves as an essential and formal tool to verify and validate the quality of software, and good documentation provides a solid basis for inspection.

Based on the *Divide and Conquer* principle [40], and influenced by Harlan Mills's *Program Function* theory [27], a *Document Driven Inspection* (DDI) approach using *tabular expressions* [33][35][37] was proposed by Parnas to formally review the software for defect detection and removal.

The *Document Driven Inspection* approach has been demonstrated to be effective in industry through the Darlington Shutdown System project [33][36][37][51] by Ontario Hydro (currently Ontario Power Generation Inc. - OPG) for the inspection of a safety-critical program for the Darlington Nuclear Power Generating Station in Ontario, Canada. The Software Cost Reduction (SCR) project [3] by the U.S. Naval Research Laboratory for the A-7E aircraft also gave birth to the idea of *Active Design Reviews* [41], which is one of the core techniques in DDI. However, to the best of the

author's knowledge, there is no previous published application of Parnas' *Document Driven Inspection* to object-oriented (OO) designs.

OO is a commonly accepted paradigm and extensive applications and technology have been developed using it. Compared to the *procedural paradigm*, the *OO paradigm* promised many benefits, for instance, the increased reusability of code. Since Parnas' approach originated before the appearance of the OO paradigm, and was based on applications developed on older systems using procedural programming methodologies, new issues may arise when it is applied to OO systems. Modifications or improvements of the specific application of our approach are always interesting, but actually become necessary when we apply this approach to OO designs, as shown in Section 3.2.

The Document Driven Inspection approach is a technique that relies a lot on rigorous and systematic documentation. If the requirements and design documents from the customer are complete and precise, the Document Driven Inspection approach is a good way to perform an "active review" of those documents, and to compare the design against its requirements to see if there is any discrepancy between them. However, in cases where either the requirements document or the design document is not complete or not even available, we can still use parts of this approach to examine the actual code to produce a precise document that reflects the design of the application. The document is complete in the sense that missing functions are explicitly highlighted and placeholders are used in local descriptions of the behavior. We can (often) also communicate with the customer to determine the actual requirements so that, eventually, the code and the design documents can be compared against their requirements. This approach provides a means of inspecting software in a systematic and mathematically rigorous way in cases in which the design and requirement

documents are not sufficient. In this case we still do the best that we can within the constraints imposed by not having complete information. Thus, the Document Driven Inspection approach is useful both when we have access to adequate requirements and design documents, and when we do not have access to such documents.

1.2 Scope

This thesis applies and illustrates a rigorous and effective approach for the inspection of object-oriented critical software. The *Document Driven Inspection* approach is first discussed and analyzed from an object-oriented aspect. The thesis further illustrates the rigor and completeness of the documentation produced through the Document Driven Inspection approach.

The case study described in this thesis was partially supported by Material and Manufacturing Ontario (MMO), an organization devoted to building industry and university partnerships, and MD Robotics (MDR), a company developing advanced robotics systems for space and terrestrial applications. The case study was conducted on code extracted from an existing MDR project.

Our initial objective was to perform an inspection on the case study code. However, the documents available were not complete and contained inaccuracies. They were simply not yet an adequate basis for an inspection. We then changed the scope of the case study, from simply performing an inspection, to producing the documentation that would facilitate an inspection.

When we started to do the case study, we did not realize the many differences between OO and non-OO systems from the inspector's point of view. Since the case study was performed on an extract from the code, and not the complete system, we

could not produce a module guide that would have given the reader an overview of the modular structure of the system. The numerous small methods and the complicated invocation relations among them also posed difficulties to us when we tried to document them. The *uses relation* [44] is messy, more like a network instead of a hierarchy. The reader has to investigate dozens of methods to understand any single method.

The major contribution of this work is that, it is the first time the Document Driven Inspection (DDI) approach has been modified to apply to an object-oriented software application, especially a system with lots of missing information. Given the fact that OO design and programming techniques are having significant influence in current software industry, it is of great practical value that we discuss our approach through an OO application. As we discussed previously, since the Document Driven Inspection was originated on a project written in a procedural language, complications were discovered when performing the inspection task on OO software. The object-oriented paradigm is generally perceived to provide several benefits, yet it also leads to some problems when we try to understand and inspect OO code. *Delocalization* [49], for instance, requires the inspector or maintainer to trace chains of method invocations through many classes, traversing both up and down the inheritance hierarchy [53]. Such a situation could be improved through good documentation of design decisions.

Furthermore, this thesis provides a series of sample documents based on a case study example, on how to precisely specify and document design decisions of OO software through *tabular expressions*. These include the *Module Interface Specification(MIS)* and *Module Internal Design(MID)* documents, which can be found in the appendix of this thesis. These examples can also serve as a starting point for generating an automated inspection system. Automation is important if we want to apply

our approach to large applications, because it will greatly improve the efficiency and at the same time preserve the correctness.

1.3 Overview

The remainder of this thesis is organized as follows:

Chapter 2 and Chapter 3 present a survey of contemporary research on critical software inspection, and on object-oriented design and code inspection technologies, respectively.

Chapter 4 introduces the basic concepts of tabular expressions and program function documentation, and then presents a Document Driven Inspection approach focusing on object-oriented aspects.

Chapter 5 and Chapter 6 illustrate our approach with a case study from MD Robotics. Chapter 5 introduces the background information of the whole system and its architecture and decomposition, then includes a more detailed discussion on the code extracted for use as our case study. Chapter 6 first analyzes the issues that arise in documenting the case study code, then presents the way in which we dealt with those issues. There is also a discussion on problems we discovered in the MDR documents.

Chapter 7 summarizes this thesis and makes suggestions for future work.

The appendices contain the example documents that can be used as a basis for inspecting the case study code. They are the MIS and MID documents for the ScriptGenerationManager class. As an illustration of the complicated *uses* relation of our case study module, the *uses* graph of ScriptGenerationManager class is in Appendix C.

Chapter 2

Survey of Inspection Techniques for Critical Software

This chapter introduces related work in current academic and industrial fields for critical software inspection. Michael Fagan and Tom Gilb's Inspection process, Harlan Mills's Cleanroon Technology, and Dave Parnas's Document Driven Inspection approach as applied to the Darlington Shutdown System project are briefly described.

2.1 Introduction

Gerald Weinberg introduced the *Egoless Programming* concept in 1971 in his first book *The Psychology of Computer Programming.* The idea is, "no matter how smart a programmer is, reviews will be beneficial" [25]. Now *reviews* have become an effective and efficient method in software quality assurance. The relationship of criticality to assurance is obvious: the more critical the software is, the more important the software assurance efforts are.

There are a variety of review techniques, among which *inspections* is a disciplined and powerful member. *Software Inspection* is defined to be a static analysis technique that often relies on visual examination of development products to detect defects, violations of development standards, and other problems [1]. Inspections on requirements specifications, designs, source code, and other work products, have been developed into a disciplined engineering practice for detecting defects and improving the quality and maintainability of software artifacts.

The following sections present a brief introduction to some of the principal software inspection techniques.

2.2 Fagan's Inspection

McConnell [25] wrote that *Fagan's Inspection* [11][12] is one of the 10 best influences on Software Engineering. *Fagan's Inspection* has now become almost synonymous with the term "Inspection". During his development career in IBM, early in the 1970s, Michael Fagan first described a software inspection process for the purpose of both improving the software quality and shortening the delivery time and cost. Interestingly enough, this idea was sparked by a method first employed in hardware engineering to examine designs after exhaustive testing.

As a software development manager, in the circumstances when software development was chaotic, Fagan's biggest challenge was to reduce the budget and deliver products on time [13]. Therefore, the key ideas of Fagan's Inspection at its inception were to focus on the management of the verification process, as well as on the definition of roles for inspection participants. A team consisting of the author of the document, a moderator, a recorder and a number of inspectors proceed to inspect the

document using a multi-stage process.

As industry accepted and began implementing his inspection process, Fagan extended and evolved his idea into the *Fagan Defect-Free Process* (FDP) [26], incorporating Formal Process Definition, and reinforcing the Continuous Process Improvement aspect of the Inspection Process.

Below is the description of these three components excerpted from [26].

> " *Formal Process Definition* is a method of defining the work process in terms that make it measurable and manageable by its users.
>
> *Fagan Inspection Process* is a seven-step process used to inspect the deliverables that have been created at the end of each phase of development to find defects.
>
> *Continuous Process Improvement* involves removing systemic defects from the work process as they are found by inspections or other operations in the life-cycle. "

Note that the *Continuous Process Improvement* stage is not defined to be removing defects from the product; it is removing them from the process.

Fagan emphasized his inspection process and improved it from the original five-step process to the current seven-step one, which comprises the following items:

Planning Choose participants and schedule meetings.

Overview Educate the inspection team by presenting the background and context of the target software.

Preparation Read documents, including a checklist of questions to aid in finding flaws.

Inspection Hold inspection meeting to identify and document defects.

Process Improvement Review the previous steps for improving future inspection, identify causes of defects and recommend process improvements.

Rework Correct all defects.

Follow-up Ensure all identified defects are corrected.

As we can see, the core idea of this new *Fagan Defect-Free Process* is still the inspection process and its management. It promotes criticizing *product* instead of *people*, and it defines *exit criteria* so that within a certain time the inspection can proceed to the next stage so that costs can be reduced and the development life-cycle can be shortened.

Admittedly, Fagan's inspection method helped improve and systematize the inspection process, and has been a great help in improving the quality of software since its wide spread adoption. Fagan's approach is satisfactory more from the management point of view. However, software inspections are not only a management issue. We would like to discuss this further in the following sections.

2.3 Gilb's Software Inspection

Since Fagan, many people have worked on the inspection process and techniques. Among these, Tom Gilb and Dorothy Graham provided probably the most comprehensive discussion of inspection available [17]. The book presents a detailed guideline on how to introduce and refine software inspection in an organization with enriched documents, including procedures and a variety of inspection forms. Experience data

from various sources were also included in their discussion of the benefits and costs of inspections, presenting people with a vivid picture of how important and effective a technique it is.

Gilb also emphasizes that the following two points must be clear to better understand Inspections [15]. First, inspection consists of two main processes: the *Defect Detection Process* (DDP) and the *Defect Prevention Process* (DPP). Second, a *major defect* is a defect that, if it is not dealt with at the requirements or design stage, will probably have an order-of-magnitude or larger cost to find and fix when it reaches the testing or operational stages.

Yet Gilb believes that [16], "it is more relevant to view Inspection as a way to control the economic aspects of software engineering, rather than a way to get 'quality' by early defect removal". He further states that inspections should be measuring that the engineering process is sound and that the "product" document at hand is consequently sound. He also says that management needs to establish a sound set of *Process Exit Conditions* before any document is allowed to be used by others in the organization. The approach Gilb advocates is to compute the number of the *probably remaining major defects*, where the level remaining is economically acceptable.

In spite of some different opinions between Fagan and Gilb on the specific steps during an inspection, they both focus on the process of an inspection, and address them more from the management point of view. This is probably one of the reasons why Fagan's and Gilb's approaches are more popular than many others in industry. As generally recognized, Fagan did a lot in helping develop a formal inspection process. However, it is actually independent of the use of formal notation and other aspects of "formal methods". "Formalizing" a process and a process that involves formal descriptions of programs are very different. A process that has not been formalized

can use formal descriptions.

2.4 Mills' Cleanroom Software Engineering

The theoretical foundations of *Cleanroom* [45] were established in the late 1970s and early 1980s, when Dr. Harlan Mills, an accomplished mathematician and IBM Fellow, related fundamental ideas in mathematics, statistics, and engineering to software. Influenced by Dijkstra on *structured programming* [6], Wirth on *stepwise refinement* [54], and Parnas on *modular design* [38], Mills defined scientific foundations for an engineering approach to software.

The name *Cleanroom* was taken from the electronics industry, where a physical cleanroom exists to prevent introduction of defects during hardware fabrication. We think that, one major difference of Cleanroom technology from other processes such as Fagan's inspection, is that *Cleanroom* is dedicated to defect prevention in the first place rather than defect removal afterwards.

Three fundamental principles capture the core of Cleanroom software engineering. They are, incremental development under statistical process control; stepwise refinement and verification; and statistical testing and software certification. Above these, *Cleanroom* added two main points that are the developers did not test and the testers used statistical methods of reliability prediction.

We regard *stepwise refinement and verification*, using *box-structured* systems [28] as the heart. As each step in a box structure design expands a previous box description into the next one, an immediate verification of correctness follows and in case it fails, it is easy to explore a new expansion. Mills proposed that the behavioral description of these boxes should be a function. This forms the solid foundation of Parnas's

approach, which will be introduced in the next section.

The following presents some principle ideas of Mills.

Incremental Development

Incremental development is based on the engineering principle of controlled iteration in product development. As described by Brooks [5], "Mills proposed that any software system should be grown by incremental development. That is, the system should be first be made to run, even if it does nothing useful except call the proper set of dummy subprograms. Then bit by bit, it should be fleshed out, with the subprograms in turn being developed - into actions or calls to empty stubs in the level below."

The idea of *incremental development* is similar to that of Wirth's *stepwise refinement*. Both these ideas were proposed as early as the 1970s. This development procedure is very similar to one aspect of the OOP approaches. First, classes with blank methods are built, with simple syntax to ensure the smooth compiling of the code, and then the detailed implementation of the methods are filled in one by one, and compiled one by one to make sure that each method is correct by itself. This way the maintainability of the code is improved greatly.

The value of statistics for management and control is also appreciated in Cleanroom techniques. As stated in [45], incremental development as practised in Cleanroom software engineering also provides a basis for statistical quality control. Based on user feedback, product quality can be measured at the end of each increment and is compared with the team's quality goals continuously.

Box Structures

Mills defined three different forms of boxes. Specifications begin with an external view called the *black box*. This is transformed into a state machine view called the *state box*, and then fully developed into a procedure view called the *clear box*.

Figure 2.1 taken from [45] depicts the three box structures that represent identical external behavior yet increasing internal visibility. A *black box* specifies the external behavior of a system or a system component. The refinement of a black box is the *state box*, which also shows the state data required to achieve the black box behavior. The *clear box* is a further refinement of a state box, specifying algorithm and procedure designs required to achieve the state box behavior. The beauty of these box structures is that each refinement is verified against the previous step. Hence, they are declared to be able to "separate three aspects of system development (specification of behavior, data, and procedures) yet relate them in a coherent process of refinement and verification" [45].

Box Structures [28] embraces an important software engineering principle of *data encapsulation*, and are developed in a stepwise *refinement* and *verification* process that integrates both system control and data operations. Stepwise refinement and verification in box structures, is built on the principles of data abstraction. We think this is a strong link to one of the essential characteristics of object-oriented programming. OOP also builds a system from sketch to the final design piece by piece, and the verification of software quality can also be done accordingly such that one can be confident about the quality of each level of the whole system hierarchy. If we try to relate *box structures* to object-orientation, a black box view represents an object's external behavior described using items external to the object only. A state

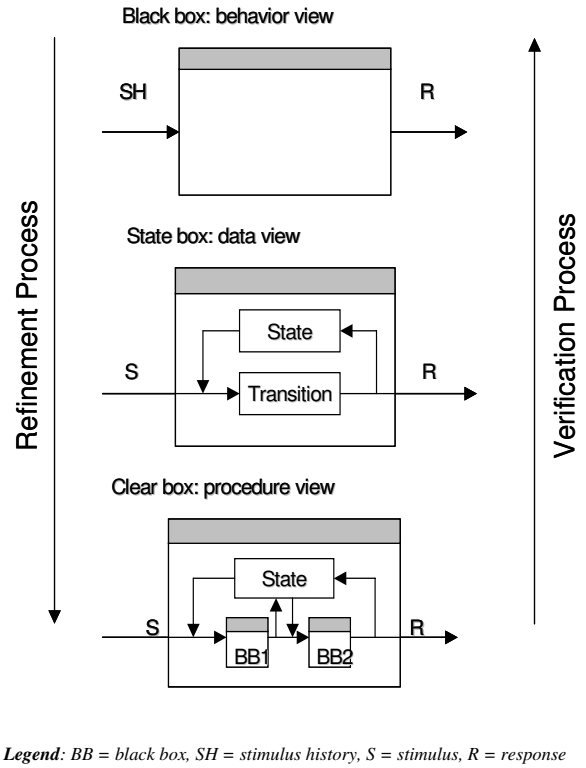


Figure 2.1: Box Structure Refinement and Verification, from [45]

box view represents an object's behavior in terms of its interface and state data. A clear box shows an object's internal behavior.

2.5 Darlington — Early Version of Document Driven Inspection

Document Driven Inspection Approach

The *Document Driven Inspection* (DDI) approach [35] proposed by Parnas is a systematic and rigorous approach to software verification based on precise, well-organized, mathematical documentation. This approach is built upon Harlan Mills's *program function* [27] and the principle of *divide and conquer*.

Tabular expressions (or *tables*) [20] is used as a basic notation in the documentation. *Tables* are a good, straightforward, yet systematic and formal way of presenting and specifying the software, because Mills (and others) showed that a terminating program can be represented as a *function* – so function tables (tabular expressions) are an easy-to-read notation capable of describing functions and therefore programs.

Good documentation is one of the keys to assure software quality [34]. Documents record requirements and design decisions throughout the software development life-cycle; therefore, the quality of documents is essential to inspection and future maintenance.

Chapter 4 will further discuss *tabular expressions* as well as the *Document Driven Inspection* approach.

Darlington Project

The practicality of this "divide and conquer", "document based" technique for performing inspections has been demonstrated by several industry applications. One of the early applications was the safety assessment of a shutdown system in the Darling-

ton Nuclear Power Station of Ontario Hydro (currently Ontario Power Generation Inc. -OPG) [32][33][37].

The Darlington Nuclear Power Generating Station is the first in which the safety shutdown systems are computer controlled. Canadian Nuclear Safety rules mandate that safety systems must be independent of the control system. There are actually three control systems, with each of them capable of shutting down the reactor in case of an accident. The major control system is responsible for power adjustments under normal operating conditions. Another two are the shutdown systems, whose only job is to shut down the station if anything abnormal happens. At Darlington, the two shutdown systems are computerized, programmed by different teams of programmers in different languages. This was intended to ensure design diversity so that even if both the reactivity control system and one of the shutdown systems fail, the plant will be shut down when necessary.

Therefore, when the Atomic Energy Control Board (AECB, currently Canadian Nuclear Safety Commission, - CNSC) wanted to license these computerized systems before its actual operation, they asked for a rigorous inspection for safety purposes. The inspection was then conducted based on the preparation of precise mathematical documentation for the code so that the complexity of the software system could be restricted and simplified so that the assessment of it could be performed thoroughly and correctly.

Two documents were used in the Darlington case. One is the *Software Design Specification* written and reviewed by nuclear engineers. The other is a set of *Program-Function Tables* derived from the code by software experts. Four groups of people conducted the following different tasks during the inspection:

- produce tabular representation of the software requirements (actually a mixture of software requirements and software design)
- generate program-function tables from the code
- compare the first two teams' work and establish the equivalence between the above two tables by showing step-by-step transformations from one to the other
- audit the above work

The first three groups were composed of staff and consultants from Ontario Hydro, and the members of the fourth group were staff and consultants from the Government of Canada.

Tripod Approach

Another important factor of this inspection approach is the certification of people. During the inspection of the Darlington shutdown system, the above four groups of people were selected from different sources, including domain experts and software consultants, to ensure that the proper experts were involved. The certification of people and processes, along with testing and systematic inspection, form the basic approach called the *Tripod Approach* [37] for the assessment of critical software products.

As for the software inspections itself, we believe that it is also a "Tripod" such that three aspects are of the same importance to a successful inspection. They are people, management process, and technical process & notation, respectively. Neglecting any of them would bring more trouble in an inspection practice and thus be less effective or efficient.

The result of the Darlington project was successful. The inspection was so formal and rigorous that the CNSC was very confident that a precise assessment on this safety-critical project had been achieved. From the customer side, it was also a success because OPG finally got the licence from the CNSC certifying that its system was safe and correct. All of these demonstrated the applicability and success of the approach of Document Driven Inspection. Even now, the CNSC continues to use similar approaches in its own research program.

Chapter 3

Related Work on Inspection of Object-Oriented Software

This chapter introduces the related work in academia and industry for object-oriented inspection. We also describe some of the issues that arise in the inspection of object-oriented software.

3.1 Similarities and Differences Between OO Languages and Procedural Languages

Our case study software is written in an object-oriented (OO) language, namely, Java, while the document driven inspection approach was previously applied more on procedural based languages. It is therefore useful for us to discuss the similarities and differences between the two classes of languages. In this thesis we focus on OO languages with procedural languages that facilitate good modularization, such as

Modula-2.

We can consider a class as a *black box* where inputs go in and outputs come out, since it does a good job of data encapsulation following the principle of *Information Hiding*. That is, state data is hidden inside a class so that it can only be accessed through an access program of the class. The program is decomposed into classes so that all the communication between each other are through method calls only, not by sharing data.

There is a key difference between an OO language like Java and a procedural language like Modula-2 though. Classes are factories of objects, i.e., you can have several instantiations of a class with the same behavior but each with its own encapsulated data structure.

In addition to providing excellent encapsulation capability, object-oriented programming (OOP) introduced inheritance and polymorphism. These are powerful and useful capabilities, but introduce difficulties when we try to document and inspect designs that use these features. Inheritance, in particular, has many problems in practice since it can be used in a way that violates information hiding terribly. The remainder of this chapter assumes that object-oriented design (OOD) means designs that take full advantage of OO languages, including inheritance and polymorphism.

3.2 Issues When Applied to Object-Oriented Inspection

Object-oriented design has developed into a popular technology since its inception in the early 1980's. Software inspection is also a mature and proven technology for

detecting and removing defects. The strange thing is that there are few academic papers that address software inspection of object-oriented software. Differences exist between the non-OO and OO paradigms. The wide adoption of object-orientation raises new problems regarding software quality assurance through inspections.

3.2.1 Inheritance and Polymorphism

The OO paradigm has been widely believed to provide a number of benefits in designing a software system. OO shares some benefits with other programming paradigms, namely, data encapsulation and information hiding, improved extensibility and maintainability. A major distinguishing feature of OO from other paradigms is the significant code reusability facilitated by inheritance and polymorphism.

Inheritance is a powerful feature of object-oriented programming. However, misused inheritance, results in an increased complexity of a software hierarchy, thus making it even harder for inspectors and maintainers to understand the behavior of the software in sufficient detail. There is difficulty in understanding code with inheritance due to the distribution of behavior over several classes. This happens for either single inheritance or multiple inheritance [24].

A "good" use of inheritance is when it is designed for refinement; that is, to allow a specification-implementation (refinement) relationship between the subclass and the superclass [46]. There is not yet an inspection technique, as far as the author is aware, to specifically check issues related to this aspect of inheritance. Some reading techniques, as will be introduced in the next section, cover part of the problem. An experiment using horizontal reading [48] made a set of questions to verify if the class descriptions conform to the related class diagrams. One question was to check that

all the class inheritance relationships were described. Reference [9] also advocates the use of a checklist to examine features like inheritance. Neither of these questions concerned the refinement of any inheritance case. More detailed inspection techniques are needed to ensure that the refinement of subclass from superclass does not break the existing code of superclass, thus preserving extensibility and reusability.

Polymorphism and *dynamic binding* are also powerful features of OO that come with a price. The structure of an OO program at run-time is greatly different from its static code structure. Since the implementation which is called at run time may not be known at the time of writing or even inspecting, it may introduce subtle and unanticipated bugs later on. That is a serious problem for inspectors since they must find out which implementation is being used to determine the behavior of the program. In this thesis, however, we will not cover the issues related to polymorphism and dynamic binding. There are two reasons. First, we have not found an effective way of coping with specifying the dynamic binding of a program through a static inspection technique. Second, as we will explain in Chapter 5, the case study program is only one class of a generic project. The implementation of this class could be different in different specific project afterwards, and the documentation of the related class in a specific project would do a better job of specifying the precise behavior. These topics are definitely very interesting topics, and should be among any list of future work.

3.2.2 Method Size and Delocalization

A typical large object-oriented system consists of many small methods, each of which provides only a little functionality. Together with inheritance, polymorphism and dynamic binding, this can lead to the code required to complete a single task being

dispersed widely throughout a program [8]. As a result, program understanding for a fragment of software requires more than just that fragment. The inspectors or maintainers need to trace up and down the invocation chain to find out what exactly that little fragment does. Note that good information hiding practice requires specifying the programs that access the hidden information, which helps inspectors understand the program.

An early observation of this problem was made by Soloway *et al.* in [49] as *delocalized plans*. Soloway described a *delocalized plan* as "where the code for one conceptually united plan is distributed non-contiguously in a program". Soloway continued to point out that, "Such code is hard to understand. Since only fragments of the plan are seen at a time by a reader, the reader makes inferences based only on what is locally apparent–and these inferences are quite error prone". Although delocalization is not limited to object-oriented software, the object-oriented paradigm exacerbates this through its huge library of small methods and the multi-interaction among them.

Let us take a look at a simple example illustrating the problem of *delocalization* (sometimes called *method distribution*). Figure 3.1 depicts an example of method invocation. A lower rectangle represents an object that inherits or extends the object of its higher rectangle. An arrow means a method call from the arrow beginning object to the arrow end object. The arrow to the lower left object indicates an invocation of one method inside this object. However, it turns out that this method again invokes other methods, which in turn invoke yet other methods, and it keeps on and on. It is often the case that there may be a lengthy chain of method invocations that must be followed to understand the implementation of the very first method. This, not only greatly increases the paths that must be followed, but also destroys the hierarchical

structure. Therefore, the complexity of the system is transferred from method bodies to the interactions between them [24]. It is not surprising that an inspector's work becomes harder, since all these interactions must be understood so as to confirm the effect of a SINGLE method call. Such problem could be alleviated if there is "black box" documentation.

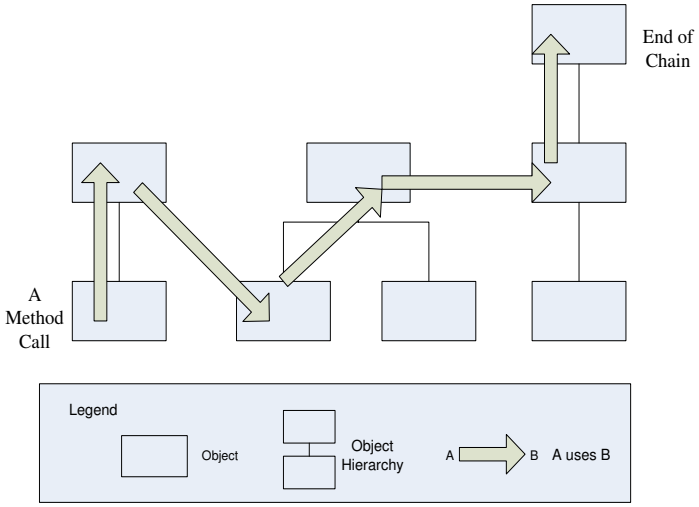


Figure 3.1: A Chain of Method Invocation in an Object

3.2.3 Other Issues in OO Code Inspection

A study from Dunsmore *et al* [10] also suggests that in addition to the *delocalization* described by Soloway, another two significant issues were identified that are arguably crucial in order to make OO code inspections practical for large-scale systems. They are, *chunking*, and *reading strategy*.

Chunking comes from the many undocumented and unspecified dependencies and links between classes that make it very difficult to isolate even one or two classes for

inspection, and delocalization complicates this further. How to partition and select a piece of code for inspection determines what the inspection result will be. *Program slicing* [52] is one of the solutions advocated by some researchers. Again, there are some issues that need to be addressed [8]: (1) the identification of suitable chunks of code to inspect, and (2) the decision of how to break the chunk free of the rest of the system, minimizing the number of dependencies and the amount of delocalization.

Reading techniques addresses the method and the procedure in which the code is read. How to read OO code more efficiently and effectively? How to make sure the reading strategy used helps the inspectors find the most defects in the code? Is the current reading strategy enough for all the issues applicable in OO inspection?

The next section will introduce current reading techniques advocated in the field.

3.3 Reading Techniques that Help Improve OO Inspection

There are several groups of researchers dedicated to improving inspection methods, and many of them suggest an approach called *software reading techniques*. In this section we introduce some of the new reading techniques aimed at understanding and inspecting high-level OO design document and code.

A reading technique can be defined as a series of steps for the individual analysis of a software product to achieve the understanding needed for a particular task [48]. Experience has shown that documents play a central role in all the phases of software development. Thus, understanding the documents is a key technical activity for software development. There are a variety of reading techniques, and most of them

fall under the following four categories:

Ad-Hoc Unstructured, absolutely no guidance for inspectors

Checklist-Based Reading (CBR) Checks for sources of common errors. This is a commonly used reading technique and also deemed as the most effective one by many.

Perspective-Based Reading (PBR) Inspectors are given a set of instructions on how to read code from different perspectives (i.e. tester, designer, user).

Use-Based Reading (UBR) Inspectors manually execute use-cases, a new promising approach to detecting usage-critical errors.

In addition to these, Shull et al. [48] defined a family of reading techniques for the purpose of defect detection of high-level OO designs diagrams represented using the Unified Modeling Language (UML). A high level design is a set of artifacts concerned with the representation of real world concepts [50]. The objective of the inspections using these techniques is to try to capture the static and dynamic views of the problem using UML notation: class, sequence, and state diagrams.

The main idea is to define one reading technique for each group of the above diagrams. For example, use cases need to be compared to interaction diagrams to detect whether the functionality described by the use case is captured and all the expected behaviors regarding this functionality are represented. The next step is to use both horizontal and vertical readings to identify different aspects of defects in all documents. While horizontal reading concentrates on comparisons of documents within a single life cycle phase, vertical reading tries to compare documents between phases. The idea of this approach could be traced back to Linger and Mills' [22].

The details of this approach and some empirical studies and experiments conducted can be found in [47][48][50]. Based on their results, vertical techniques tend to find more defects of omitted and incorrect functionality, while horizontal techniques tend to find more defects of ambiguities and inconsistencies between design documents.

There are also some other reading techniques tailored for OO software. Dunsmore *et al.* [10] developed three techniques to inspect practical object-oriented code.

Systematic Abstraction-Driven Strategy Aims to enforce an understanding of the code by having the inspectors create abstract specifications for each method as they read them. The ordering of code for inspection and the use of stepwise abstraction to deal with delocalization are aspects of the technique that were recommended by the author.

Use-Case Reading Strategy Attempts to support inspection from the dynamic aspect of object-oriented systems. The procedure of this technique is to check the response of each object in all the possible scenarios from the user's point of view. Therefore the benefit is obvious in that it checks the code explicitly against the requirements.

Checklist Strategy Checklist as introduced above is a well-established reading support approach often adopted by inspectors during the preparation of the inspection. Gilb and Graham [15] recommend that checklists should be based on historical information and should not be composed of general, and therefore potentially irrelevant, questions obtained from elsewhere.

Note that the first technique, Systematic Abstraction-Driven Strategy, is essentially what is done in the display method [40] of the DDI approach, which will be introduced in the next chapter.

According to the evaluation of these three techniques through an experiment done at Strathclyde University, Dunsmore claimed in [10] that Checklist is the most effective approach but the other two also have potential strengths and so for the best results in a practical situation a combination of techniques was recommended.

Chapter 4

Document Driven Inspection of Object-Oriented Software

This chapter briefly introduces the *Document Driven Inspection* approach proposed by David Parnas, and further discusses the approach from the aspect of Object-Oriented software. Readers interested in the Document Driven Inspection approach and its previous applications may explore the following references: [19][32][33][37][51].

4.1 Tabular Expressions

4.1.1 What is a Table?

Tabular Expressions (or *tables*) [20][21][31] developed by Parnas et al. are an important concept in the Document Driven Inspection approach. We assume that the readers already have basic knowledge of standard mathematical notations. This section reviews the concept of *tables*.

Tabular expressions are practical notations influenced by Mills' *Program Function* [27]. Tables can represent any functions or relations, especially those complex ones to specify software behavior precisely, unambiguously, and completely.

Tables are essentially multi-dimensional expressions, represented by sets of *cells* (*headers* and *grids*). The use of headers and grids logically separate the complex conditions of a software system into many small cases. Two important characteristics of computers [31] suggest describing functions in computer systems in the form of tables. First, digital technology requires us to implement functions that have many *discontinuities*, discontinuities that can occur at arbitrary points in the domain of the function. Second, the range and domain of these functions are often *tuples* whose elements are of distinct types and the values of the tuples cannot be described in terms of a typical element.

Formally, as defined in [21],

- A *header* H is an indexed set of cells, $H = \{h_i | i \in I\}$, where $I = \{1,2, ..., k\}$ (for some k) is an index set.

- A *grid* G indexed by *headers* H_1, ... ,H_n, with $H_j = \{h_i^j | i \in I^j\}$, $j = 1, ..., n$ is an indexed set of cells G, where $G = \{g_\alpha | \alpha \in I\}$, and $I = \prod_{i=1}^{n} I^i$ (or $I = I^1 \times ... \times I^n$). The set I is the index set of G.

- A *raw table skeleton* is a tuple

$$T = (H_1, ..., H_n, G)$$

where $H_1, ..., H_n$ are headers and G is the grid indexed by the headers. The

elements of the set $Components(T) = \{H_1, ..., H_n, G\}$ are called *table components*.

A variety of tables have been developed by SQRL researchers. Among them are *Normal Tables*, *Inverted Tables*, and *Vector Tables*. The syntax and semantics of those tables can be found in [31][39].

A *Normal Function Table* is a common table format that we used in our case study documentation. An example of a *Normal Function Table* is given below [20]. Consider a function f:

$$f(x,y) = \begin{cases} 0 & \text{if } x \geq 0 \wedge y = 10 \\ x & \text{if } x < 0 \wedge y = 10 \\ y^2 & \text{if } x \geq 0 \wedge y > 10 \\ -y^2 & \text{if } x \geq 0 \wedge y < 10 \\ x + y & \text{if } x < 0 \wedge y > 10 \\ x - y & \text{if } x < 0 \wedge y < 10 \end{cases} \tag{4.1}$$

A conventional mathematical expression of function *f*, which is not very readable, is:

$$\begin{aligned} (\forall x, (\forall y, ((x \geq 0 \wedge y = 10) \rightarrow f(x,y) = 0) \wedge ((x < 0 \wedge y = 10) \rightarrow f(x,y) = x) \wedge \\ ((x \geq 0 \wedge y > 10) \rightarrow f(x,y) = y^2) \wedge ((x \geq 0 \wedge y < 10) \rightarrow f(x,y) = -y^2) \wedge \\ ((x < 0 \wedge y > 10) \rightarrow f(x,y) = x + y) \wedge ((x < 0 \wedge y < 10) \rightarrow f(x,y) = x - y))) \end{aligned} \tag{4.2}$$

The equivalent tabular expression of function f is:

H1

$\frac{H1 \wedge H2}{G}$	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	x	$x + y$	$x - y$

H2 G

Table 4.1: An Equivalent Table Defining f

4.1.2 Why Tables Help?

The experience on a variety of projects, for instance, the Darlington project we introduced in Chapter 2, demonstrates that *tabular expressions* are of great help in both documentation and inspection. As summarized in [20], *tables* help in thinking, and thus in inspection, through a *divide and conquer* policy.

Software is becoming more and more complicated, and typically people cannot understand it easily. When people try to inspect such software, the complexity often overwhelms them and poses difficulties in correct understanding. *Tables* under such circumstances act as a rigorous yet practical assistance to divide one complex condition into many small ones, so that each case covers a logically small unit precisely, and all the cases together specify the whole program completely. In this way, we reduce the complexity of the task of analyzing a big system as a whole. Further, fighting only small cases one by one, fewer errors will escape from our eyes.

Most importantly, two properties of a table, *completeness* and *disjointness*, facilitate a mathematically rigorous and correct procedure, yet an easy-to-check approach of specifying the software behavior exhaustively. With such properties, one can be confident that the specification is complete and unambiguous. The definitions of these properties will be introduced in Section 4.3.

Tables also help us do a better job when documenting software. It is obvious that natural language expressions are often vague and imprecise and thus misleading. Conventional mathematics expressions, which are normally rigorous and precise, are often too complex and hard to read and check. As the previous example shows, even for a simple function, conventional logical expressions are not very readable. Tables, instead, can easily provide much more readable documentation.

4.1.3 A "While" Table

One of the challenges we faced when producing the *Module Internal Design Document* for our case study is that we need new notations to fully and precisely specify the behavior of a program in a table. An example is the development of a so-called "while table".

It is well recognized that a digital computer can be viewed as a finite state machine that is always in one of a finite set of states, and whose operation consists of a sequence of state changes, i.e., transitions from state to state. Further, every *terminating deterministic* program can be described by a *program function* [40]. However, difficulties arise when specifying behaviour in which the termination condition of a loop depends on an external call to a program that has not been written or for which there is no specification, or we may not want to (or cannot) include the behaviour of the external call within the specification of the local behaviour.

Consider the example in Figure 4.1 that illustrates the case when behavior is determined by the return value of an external call from within a loop.

Assume that *outValue.show()* is an external call that prints the number of invocations of *outValue* on the screen; *outValue.init()* sets the current invocation number

```
outValue.init;
while (!cond()){
    outValue.show();
}
```

Figure 4.1: A Non-Deterministic While Loop

of *outValue* to 0; and *cond()* is an external call that returns a *boolean* value.

We could describe the behavior by documenting output traces. Such documentation may not be very readable. For example,

$$NOT\ [cond()].NOT\ [cond()].cond() \Longrightarrow$$
$$\text{invoke } outValue.show.\text{invoke } outValue.show$$

We suggest rather that we document such program through a *while table*. Table 4.2 shows the while table specifying the behavior of the example.

n = 0, 1, 2, ...	outValue.show<n>	Loop Terminates
cond() <n>	False	True
NOT [cond() <n>]	True	False

Table 4.2: Specification of the While Loop Program Example

We introduce a tag $< n >$, indicating the value of the identifier at the n^{th} iteration through the loop. Results from the loop may be tagged with $< n+1 >$. The outputs and/or the termination condition of a *while* loop may be different for each iteration. That is, their values are determined by other programs or sometimes by the end-user input. In our example, the output is generated outside of the loop body, therefore the different values of output for each loop iteration may or may not cause loop termination. Since the value of the termination condition may also change, loop termination must be explicitly described as a boolean value in the table.

The advantages of the *while* table are:

- They describe one iteration of the loop behavior in an understandable way;
- The total loop behavior is derivable from the table. For instance, in more complex cases we could functionally compose $n = 0$ and $n = 1$ to obtain the behavior after the first two iterations. We could then compose the result with $n = 2$ and so on until *Loop Terminates* is *True* under all conditions. In cases where there is no specification of the invoked programs in a loop, the effective loop behavior can still be documented in a readable way using the while table.

4.2 Precise Documentation of Design Decisions

4.2.1 Software Design Principles

This section provides an overview of software design principles that we use to examine and judge the case study design and documentation. These principles also act as our philosophy in producing our documentation. More details about these principles can be found in [3][19][29][38][41][42].

Design Philosophy: Information Hiding Principle

The concept of *information hiding* comes from the seminal paper, "*On the criteria to be used in decomposing systems into modules*" authored by Parnas in 1972 [29]. The principle was induced from Dijkstra's THE paper in 1968 [7] which described a nice system with "a strict hierarchical structure". The purpose of *information hiding* was to obtain a modularization structure of a system that can minimize the number of modules affected when an unexpected change is made.

Information hiding is the key to reuse. The core idea is to hide implementation details of one module from its outside world, and keep its interface defined and not changed while the implementation or *secrets* of this module can be changed without affecting its interface, and thus the other modules which may use it. As defined by Parnas [30], "secrets" are design decisions that are likely to change, and these *anticipated changes* should not be revealed by the module interface. These concepts are critical to system decomposition and the designing of modules.

As a software design principle, information hiding is widely accepted in both academic circles and industrial practice, and many successful design ideas come from it. For instance, one of the most important traits of object-orientation, *encapsulation*, was inspired by information hiding.

Information hiding is the theoretical foundation of both *functional documentation* [39] and the *document driven inspection* technologies.

Precise Specification of Requirements

A requirements document is a primary document and the basis of the entire software development life cycle in that a software application is made to fulfill the *requirements*

from the customer. A precise and complete requirements document is crucial to the ultimate success of a system. A precise and complete specification of requirements can be achieved through a formal and rigorous notation. *Tabular expressions* as discussed previously, provides rigorous while readable and practical methodology that has been illustrated in several projects. Tables are good for many circumstances, for example, real-time systems as in our case study, but not for everything. For instance, in cases that algorithms or Graphical User Interfaces (GUIs) must be described, tables would not serve as the best documentation means.

Decomposition into Modules

Decomposing a large program into information hiding modules is the first step of design. This should be done by identifying design decisions that are likely to change, known as *secrets*, and then designing a hierarchical system of modules, so that each secret is hidden inside a module. If this can be achieved, any one of these identified design decisions could be changed by modifying a single module.

Interface Design

Based on [3], the *interface* between two programs consists of the set of assumptions that each programmer needs to know about the other program in order to demonstrate the correctness of his own program. The design decisions that are unlikely to change facilitate an interface design so that all the assumptions that other programs can make about this module can be found in its interface document. Therefore, a good interface design document should include not only *type* information but the external *behavior* of this module that the other modules can rely on.

4.2.2 Precise Documents that We Need

As Parnas pointed out [34], documentation is the key to better design and thus better software, and precisely documenting all the design decisions plays a crucial role in the inspection. In this work, we are building on this by using the same document model even if the inspectors themselves are forced to develop the documents as introduced below.

According to the *functional documentation* approach [34][39], several documents are required for the purpose of precisely specifying and documenting the software.

- **A Requirements Document** that tells the users exactly what they will get and tells the programmers exactly what to build.
- **A Module Guide** that provides guidance on exactly how the program will be affected by a proposed change, as well as a hierarchical structure of the system.
- **A set of Module Interface Specifications (MIS)** that tell programmers what they must build and other programmers, what they can expect of a module.
- **A set of Module Internal Design Documents (MID)** that record major internal design decisions, guide the programmers in coding, and help maintainers to understand the code.

In this work, two of the documents were produced, i.e. the Module Interface Specification and the Module Internal Design Documents for the case study. The MIS works as a "black box" description of the module, while the MID treats the module like a "clear box". Chapter 6 will discuss the issues on and the considerations about producing these documents. As mentioned, although both the requirements

documents and the module guide are very important to the design and inspection, we are not going to discuss the application of documenting them from the point of view of OO, simply because we did not have a chance to work on them in our case study.

4.3 Document Driven Inspection Techniques

Inspection must be performed in a disciplined way so that no cases are overlooked and each case can be examined carefully. Document Driven Inspection (DDI), proposed by Parnas [33][35][37], is a systematic and mathematically rigorous approach to perform a formal and effective review on design and code. This approach depends heavily on rigorous and systematic documentation.

The DDI can be outlined by the following characteristics [35]:

- Employs hierarchical decomposition rather than sequential reading for systematical inspection of a system.
- Uses mathematical notations to provide precise and rigorous specifications rather than informal paraphrases.
- Produces useful precise documentation as a basis for inspection when it is not available.
- Has the confidence that cases are not overlooked or overlapped based on the properties of Program Function Tables.
- Applies mathematics to check for completeness and consistency of documents.
- Can take advantage of mathematics based tools such as theorem provers and computer algebra systems.

4.3.1 Active Review on Design Documents

A number of techniques have been developed to perform the task of inspection, among them, the *active design reviews* technique [41] introduced by Parnas and Weiss is an effective review on design documents provided that the documents are complete before the review. This technique makes good use of both the skills of the reviewers and the design documentation. Rather than conducting a large inspection meeting, this approach focuses on identifying distinct types of reviews for finding different types of design errors. The reviewers are also classified based on their expertise. Therefore the reviewers can concentrate on a clear responsibility to perform the task. Conducting the active review is beyond the scope of this work. As we explained in Chapter 1, the reason why we did not perform an inspection is we simply did not have enough time to do both tasks of documentation (as the basis for the successive inspection) and inspection on our case study. Interested readers on how to perform an active design reviews may consult [41] for details.

4.3.2 Code Inspection through Documents

The goal of our inspection process is to develop confidence in the quality and correctness of the code.

Code Inspection Process

Our code inspection process comprises the following steps:

1. Prepare a precise specification of what code should do based on the design.
2. Produce the descriptions of what code actually does using either *Displays* or *Program Function Tables*.

3. Compare the "top level" display or the major function table description with the design specifications.

Why Choose MIS/MID Instead of Displays?

We decided to use the module interface specifications (MIS) and the module internal design document (MID) through the *Program Function Table (PFT)* notation to produce our documentation.

The *display method* approach [40] works best in the inspection of a long program in that it systematically describes the invocation chain (data flow) of such a program. However, we made the decision to choose MIS/MID instead of *Displays* for the following reasons.

First, at the time that we did the documentation, we did not have access to adequate documents that can provide the details for the complete data flow of the examined module. Second, there are some new issues when documenting OO code using Displays. Not only are there numerous invocations, but those invoked programs are usually non-local to the invoking programs (methods or classes). This is called *delocalization* as discussed in Chapter 3. This phenomenon limits the advantages of displays from specifying behavior of programs and their subprograms (i.e.,the invoked access programs) completely.

We then turned to the alternative method of MIS/MID. We still take advantage of tabular expressions, but leave the details of the invocation chain to the original programs themselves. That is, we do not specify the internal design decision of invoked access programs of a program, instead, we only specify the origin of those access programs so that readers may refer directly to their specifications.

Disjointness and Completeness of *PFT*

The structure of the *Program Function Table* (PFT) used in our case study is derived from that used in the Darlington project [51] as shown below in Table 4.3.

	Result		
Condition	$name_1$	...	$name_m$
$condition_1$	$result_{11}$	...	$result_{1m}$
...	...	...	...
$condition_n$	$result_{n1}$	...	$result_{nm}$

If $condition_1$ **then** {$name_1 = result_{11}$; ... ; $name_m = result_{1m}$}

elseif ...

elseif $condition_n$ **then** {$name_n = result_{n1}$; ... ; $name_m = result_{nm}$}

Table 4.3: Program Function Table Template

Based on [51], two primary properties of *PFT*, *disjointness* and *completeness*, are defined as,

- **Disjointness**

  $Condition_i \wedge Condition_j \Longleftrightarrow FALSE, \forall i, j = 1...n, i \neq j$

- **Completeness**

  $Condition_1 \vee ... \vee Condition_n \Longleftrightarrow TRUE$

Disjointness holds if no conditions are overlapped, and *completeness* holds if the conditions cover the complete input domain.

Thus, these two properties make the application of *PFT* precise and complete in specifying the behavior of a program.

4.4 Dealing with the Complications Arising from OO

In Chapter 3, we already discussed the issues raised when inspecting OO software. They are equally important. However, due to the limited case study code, only some of them have brought to our considerations during our documentation.

- Delocalization

  Delocalization, as indicated in Chapter 3, is a common issue that is observed in OO code. Although delocalization is not confined to OO software, the OO paradigm exacerbate this through its large and powerful library of many small methods and the multi-interactions among them. The use of *displays* was intended to deal with the issue of delocalization. However, as discussed in the previous section, we decided to use the approach of MIS/MID to produce the design documents due to the delocalization problem. Delocalization is widely observed in OO code in that many invocations from one method are distributed across the whole system. It is not practical to give specifications of every single method in each method that invokes it. A method may be invoked, say 10 times, in different classes, so there would be unnecessary redundancy of information if we have to document the same specification of this method 10 times in different

classes. Another observation is, multiple invocation chains are typical in OO code. To document one program correctly and completely, one may have to trace a long way back to reach the last method in the invocation chain, i.e., a method that does not invoke other methods. To cope with such situations, we used MIS/MID as introduced previously for documentation. We treat the non-local invocations as external, and then only indicate the origin of those invoked programs, so that the reader can refer to the complete and precise specifications of those programs. In this way, we will have the confidence that the specification of one program is complete and correct, with the assumption that each of its invocations has a correct interface specification.

- Uses Relation

  In [44] "uses" is a relation defined as: P_1 *uses* P_2 if a working copy of P_2 is needed in the system in order for P_1 to meet its specification.[1] Further, [30] states that "by restricting the relation *uses* so that its graph is loop free we can retain the primary advantages of having system parts *use* each other while eliminating the problems". The powerful library of Java langauge (in our case) complicates the *uses* relation among the modules, probably because a program can conveniently use the available methods from the library to implement its own functionality. In OO code, it is quite usual to find that the *uses* relation contains loops, that is, a program A may use methods of another program B while at the same time B has to use methods of A for its implementation. This causes problems especially when we try to develop a big system in a team. Therefore, to design with *uses hierarchy* [42] among classes in mind is of great

[1] P_1 and P_2 are program names.

practical importance. We also found cases of *uses* loop in our case study. To precisely document such program, the only thing to do is to regard all of the programs in the loop as a single program that must either be included as a whole or excluded, which seems impractical in practise due to the large body of such a loop. Ideally, *uses* loops should be eliminated through a clear and modular *uses* hierarchy of all the modules in a system.

- Inheritance

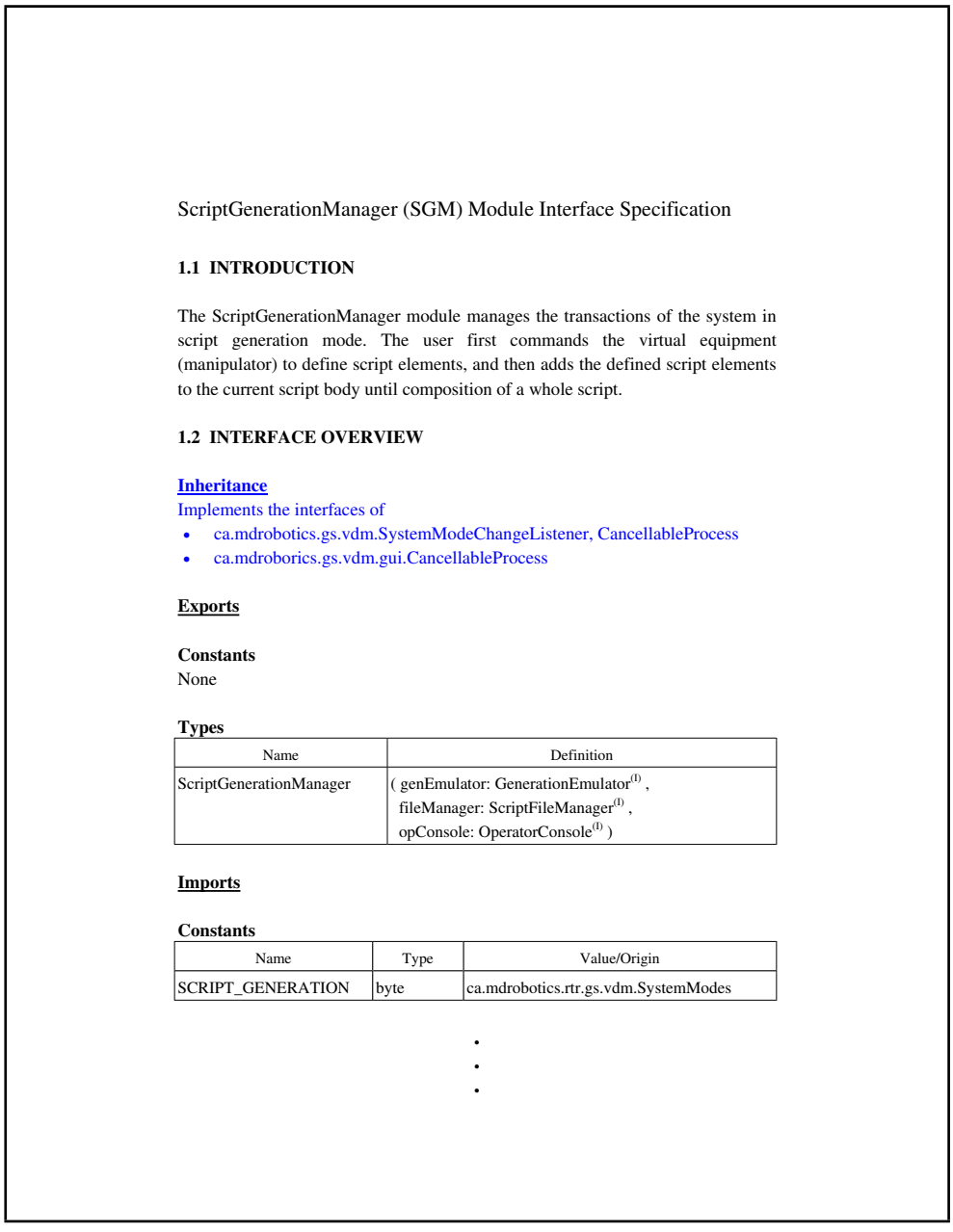
  Inheritance is no doubt an important feature of the OO paradigm. To document a program with OO design and language using a *functional documentation* approach is not a new topic. The *program family* concept [43] by Parnas already brought forward the idea of *module specification* that is similar to *inheritance*, yet focuses more on the essence in a different way. In [2], a solution was also proposed to document class inheritance in module *internal* specifications via the "foreign types lists".

  Java does not support true multiple inheritance. Besides the basic form of inheritance, Java decided to use a notion of *interfaces* instead of superclasses to deal with the situation of requiring a class to reflect the behavior of two or more parents. Note that "interface" in this context means Java's specific definition of an interface, i.e., a declaration of a type without an associated implementation. Both kinds of inheritance are used in our case study systems. For the inheritance of classes, our solution is to document the inheritance relationship of two classes by first indicating the parent class as a source type in the *MIS* instead of the MID of the child class. We then document the names of the methods that inherit from the parent class in the MID of the child class, without the detailed

implementation in cases where there is no overriding methods. The benefit of such documentation is, the MIS would give the readers a whole picture of inheritance hierarchy from the parent class to its child class, while on the other hand, the associated MID is clear and simple without duplicated information and the readers can easily refer to its parent class for the detailed specifications of any inherited method.

The second kind of inheritance, from interfaces, occurs in our case module. Figure 4.2 shows an example of documenting a case of interface inheritance in its MIS document. Figure 4.3 shows the same example in the associated MID. As the example shows, two interfaces *SystemModeChangeListener*, and *CancellableProcess*, are required to be implemented in the class of *ScriptGenerationManager*. Similar to the class inheritance, we document interface inheritance in both MIS and MID documents of the inheriting class, but with all the implementation details since no implementation is required in an interface. In the MIS (as shown in Figure 4.2), we first specify the name of the interface (with the long class source, if any) in the overview, and then the detailed specifications of the methods of that interface in the access program section. We also denote the interface name before the behavior specification of its associated methods in the MID (as shown in Figure 4.3).

There are also some other issues that are important. One of them is *dynamic binding*. It is commonly applied in our case study system. However, we did not have the opportunity to document *dynamic binding* in the slice of code used for our case study. Admittedly, it is not easy to document it using tabular notation due to the static characteristics of *tables*. New notation might be needed. Future work on this



ScriptGenerationManager (SGM) Module Interface Specification

1.1 INTRODUCTION

The ScriptGenerationManager module manages the transactions of the system in script generation mode. The user first commands the virtual equipment (manipulator) to define script elements, and then adds the defined script elements to the current script body until composition of a whole script.

1.2 INTERFACE OVERVIEW

Inheritance
Implements the interfaces of
- ca.mdrobotics.gs.vdm.SystemModeChangeListener, CancellableProcess
- ca.mdroborics.gs.vdm.gui.CancellableProcess

Exports

Constants
None

Types

Name	Definition
ScriptGenerationManager	(genEmulator: GenerationEmulator(I) , fileManager: ScriptFileManager(I) , opConsole: OperatorConsole(I))

Imports

Constants

Name	Type	Value/Origin
SCRIPT_GENERATION	byte	ca.mdrobotics.rtr.gs.vdm.SystemModes

.
.
.

Figure 4.2: An Example of Documenting Inheritance of Interfaces in MIS

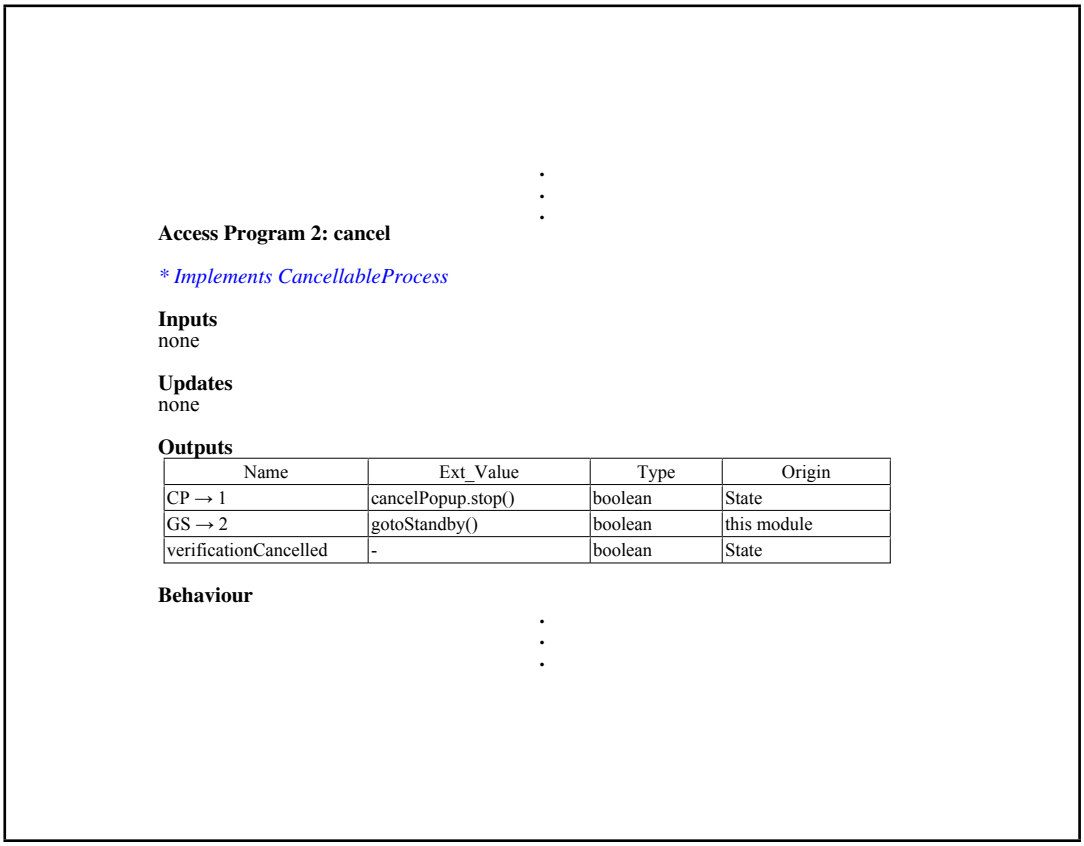


Figure 4.3: An Example of Documenting Inheritance of Interfaces in MID

topic is not only necessary but challenging.

Chapter 5

Introduction to the Case Study

We have been working with industry to apply our approach in practice through a project supported by Materials and Manufacturing Ontario (MMO) and MD Robotics (MDR). This chapter gives a brief introduction to the background of the project, as well as a brief description of the program that was used as our case study.

5.1 Project Objectives and Scope

5.1.1 Objectives

This project applies a sound procedure for the documentation and inspection of critical software, using extracts from a software application supplied by MD Robotics. The project also provides sample software design documents for the MD Robotics code, and suggests some improvements to the documentation and inspection techniques used in the case study.

5.1.2 Scope

We intended to do an inspection on the case study software. The class *ScriptGenerationManager* in the Remote Teleoperation of Robotics (RTR) system was selected due to its important role in the system, and because it is large enough to test our methods. However, it turned out that the relevant documents we had access to are vague and incomplete and thus cannot be used as a solid basis for our inspection. Therefore, after communicating with the customer, the scope of our task was changed from inspection to documentation.

The appendix shows the sample documents we produced for MD Robotics:

- Module Interface Specification for the identified subset module, *ScriptGenerationManager*
- Module Internal Design Document for the identified subset module, *ScriptGenerationManager*

These documents were produced based on the source code provided by the customer.

5.2 System Overview

To help the readers better understand the complete system, this section will briefly describe the Remote Tele-operation of Robotics (RTR) Operator Station, and its implementation instance, the Remote Operation with Supervised Autonomy (ROSA) Ground Station. The following description is extracted from the Software Architecture Document from MD Robotics.

“ Purpose

Operating robotic equipment at a distance requires a facility for generating the commands, relaying the commands to the robots and monitoring the progress of the robots. Scripted control of robotics equipment is a useful technique for overcoming communication latencies large enough to preclude direct control. Scripts that enable autonomous behaviour of the robots can reduce the communication bandwidth required.

RTR Operator Station

The RTR Operator Station (ROS) is a system for generating variable-autonomy scripts, rehearsing the scripts and monitoring execution of the script via telemetry received from the remote robotic equipment. A 3D virtual representation of the remote worksite equipment is used to plan, rehearse and monitor the execution of a script.

The ROS is not project-specific and can be tailored to control any robotic equipment.

The ROS system supports seven operational modes identified as relevant during requirements analysis: Script Generation, Script Rehearsal, Observation, Script Execution, Import Worksite Data, Model Correction, Direction Execution. ”

Our task is to examine one of the modes, that is, the Script Generation mode, as described below in the MD Robotics documents.

> **“ Script Generation Mode**
>
> *Script Generation* mode enables the user to create a new script. The virtual manipulator can be driven using either the hand controllers or the GUI, and keyframes can be created manually or automatically. High-level script elements, if applicable, can be added to script and are verified in the virtual environment.
>
> No commands are sent to the remote worksite equipment in Script Generation mode. ”

ROSA Implementation

The Remote Operation with Supervised Autonomy (ROSA) is the current project implemented by MD Robotics on the basis of the RTR framework. The objective of this project is “to develop and demonstrate a *variable autonomy* architecture to support ground-based control of space-deployed robotics systems in dynamic workspace environment”.

The ROSA Ground Station (RGS) is an implementation of the RTR Operator Station (ROS). As introduced in the previous section, ROS is only a generic tool providing some capabilities to generate, rehearse, and monitor execution of scripts. Any project that wishes to use these capabilities must customize ROS by implementing several additional modules. In RGS, for instance, a *variable-autonomy* script is a list of instructions to be executed by the remote worksite equipment and may contain a mixture of low-to-high-level script elements. To implement that, the ROSA script language defines one low-level controller (i.e., the manipulator), and five high-level controllers: visual servo, vision system, behaviour executor, cognitive controller, and

hierarchical task network.

The ROSA project is discussed in more details in [18].

5.3 System Decomposition

5.3.1 System Architecture

At a high level, the ROS system contains the following classes:

- Visual Display Module (VDM)
- User Input Devices
- 3D Model Files
- Generation Emulator
- Rehearsal Emulator
- Contextual Display Module
- Remote Worksite Equipment

The VDM possesses the core functionality of the operator station. This class contains 3D virtual environment display and all of the graphical user interface tools. It controls the flow of events through each of the system's modes, provides the graphical user interface, and controls the communication of the various external systems. The VDM interacts with the other six ROS classes via abstracted class interfaces. The VDM is common to all ROS implementations, however, the other six classes are project-dependent.

Figure 5.1 depicts the decomposition of the Visual Display Module, as described in the MDR documentation.

An alternative view of the decomposition of VDM is shown in Figure 5.2. In this view, VDM is treated as the software application and the classes are documented as software modules, in the style of [4]. The target of this case study, *ScriptGeneration-Manager*, is highlighted in the figure. As indicated by the *legend* in the figure, the leaf modules represent classes containing code.

5.3.2 ScriptGenerationManager Module

As we mentioned above, the ROS system supports seven operational modes, and different mode managers are used to coordinate activities during a single mode of operation. We chose the *ScriptGenerationManager* class as our case study for documentation.

ScriptGenerationManager and Related Classes

The *ScriptGenerationManager* coordinates activities of the system in script generation mode. It follows commands from the user, given through the graphical user interface (GUI), to move the virtual equipment (i.e., the manipulator) to define script elements, and to add the defined script elements to the current script body until the composition of a whole script is complete.

There are a number of classes which are used by the *ScriptGenerationManager* during the process. The essential ones are described below.

The *ScriptGenerationConsole* is the GUI to invoke many of the *ScriptGeneration-Manager* methods to move the manipulator, add script elements, etc.

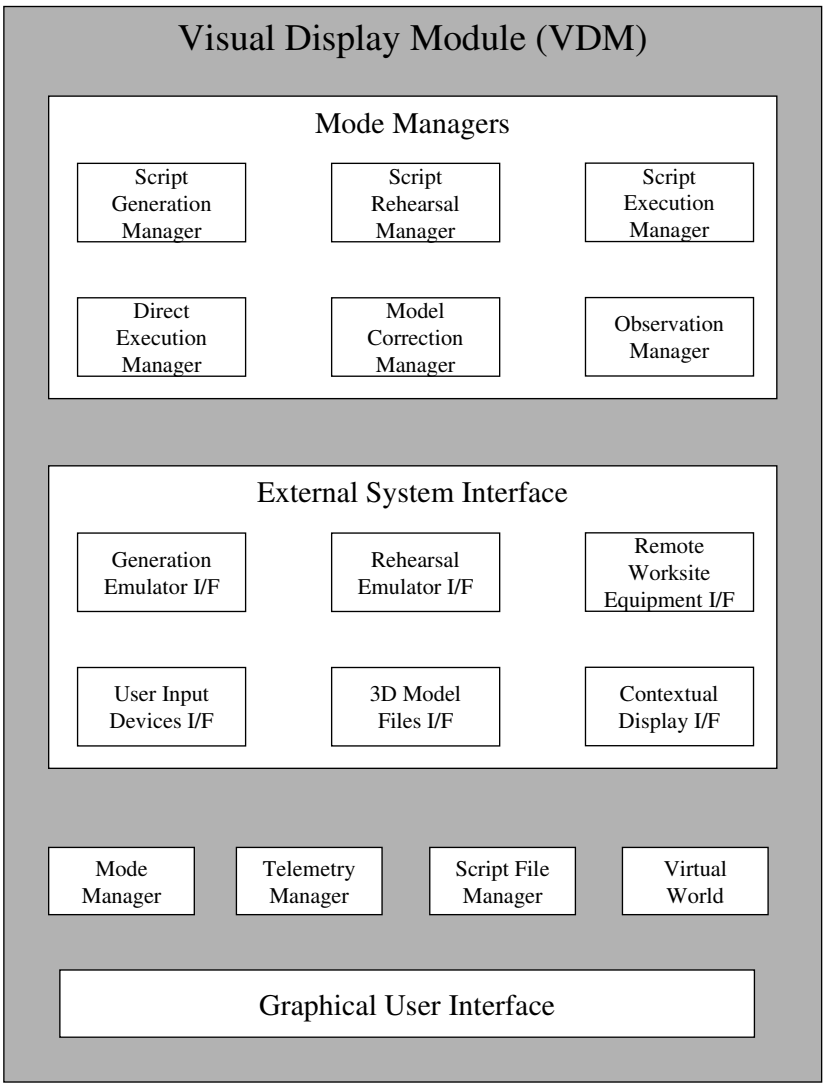


Figure 5.1: VDM Decomposition from MDR Documentation

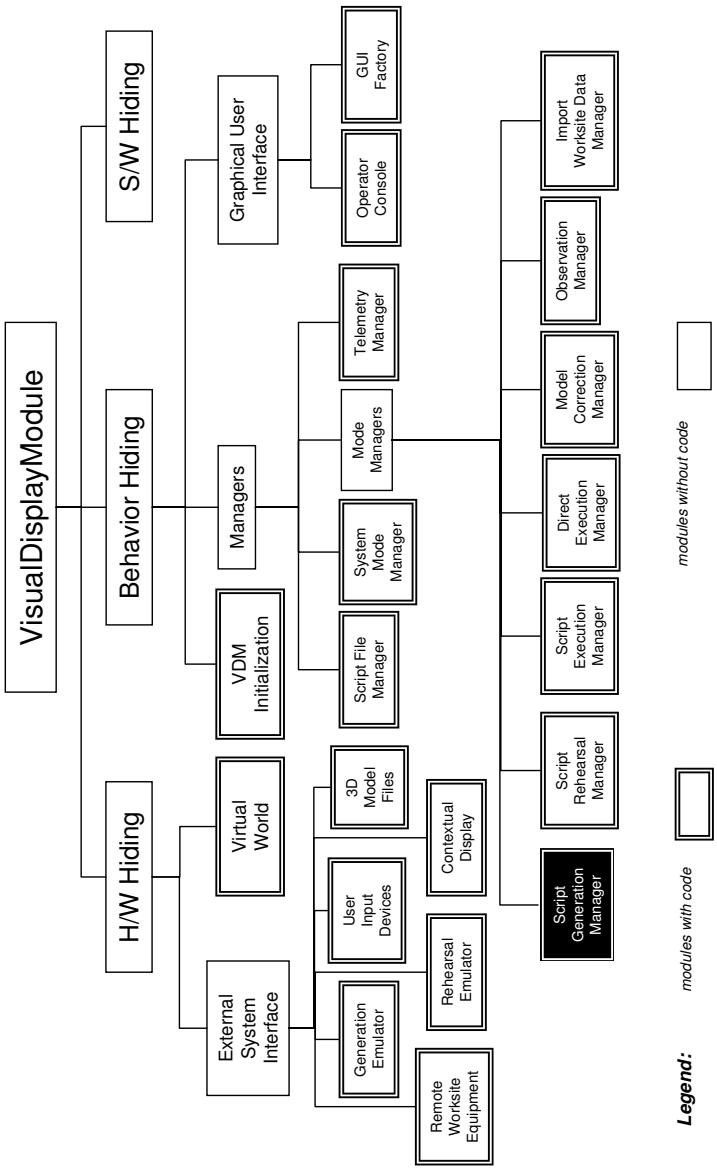


Figure 5.2: An Alternative View of VDM Decomposition

The *GenerationEmulator* is used to execute script elements during script generation mode.

The *Script* class represents a script in any *ScriptLanguage*. A *script* is essentially an ordered sequence of script elements and script segments that are executed to achieve a specific mission objective.

A script segment in the class of *ScriptSegment* is a sequence of script elements and calls to other script segments that can be invoked by a script or script segment.

A script element in the class of *ScriptElement* is the actual script command, and the smallest unit of a script or script segment. Script elements can be either high-level commands, low-level manipulator commands or motion keyframes. An example of a low-level script element is *move to point (x1, y1, z1)*. An example of a high-level script element is *capture satellite*.

Figure 5.3 gives an example of a script with both script elements and script segments inside.

The *ScriptLanguage* object defines the scriptable capabilities of the remote worksite equipment. The *ScriptLanguage* defines the number of low-level controllers, high-level controllers and each of their available script elements. A description of the script elements defined in the RTR ScriptLanguage class follows.

Definition of Script Elements Type

Creating a single script language that supports every type of robotic equipment is a difficult task. Instead, each project may specify its own script language by extending the *ScriptLanguage* class in ROS. As described in MDR documentation, the *ScriptLanguage* class is designed following the singleton and factory patterns [14]. Methods are included for creating the basic elements of any supported script lan-

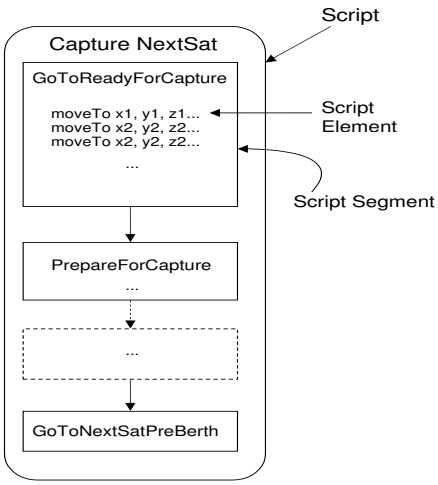


Figure 5.3: A Script Example

guage: POR commands, joint commands, segment calls, high-level command and manipulator commands.

Table 5.1 provides the definition of these basic elements. If an element type is not supported, the *UnsupportedScriptElementException* exception should be thrown.

5.3.3 Script Generation Process

The following sequence walks the reader through the process of generating and saving a script.

1. Selects *Script Generation* mode from the main GUI to display the script generation console.

2. Presses the "New Script" button on the console and is prompted to enter the name of the new script.

Element Type	**Definition**
POR Commands	Drive a manipulator by specifying a 6-DOF destination frame for the manipulator's tip (absolute or relative) within a specified command frame.
Joint Commands	Drive each manipulator joint to a specific (absolute or relative) value.
Segment Calls	Script elements that call another script segment. The script executor will execute the called segment, and then return to this point.
High-level Commands	Script elements whose outcome depends on state and sensor variables. High-level commands can be configured using a custom user interface console.
Manipulator Commands	Miscellaneous commands supported by a manipulator (e.g. latch end-effector). Manipulator commands can be configured using a custom user interface console.

Table 5.1: Definition of Script Elements Type from MDR Documentation

3. Defines the parameters of the first script element using the tabs of the script generation console to assist in controlling the virtual manipulator during configuration.

4. Adds the script element after being verified by the generation emulator. The *cancel* method can be called during verification of the script element.

5. Repeats steps 3 and 4 until mission completion.

6. Save the whole script.

Chapter 6

Application of Document Driven Inspection Approach

This chapter presents the application of the *Document Driven Inspection* approach in documenting a case study that is extracted from an object-oriented code. We start from an analysis of the case study code, and further discuss our considerations when we tried to produce the documentation using *tabular expressions*. We conclude this chapter with the findings and our concerns revealed during the course of producing the documentation.

6.1 Case Study Analysis

ScriptGenerationManager Code Analysis

The *ScriptGenerationManager* class is a public class extended from *java.lang.Object* and implements two interfaces, *SystemModeChangeListener* and *CancellableProcess*. As introduced in Chapter 5, this class is under the Visual Display Module (VDM) of

Item	*Quantity*
LOC (Line of Code)	750
Fields	10
Constructors	1
Methods	28
Imported class	25
Invoked methods	98

Table 6.1: ScriptGenerationManager Code Summary

the ground station system of Remote Tele-operation of Robotics (RTR) project, and is used to manage the generation of scripts during script generation mode.

Table 6.1 presents a summary of the code of the *ScriptGenerationManager* class. As summarized, this class imports 25 other classes from other packages and invokes nearly 100 methods from these classes. This does not include invoked methods from the same package. Therefore, to understand this seems-not-so-big program, which is just an ordinary size program compared to thousands of others in the RTR/ROSA system, the reader has to investigate 100 other programs scattered elsewhere. This is a typical *delocalization* problem as we discussed in Chapter 3.

Moreover, since the RTR project is, as we explained in Chapter 5, a generic project that may be tailored to any specific project in the future, there are many abstract classes and abstract methods that do not contain solid code. The class *ScriptGenerationManager* in the RTR ground station system is not an exception to such a situation. We had to go to its implementation project, in this case ROSA, to find out what the actual behavior is for any method. Such abstraction poses a difficulty in documenting the actual behavior and outputs of this class. Although we

can find the specific implementation in ROSA, other issues remained. We will discuss it in the next section.

ScriptGenerationManager Uses Hierarchy

As Parnas [30] pointed out, "the design of the *uses hierarchy* should be one of the major milestones in a design effort", and "unless some restraint is exercised, one may end up with a system in which nothing works until everything works." Therefore, a system should be designed by considering how to restrict the relation *uses* such that its graph is "loop free" and each level of a uses hierarchy is a *self-contained* unit that is testable and usable. Such systems are easy to extend or contract and thus easy to maintain and inspect.

The *uses* relation of the "ScriptGenerationManager" is somewhat complicated and is not hierarchical. Not only there are many *uses* relations between the classes, but some of them contain *loops*. An example of such a loop is given in Figure 6.1. The method "addScriptElement" of the class of *ScriptGenerationManager* calls several programs including methods named "start", "stop" and "actionPerformed" in the class of *WaitOrCancelPopup*, while at the same time, the class *WaitOrCancelPopup* calls the "cancel" method in *ScriptGenerationManager*. Thus these invocations form a loop in which the programs rely on each other.

In Figure 6.1, each line of text represents a method/access program. Indentation is used to show a method call. Note that the order of invocations is not indicated. Also note that the statements with dotted underlines in *figure 6.1* represent methods that do not have implementations. These include two cases, (1) an *abstract method* that acts as a placeholder, while the subclasses must have concrete implementations of those methods, and (2) an *interface* that cannot have concrete implementations

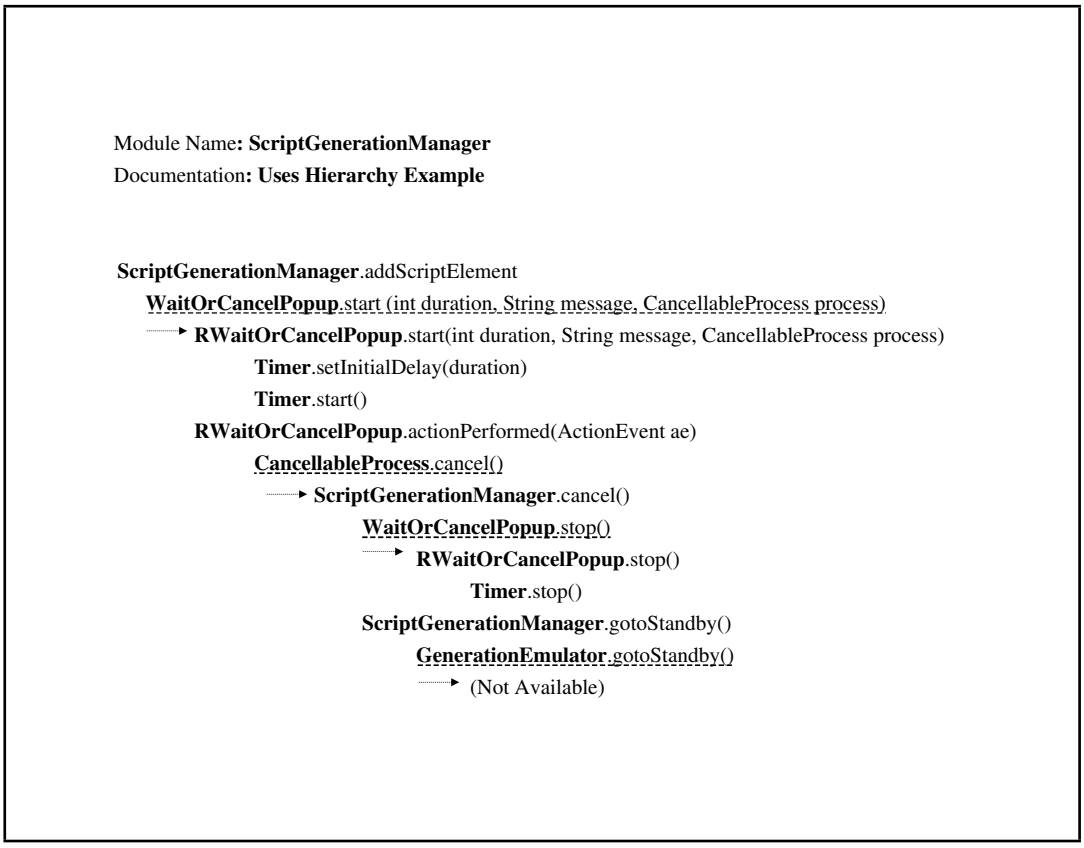


Figure 6.1: An Example Uses Hierarchy

and serves to specify a set of methods that different classes can implement polymorphically.

The readers can refer to the appendix for a big picture of the uses relation of the ScriptGenerationManager class. There are two graphs in Appendix C. The first graph shows a single level of the uses relation for this class, and the second graph tries to present the chain of uses relations for the class, terminating at a class with a detailed implementation or a method call to a Java standard library class. It is apparent from the graph that readers are likely to get lost in the face of such complicated uses.

6.2 On Documenting the Case Study Module

The current available documentation of our case study code is similar to the standard Java code document, namely, its web-based API specification derived from the comments in the source code using a tool named *javadoc*. This API document is convenient and powerful when we want to browse it to search a method or a variable, and ideally, it describes all aspects of the behavior of each method on which a caller can rely. But in practice, it is very hard to make precise and complete comments, and thus the document generated from the comments may not be formal and correct as expected. Therefore, we decided to use a more rigorous and systematic documentation that would be helpful for both design and inspection.

6.2.1 How Did We Produce Module Interface Specifications

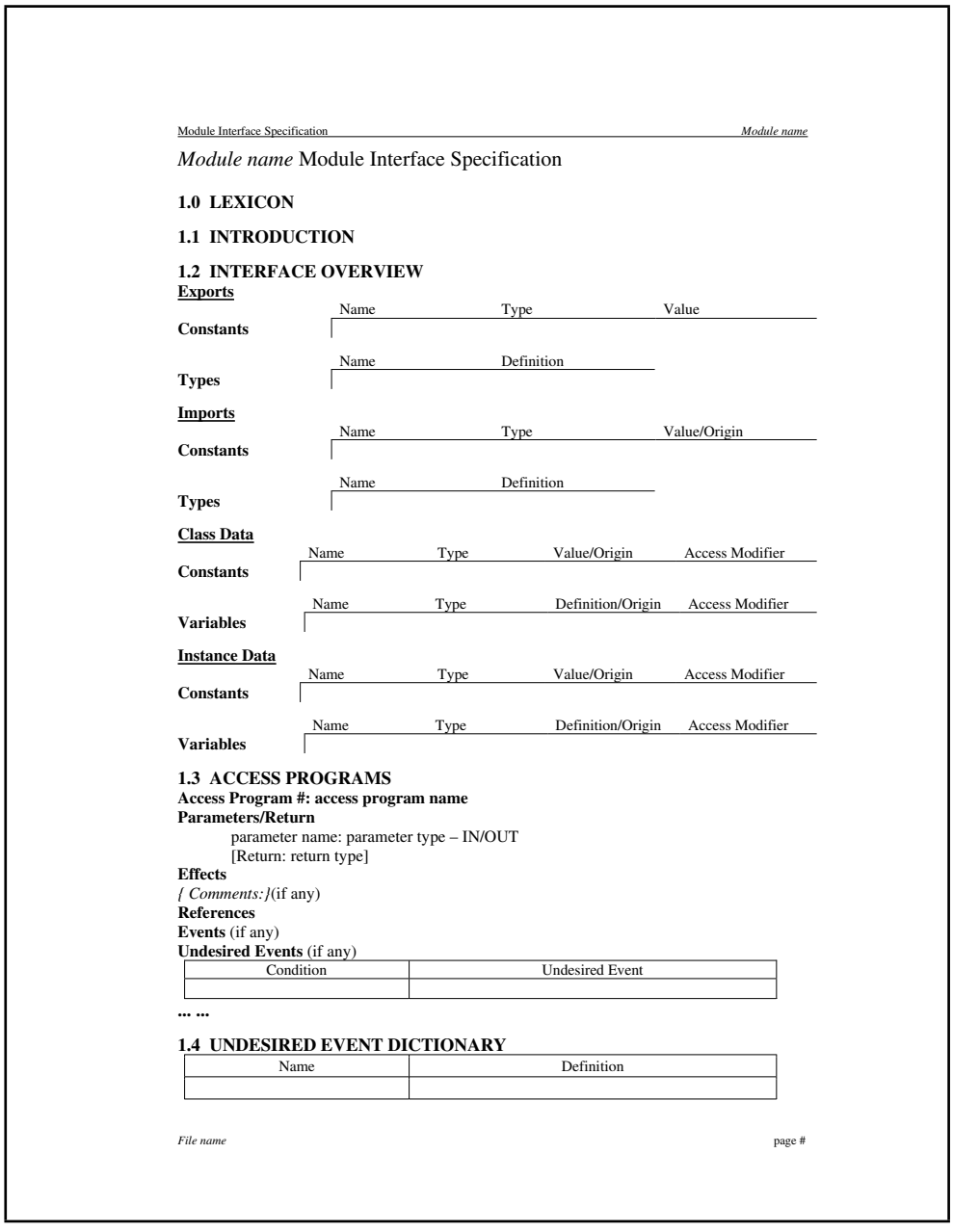
Figure 6.2 shows a template for producing the Module Interface Specifications (MIS) for object-oriented software.

We considered the following when we produced the MIS of our case study module *ScriptGenerationManager*.

Definitions of the Template Headings

- We consider four parts of data in the MIS. *Exports*, *Imports*, *Class Data* and *Instance Data*.

- Basically, both *Exports* and *Imports* have the categories of *Constants*, *Types*, and *Functions* respectively. *Constants* shall be defined by name, type and value (or origin). *Types* shall be specified by name and definition (or origin). *Types* in *Imports* are actually classes, which are normally instantiated locally in the



Module Interface Specification *Module name*

Module name Module Interface Specification

1.0 LEXICON

1.1 INTRODUCTION

1.2 INTERFACE OVERVIEW

Exports

Constants

Name	Type	Value

Types

Name	Definition

Imports

Constants

Name	Type	Value/Origin

Types

Name	Definition

Class Data

Constants

Name	Type	Value/Origin	Access Modifier

Variables

Name	Type	Definition/Origin	Access Modifier

Instance Data

Constants

Name	Type	Value/Origin	Access Modifier

Variables

Name	Type	Definition/Origin	Access Modifier

1.3 ACCESS PROGRAMS

Access Program #: access program name
Parameters/Return
parameter name: parameter type – IN/OUT
[Return: return type]
Effects
{ Comments:}(if any)
References
Events (if any)
Undesired Events (if any)

Condition	Undesired Event

... ...

1.4 UNDESIRED EVENT DICTIONARY

Name	Definition

File name page #

Figure 6.2: Module Interface Specification Template

related invoked methods. *Functions* in *Exports* shall list the access programs of the module and their specification. Since the imported functions are methods of the imported classes, and once a class is imported, all its methods are then also imported, the names of the imported classes should be listed here. For the sake of simplicity, we do not list the invoked methods of these classes.

- Compared to the original MIS documents for procedural-oriented programming, data is now divided into two categories, i.e., *class data* and *instance data*. Such separation is due to a special concept, *class*, in object-oriented languages like Java, where we can think of a class as a *factory* that can create objects.

  Class Data is static and loaded when the class is loaded, describes the data for the class itself, and has one value for all the instances (objects) of the same class.

  Instance Data on the other hand, is only instantiated when an instance of the class is newly created. Hence, the value of instance data is different for different instances.

  Both class data and instance data may have constants and variables.

- Instance data should be declared *private* and other modules can only access the data through the invocation of the public methods.

- The *Access Program* section specifies the program effects of the access programs. *Parameters/Return* defines the parameters and/or return value of the program. Program effects are specified through Program Function Table (PFT). If available, name of the relevant requirements document regarding to this program effect shall be documented in the *Reference*.

- Undesired events are defined in the section of Undesired Events Dictionary.

Considerations On Documenting the Case Study Program

- Normally instance data should be declared *private*, in the case study class, however, most of the instance data are objects, and they are declared public here. This does not violate the principle of *Information Hiding*, instead, it is powerful to do that. Objects encapsulate *data* (also known as *properties* or *fields*) and export *methods* (or *operations* on themselves). The difference between an object and a primitive data type, for instance, *boolean*, is that an object can be *public* while still keeping its data hidden from outside. The reason is, the data of an object is always *private*, only its methods can be *public*. A *public* object only means that its public methods can be invoked in other modules.

- Every class has a *constructor*, and every time when an object of the class is created, it is created by calling the constructor. There is no *de-constructor* in Java, since Java has an automatic garbage collection system to destroy an object when it is no longer in use.

- In the *access programs* section, the effects of a method are specified in *program function tables*.

- The *event* means during the execution of this program, the user may take some actions to activate an event to change the effects. In our example, if the user pushes the *cancel* button, *verificationCancelled* will be set to *true*, and an event is triggered (message written on console) immediately prior to exiting the access program.

- One more condition is added to the effects table to reflect time-determined functionality of access program *addScriptElement*. "WAIT_DURATION" is a constant with the value of 2000 millisecond. If the execution of script element at the *Generation Emulator* is lengthy, after 2000 milliseconds of delay, a popup window will show up with a *cancel* button, providing the user the choice of pressing the *cancel* button to stop the execution and exit the program directly.

6.2.2 How Did We Produce Module Internal Design Documents

Figure 6.3 shows a template for producing the MID for object-oriented software.

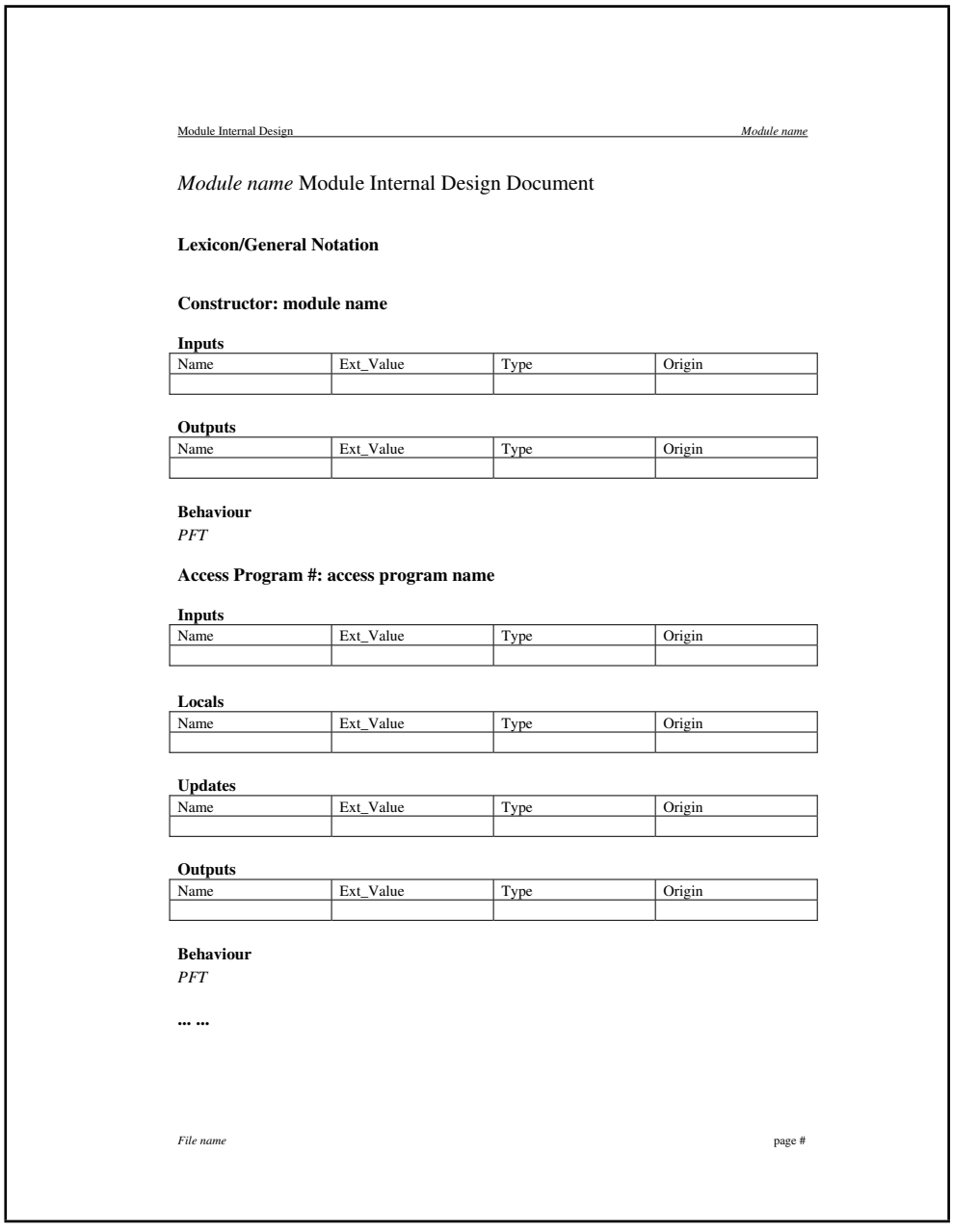
Definitions of the Template Headings

- We use the *inputs*, *outputs*, *updates*, and *behavior* to represent data flow of a module.

- The *inputs*, *outputs* and *updates* have their own *name*, *ext_value*, *type*, and *source*.

- The *name* field is the name of the data flow (parameter name, state data name, or a parameter of another access program). In this field, := means the name on its left is assigned the external value on its right; → denotes that the value of the boolean variable on the left indicates whether or not the external program in the *ext_value* field is invoked. "True" means invoked.

- The *ext_value* field contains an external program name (if any) and its parameter name(s) in parentheses in the form "program_name(param1_name,



Module Internal Design *Module name*

Module name Module Internal Design Document

Lexicon/General Notation

Constructor: module name

Inputs

Name	Ext_Value	Type	Origin

Outputs

Name	Ext_Value	Type	Origin

Behaviour
PFT

Access Program #: access program name

Inputs

Name	Ext_Value	Type	Origin

Locals

Name	Ext_Value	Type	Origin

Updates

Name	Ext_Value	Type	Origin

Outputs

Name	Ext_Value	Type	Origin

Behaviour
PFT

... ...

File name page #

Figure 6.3: Module Internal Design Document Template

param2_name, ...)". The parameters are defined by their own value, type and source if necessary.

- The *source* field shows the source/origin of the data flow:

  1. If the data flow represents a global variable, the module name in which the global variable is defined is entered.
  2. If the data flow represents a program call from another access program, then "Call" is entered.
  3. If the data flow represents a parameter from another access program, the module name in which the access program is defined is entered.
  4. If the data flow represents a parameter (or return value) belonging to the access program itself, then "Param" (or "Func") is entered.
  5. If the data flow represents a local value from (or to) another design block of the access program, "Local" is entered.

- The effects of an access program are represented through a PFT in the *Behaviour* section. There are some alternative formats of program function tables. We use the Condition Table with the condition listing all the conditions. and result listing values of all the output variables for each condition. A normal function table with the headers representing the conditions and the grids showing the effects of an access program can also be used.

Considerations On Documenting the Case Study Program

- An Object-Oriented program consists of one or more objects that interact with one another to solve a problem. Such interaction is implemented through calls

to methods (or access programs). There are two choices for us to document the effect of an access program: one is to trace down the invocation path and describe the eventual change that will be made to this program. An alternative choice is to just describe the calls to the external programs in terms of the variables inside this access program itself.

In this case study we made the second choice. We treat the calls as *external*, and document the calls to the external programs directly using the variables in the calling program. The reason lies in the particular traits of OOP languages. Objects in object-oriented programming can be considered as variables and constants in procedure-oriented programming, while classes correspond to types. An object is different from a normal variable in that, it can be *global* through method calls to its public methods. That is why we can regard the object external and leave the actual effects of the method calls to the specification of the type of the object(class). Since the effects of these invocations can be found in their corresponding modules, we only document one thing in one place. The other choice could land us in trouble in two ways: (1) the invocation chain can get very long and the relations become horribly complex, and (2) because of this, the reader has difficulty understanding the effect of that program.

- All the outputs are variables that have defined values. Therefore we define a call using a *boolean* variable with the value of either *True* or *False*, which implies the invocation of that call or not.

  The name field is the name of the data flow (parameter name, state data name, or a parameter of another access program). In this field, := means the name on its left is assigned the external value on its right; $\rightarrow$ denotes

that the *boolean* variable on the left determines whether or not the external program in the external field is invoked. *True* means invoked. For example, $mes \rightarrow messenger.displayErrorMessage(msgTitle, msg)$ means the program "messenger.displayErrorMessage(msgTitle, msg)" is invoked if "mes" is true.

- How to specify the intermediate states of a program execution, i.e., to document what happens during the run time instead of write time, was a problem to us. In most cases, function tables describe the complete behavior within a *program block*. This includes *loops* within the block. In a few cases, it is important to describe the loop itself. For example, if an output is generated within the loop body and the output is different for each iteration of the loop. [2] presents a way to document a *while* loop by creating separate displays for both the loop body and the overall loop construction. In this work, we try to use one table to specify the complete loop behavior. As we explained in Chapter 4, a *while* table was used.

- Inside any particular access program, it is assumed that all *get* programs are invoked before any processing is performed. The sequence of invocations is not significant. Similarly, all *set* programs are invoked after all processing is performed. Unless otherwise indicated, the sequence of invocations is not significant. If the sequence is important, the order of invocations is indicated by integers, *1, 2, 3, etc.* A program labelled n is invoked before one labelled $n+1$. An indication of "FINAL" indicates that the program shall be the last one invoked in the access program.

Sample MID Documents for While Tables

In this section, we introduce in more detail how to document program behavior through the MID. We show the documentation through the example of the while loop table as discussed in Figure 4.1 in Chapter 4.

We explained in Chapter 4 that a *while* table was created to cope with the problem of documenting non-deterministic programs introduced by *while* loops. We also suggested some new notation through an example in Figure 4.1, and provided the specification of that example in Table 4.2. The question is, how to document such behavior in the MID? When we looked at Table 4.2, the headers looked a little bit complicated. This becomes worse when a table gets bigger and the number of its columns increases, which is common in a real-world application. That is why we made some changes inside the behavior table.

Table 6.2 illustrates the MID produced based on the above example. First, we need to find out the data flow of that program. We have an input, *cond()*, which returns a boolean value from a method call outside of this module. In fact, *cond()* is only an input to the while loop, not to the whole program. Therefore we put it into the category of local variables named "Locals". We also have an output, *outValue.show()*, which is also an invocation that will print some numbers on the screen. To simplify the program behavior table, we specify both the input and the output with boolean values, and use only those boolean variables inside the table. Here, $c <>$ is a boolean variable with a tag, whose external value is $cond()$; and $ov <>$ is also a tagged boolean variable which implies the invocation of the method $outValue.show()$. Since the loop termination condition may change according to the values of the variables, we use another boolean variable, named "Loop Terminates",

	name	ext_value	type	source
Inputs:	-			
Locals:	c<n> : =	cond()	Boolean	Call
Outputs:	ov<n> →	outValue.show()	Boolean	Call

Behaviour:

n = 0, 1, 2, ...	ov<n>	Loop Terminates
c<n>	False	True
NOT (c<n>)	True	False

Table 6.2: Specifications of the While Loop Example in MID

to represent the value of the termination condition.

This way, we can specify the design of a program with detailed data flow and program behavior through the use of MID, rigorously yet understandably.

6.3 Discussion on the Case Study Project

In this section we will discuss our findings and concerns discovered from the examination of the code and the documents of our case study during the preparation for our documentation task.

6.3.1 Findings

Incomplete Documentation of Behavior

The documents we had access to from MDR include an API document describing the functionalities of the classes in the RTR system. This API document, as the only

interface specification available, turned out to be vague and incomplete in specifying the program behavior of the module we examined.

As an example, let us look at a method called *addScriptElement* of the class *ScriptGenerationManager*. From the code, we can see several program effects, but its API document does not describe all of them. Such a lack of rigor is not trivial for a critical software system. Table 6.3 lists the comparison of the coverage of program behavior (effects) between the current API document from the customer and our improved interface specification document (MIS) concerning this example.

Program Effects	***API Document***	***MIS Specification***
add an element	YES	YES
execute the element to verify its actual effects against expected	YES	YES
specify the cases when the element is added to the script main body or a segment	NO	YES
display script preview window after the adding of the element	NO	YES
draw a graphical representation of the element after the addition	NO	YES

Table 6.3: An Example of Comparison of Program Effects Specification Coverage

Vague Document

Another example from the method *addScriptElement*: As the API document specifies, the function of this method is, "Adds an element to the script. If *verify* is true, the element is executed by the generation emulator to verify it." Our question was, what

will happen if *verify* is not *true*? Does this parameter have to be set *true*? We could not find the answers from this API document. What we can do is to check the code and confirm that, the script element will be added no matter what the value of *verify* is. So we had to communicate with the designer to confirm the actual and complete functionality and behavior of this method. "Verify" is normally set to *true* while a special case would have it set to *false*. Setting *verify* to false is used when automatically adding scripts element by *autosampling* trajectories from the user input devices.

A specification document should specify all the cases explicitly, strictly, and correctly. Our improved documentation avoids such confusion through the use of tables to list all the cases. The reader can refer to the appendix for details of our documentation.

Discrepancy

Several discrepancies between the API documents and the code were also found. Discrepancies not only increase our work load, but also tend to cause confusion and misunderstanding. We had to go into both the relevant documents and the codes to confirm their conformance, while some of them require the reader to struggle through a long invocation path to completely understand the code.

1. In the code there is a method named "isModeSupported(byte)", but we could not find this method in the API document. There is only a similar method named "isModelCorrectionSupported" in the same class. But are they the same methods that are just caused by a typo error, or not? Again we had to clarify this with the original programmer. In fact, these are two different methods. The "isModelCorrectionSupported" was already depreciated and replaced by

the new method "isModeSupported". The document was wrong since it did not update this information. Now the related API document has been updated based on our feedback.

2. According to the API, the method *addScriptElement* should "return TRUE if the script element was added and verified". but based on the actual code, the method is not a *boolean* in the first place. This is another case in which the programmer did some changes on the actual implementation, but forgot to update the related documentation. The return value is actually replaced by two exceptions since the exceptions can provide more information to the user.

3. Another kind of discrepancy concerns the behavior description between the code and the relevant API document. That is, the actual code and the API document may exhibit different effects on one program. For instance, the method "cancel". The document says that it only cancels the verification of script element execution but does not cancel the addition of the element. But the actual code does cancel the addition anyway. We certainly have to follow what the code really does.

4. There are several unimplemented methods in the code. In contrast, the related API documents still provide the description of their (intended) functionality. Some of them might need to be implemented in the future but others might not. For instance, the method "elementComplete" is actual an empty method. It is again a case that the programmer did not clean up code after a design change. These problems were automatically generated from the ROSA model based on the original design from *UML*. Tools are generally powerful in helping design, however, inspections are still needed to keep the documents compliant

with the code.

Loose definition

There are several loose definitions of terminology in the documents. In a mission critical system, all the definitions must be strict and precise. A loose definition may impose a hidden risk to the execution of the whole system, and such risk might cause unexpected danger.

- Definition of class *ScriptSyntaxException*

  The definition of this class from the API document is: "An exception thrown by the recipient of a syntactically incorrect script, segment, or element". But what again is the definition of "syntactically incorrect"? We could not find any in the documents on hand. Even after we tried to clarify those definitions with MDR, we could not find the answer, since there is no strict definition but a loose one for terms like "syntactically". The reason is, the system is intended to support an arbitrary script language and arbitrary robotics. The precise definition of what is considered a script syntax exception is left to the discretion of the *script language* designer. It could include things such as a script that has an 'else' without an 'if', or a command that was issued with too few parameters, or a command that was issued with a wrongly-typed parameter. The problem is, in this case, we cannot specify the precise and complete behavior of such a class and must wait for the documentation of other classes.

- Definition of class *UnsupportedScriptElementException*

  Similar to the problem of the above class *ScriptSyntaxException*.

- Completion code

Based on the API, the method *executeScriptElement* will return a "completion code", but there is no precise definition of "completion code" either. According to the original designer, this is a concept not entirely "fleshed out". Because the system is meant to be extendable to any script language and executor, the designer could not define what return codes were errors at the time of the design. The intent was to check the completion code that is returned, but what he actually did was to count on an exception, say, ScriptSyntaxException, being thrown if an error occurs. We think the designer needs to denote this background information, and gives the precise definition of "complete code" later on so that the user would know what to expect for the return value.

- Concepts of *keyframe* and *waypoint*

  Several terms like *keyframe* or *waypoint* are found in many places in the document, but without any definition of these terms. Thus the reader may be confused as to what they actually are. Are they the same as script element, or only one type of script element? And what is the difference between them? The fact is, this is a case of terminology migration. In a previous MDR project, there was only one type of script element which was called *keyframes*. *keyframes* were defined as positions of the arm defined by one set of joint angles. A *waypoint* is similarly a tip position in the script. The later project, i.e., the project we are using, expanded the script language to contain non-position commands. Therefore "script element" is now used as a general term. The reason why the previous terms were not eliminated from documentation might be they are still useful to some readers. However, mixed terms decreases the readability of the documents and may introduce unnecessary misunderstandings.

Misuse of the Terminology

The *emulator.executeScriptElement(element)* is one of the invoked methods, and its functionality is documented as "to execute the script element for verification". When we looked at the word "verification", a normal meaning that we can think of is, to examine through some tools or based on some criteria to make sure that the element would perform an intended behavior given a proper condition, and such kind of examination should be quantitative. However, the actual verification implemented is just a manual and qualitative task. Currently the only "verification" is a visual inspection by the user to make sure the manipulator does as expected according to the script element command. The other part of verification is to make sure the emulator does not return a *ScriptSyntaxException* or *UnsupportedScriptElementException*.

If such explanation does not come with this terminology, confusion may arise. So our suggestion is to use another word here, say, *visual examination*.

Need-to-improve Capability

We had a question concerning the method "GenerationEmulator.executeScriptElement" on how does the user show his/her satisfaction for the verification result of a script element during its execution. The problem is, there is only a variable called *verificationCancelled* to stop the verification in case of a lengthy wait. Otherwise, once the script element is sent to the emulator and no exception or cancelling action found, the script is supposed to be correct and thus added anyway. Our question is, can the user change the script element based on the simulation result? Or, if the user is not happy with the result, does it allow the user to remove or edit the script element? In the current version we got for the ROSA

project, as we confirmed with the MDR, this capability is sadly not implemented. We think this needs to be improved in the later version.

6.3.2 Concerns

Design and Documentation of Abstract Interfaces

We have introduced the concept of an interface of a module in previous chapters, which can be treated as a black box for communications between this module and others. Abstraction is one of the principles for designing a module interface, which is also a difficult one to apply in practice.

Based on [3], an *abstract interface* is defined as an abstraction that represents more than one interface, that is, it consists of the assumptions that are included in all of the interfaces that it represents. A further idea from Parnas is, it might be helpful to divide an interface into two parts, an upper face and a lower face. The upper face contains only the information that the user of a module (class, package) needs to know, while the lower face is made up of the assumptions that the programs (methods) in that module make about the programs (methods) that they use. Ideally, the upper face can be described without referring to the lower face. The user of the module can predict its behavior without having to know anything about the programs used in the implementation of that module.

As for our case study program, the class of ScriptGenerationManager, as described previously in this chapter, lives under a generic system whose further implementation can be tailored under different projects. This class contains many methods whose definition can only be found outside of the class, that is, the only definition of what happens on the upper face is a description of the lower face. This is because the pro-

gram was written to work with a lot of different lower faces and the behavior is really described in terms of the underlying mechanisms. For instance, what constitutes a correct script will vary with the capabilities of the underlying robotics equipment. Hence, the behavior of the upper face for, say, a generic script module, cannot be defined completely without references to its lower face. This represents one of the three cardinal principles of object-oriented programming, polymorphism. Polymorphism has a weighed beneficial factor of code reuse and program flexibility. However, how to document programs with such features is still a challenge to us.

6.4 Case Study Conclusion

This case study provided sample documentation for a critical part of a critical software system. Due to document unavailability, we could not produce a sample Module Guide for the whole system. Only the Module Interface Specification and the Module Internal Design Document for the example class "ScriptGenerationManager" were produced. We use tabular expressions to represent the interface and behavior of the example in a precise and complete way. These documents are able to serve as a basis for the next step of the inspection through a document driven approach.

Our documentation successfully specifies the data flow and program effects through a formal and strict notation which is, at the same time, readable and understandable by software practitioners. It also proposes an approach through the use of *tables* to overcome difficulties and problems on documentation brought by some special attributes of OO code, for example, *delocalization*. We are confident that the behavior of the program is now documented precisely and thoroughly enough to conduct a successive inspection.

Chapter 7

Conclusions and Future Work

This chapter summarizes this thesis, draws conclusions, and makes suggestions for future work.

7.1 Summary

In this thesis, we have discussed the *Document Driven Inspection* approach from the aspect of object-orientation, investigated issues that may arise due to special aspects of OO design and code during the documentation phase of an inspection, and illustrated this approach through a real-world case study.

This is the first time (to the best of our knowledge) that the DDI approach has been applied to an object-oriented software application from industry. The object-orientation paradigm is now widely accepted in industry because of its many benefits. However, the power and benefits may lead to difficulties and problems in the inspection. In this work, we developed an approach to deal with some of the difficulties, for instance, *delocalization* and *inheritance*, which was discussed in Section 4.4.

We explored and discussed the related work in both the traditional inspection field and some new techniques focused on the inspection of OO design and code in Chapter 2 and Chapter 3. We then introduced the Document Driven Inspection approach and further discussed the specific techniques and process for conducting an inspection as well as a documentation in Chapter 4. We felt that MIS/MID did a better job than *Displays* in documenting our case study program, since the displays can do a good job on a long program but not a program with lots of "delocalized" references. Instead, using MIS/MID, we can treat all the delocalization method calls as external, and document the specific behavior of these programs in its own MID.

We also illustrated the applicability of our approach through the production of a series of sample documents for the case study program as described in Chapter 6. Software engineers in industry might have been hesitant to apply our approach to real-world applications, partly because there are few real-world examples that they can refer to. In this work, we provided the templates of both MIS and MID. With the MIS and MIDs of a module, the complete program behavior and data flow can be specified precisely. We use the notation of tabular expressions which is formal, rigorous, and at the same time, readable and understandable by software practitioners. More examples of how to document an OO code can be found in the appendix sample documents, i.e., the MIS and MIDs of the case study program. The considerations that we thought about during the course of documentation were also given so that practitioners can better understand why the documentation should be like this. As an example, the idea of a *while* table was proposed to cope with the specification of the intermediate states of a program execution in cases where the program is non-deterministic.

Some software design principles, including the principle of information hiding, were used in both our documentation approach and the examination of the case study code

and documents. We did the documentation assuming that the original design was built on the information hiding principle. We then used these principles to judge how good the current design of our case study system is. From the examination, we have discovered problems and issues in the case study code and documents. To name but a few, the incomplete documentation of program behavior, and the discrepancy between the current API documents and the actual code. Such problems may not have direct impact on the functionality of the whole system, but made the inspection more difficult, hence would result in inefficient maintenance.

7.2 Conclusions

This work can be concluded by the following observations:

- The *Document Driven Inspection* approach as well as the *tabular expressions* are applicable and helpful to the inspection and/or the documentation of object-oriented critical systems. The tabular notation provides a formal and rigorous means of specifying a program. as discussed in Chapter 4.

- From this work, we have learned that precise documentation should be included in the design phase, instead of being an "afterthought". The use of OO techniques without precise specifications for the classes and methods leads to obscurity and forces people to either reconstruct the abstractions by reverse documentation or to try to read across the abstractions.

- The Document Driven Inspection approach is a technique that relies on rigorous and systematic documentation. The DDI approach is useful both when we have access to adequate requirements and design documents, and also when we do

not have access to such documents. As shown by the application of the DDI approach in our case study in Chapter 6, we illustrated the application of this approach (the first step) through producing a precise and complete document under the circumstance that the requirements and the design documents were not available at the time of documentation. However, it is always better to do the documentation as part of the design than to do it as part of the inspection.

- A series of documents are required for the purpose of precisely specifying and documenting the software, including a Requirements Document, Module Guide, Module Interface Specification and Module Internal Design Document. When an inspection is performed on an existing software system, the Module Guide, MIS and MID would serve well for the comparison of design against the original requirements. The Module Guide is important because it not only shows the hierarchical structure of the system, but also provides a guidance on what modules will be affected by anticipated changes. Since secrets of the modules are a virtual aspect of the Module Guide, it would be preferable to have the Module Guide prepared by the software designers. We did not prepare one for this case study because we were working on one sample module (class) extracted from a big system. In our case study, the sample MIS and MID produced illustrate the effectiveness of our approach when it is applied to documenting the behavior of an OO code.

- The *while* table introduced in Section 4.1.3 is proposed to document non-deterministic programs, in cases that such non-determinism is introduced through a while loop. The notation appears to document this type of behavior more efficiently than an output sequence.

7.3 Future Work

More work and improvements are always necessary in the future to continue this work. Some of them are listed below.

1. A Module Guide sample. A Module Guide serves to give an overview of the modular structure of the system. Trying to develop a module guide after the fact almost never succeeds because the modules have no secrets. Producing the module guide is a step in the design that is designed to assure that modules do have secrets. We have explained why we did not not prepare one for this case study in the previous section. However, it may still be useful to develop one as part of the Inspection Documentation. Such a Module Guide can show the structure of the design, and the services provided by each module. Inclusion of "secrets" is more problematic, since without the original requirements, and without the software designers' understanding of what design decisions were likely to change, all "hidden" entities are likely to be classified as secrets – even if they were not. One of the goals of developing a Module Guide, after the fact, would be to help judge how well information hiding was applied during the software design process.

2. A continuing inspection based on the documentation produced. As mentioned, this work is only the first step in the inspection process. The inspection would also help to evaluate the work of our documentation.

3. More examples that facilitate the documentation of programs with more OO traits, for instance, polymorphism and dynamic binding, using the Document Driven Inspection approach. Polymorphism and dynamic binding are very im-

portant concepts in OOP, but due to the limitation of our case study, we did not have an opportunity to deal with them using our approach.

4. Automation. Our documentation of this case study was done manually, however, automated tool support is desirable in that it can accomplish much of the documentation that is lengthy and thus save human efforts that are tedious and error-prone. With the tool assistance, the inspection as well as the documentation can be conducted in a more efficient way while preserving the correctness.

Bibliography

[1] "IEEE Standard for Software Reviews," 1998. IEEE std 1028-1997.

[2] B. Bauer, "Documenting complicated programs," Tech. Rep. CRL316, McMaster University, 1995.

[3] K. H. Britton, R. A. Parker, and D. L. Parnas, "A procedure for designing abstract interfaces for device interface modules," *Proceedings of the Fifth International Conference on Software Engineering*, pp. 195–204, March 1981. reprinted as Chapter 15 in [19].

[4] K. H. Britton and D. L. Parnas, "A-7E Software Module Guide," NRL 4702, United States Navel Research Laboratory, December 1981. Memorandum Report.

[5] F. P. Brooks, "No silver bullet: Essence and accidents of software engineering," *Computer*, vol. 20, no. 4, April 1987.

[6] O.-J. Dahl, E. Dijkstra, and C. Hoare, *Structured programming.* Academic Press, 1972.

[7] E. Dijkstra, "The structure of the 'THE' - multiprogramming system," *Commun. ACM*, vol. 11, pp. 341–346, 1968.

[8] A. Dunsmore, M. Roper, and M. Wood, "Object-Oriented Inspection in the face of Delocalisation," *Proceedings of the 22nd international conference on Software engineering*, June 2000.

[9] A. Dunsmore, M. Roper, and M. Wood, "Practical code inspection for object-oriented systems," *WISE'01: Proceedings of the 1st Workshop on Inspection in Software Engineering*, July 2001.

[10] A. Dunsmore, M. Roper, and M. Wood, "The development and evaluation of three diverse techniques for object-oriented code inspection," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, August 2003.

[11] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, 1976.

[12] M. E. Fagan, "Advances in Software Inspections," *IEEE Transactions On Software Engineering*, vol. 12, no. 7, July 1986.

[13] M. E. Fagan, "A History of Software Inspections," *Software Design and Management Conference, Software Pioneers*, 2001. Eds.: M. Broy, E. Denert, Springer, 2002.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[15] T. Gilb, "Optimizing Software Inspections," *The Journal of Defense Software Engineering*, March 1998.

[16] T. Gilb, "Software Inspections are not for quality, but for engineering economics," *IEEE Software on Inspection*, 1999.

[17] T. Gilb and D. Graham, *Software Inspection*. Addison-Wesley, June 1993.

[18] R. Gillett, M. Greenspan, L. Hartman, E. Dupuis, and D. Terzopoulos, "Remote Operation with Supervised Autonomy (ROSA)," in *Proceeding of the 6th International Symposium on Artificial Intelligence and Robotics & Automation in Space: i-SAIRAS 2001*, (Canadian Space Agency, St-Hubert, Quebec, Canada), June 2001.

[19] D. M. Hoffman and D. M. Weiss, eds., *Software Fundamentals, Collected Papers by David L. Parnas*. Addison-Wesley Publishing Company, March 2001.

[20] R. Janicki, D. L. Parnas, and J. Zucker, "Tabular representations in relational documents," tech. rep., McMaster University, November 1995. reprinted as Chapter 4 in [19].

[21] R. Janicki, "Tabular Expressions and Their Relational Semantics," SERG 377, McMaster University, Hamilton, Ontario, July 1999.

[22] H. D. M. Linger, Richard C. and B. I. Witt., *Structured Programming: Theory and Practice*. Addison-Wesley, 1979. Reading, Mass.

[23] R. Luciw, "RBC battles to update customer accounts." World Wide Web, `http://www.globetechnology.com/servlet/ArticleNews/TPStory/LAC/20040605/RRBCGLITCH05/TPTechnology/`, June 2004.

[24] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood, "Applying Inspection to Object-Oriented Software," tech. rep., Department of Computer Science, University of Strathclyde, Glasgow, U.K, June 1995. Empirical Foundations of Computer Science (EFoCS).

[25] S. McConnell, "The Best Influences on Software Engineering," *IEEE Software*, January/February 2000.

[26] Michael Fagan Associates, "The Fagan Defect-Free Process." World Wide Web, `http://www.mfagan.com/process_frame.html`, 2003.

[27] H. D. Mills, "The new math of computer programming," *Communications of the ACM*, vol. 18, no. 1, January 1975.

[28] H. D. Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *IEEE Computer*, vol. 21, June 1988.

[29] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, December 1972.

[30] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE*, pp. 264–277, May 1978. Proceedings of the 3rd International Conference on Software Engineering.

[31] D. L. Parnas, "Tabular Representation of Relations," CRL 260, McMaster University, Hamilton, Ontario, October 1992. Telecommunications Research Institute of Ontario.

[32] D. L. Parnas, *Applications of Formal Methods*, ch. 2, Using Mathematical Models in the Inspection of Critical Software. Prentice Hall International Series in Computer Science, 1995. Hinchey M.G., Bowen J.P.(eds.).

[33] D. L. Parnas, *Software Fundamentals: collected papers by David L. Parnas*, ch. 19, Inspection of Safety-Critical Software Using Program-Function Tables. Addison-Wesley, March 2001.

[34] D. L. Parnas, "Design through documentation: The path to software quality." SQRL, University of Limerick, Limerick, Ireland, February 2003.

[35] D. L. Parnas, "Software inspections we can trust." SQRL, University of Limerick, Limerick, Ireland, March 2003.

[36] D. L. Parnas and G. Asmis, "Quality Assurance for Safety-Critical Software," *CIPS '90 Information Technology Conference, Edmonton, Alberta*, October 1990.

[37] D. L. Parnas, G. Asmis, and J. Madey, "Assessment of Safety-Critical Software in Nuclear Power Plants," *Nuclear Safety*, vol. 32, no. 2, April-June 1991.

[38] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 259–266, March 1985. reprinted as Chapter 16 in [19].

[39] D. L. Parnas and J. Madey, "Functional Documents for Computer Systems," CRL 309, McMaster University, Hamilton, Ontario, October 1995.

[40] D. L. Parnas, J. Madey, and M. Iglewski, "Precise Documentation of Well-Structured Programs," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, December 1994.

[41] D. L. Parnas and D. M. Weiss, "Active design reviews: Principles and practices," *Proceedings of the 8th International Conference on Software Engineering, London*, August 1985. reprinted as Chapter 17 in [19].

[42] D. Parnas, "On a 'buzzword': Hierarchical structure," *IFIP Congress 74, North-Holland Publishing Company*, pp. 336–339, 1974.

[43] D. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE2, no. 1, March 1976.

[44] D. Parnas, "Software engineering principles," tech. rep., University of Victoria Report No. DCS-29-ir, February 1983. reprinted as Chapter 13 in [19].

[45] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering Technology and Process*. Addison Wesley Longman, 1999.

[46] E. Sekerinski, September 2004. private correspondence.

[47] F. Shull, F. Lanubile, and V. R. Basili, "Investigating Reading Techniques for Object-Oriented Framework Learning," *IEEE Transactions on Software Engineering*, vol. 26, no. 11, 2000.

[48] F. Shull, G. H. Tracassos, J. Carver, and V. Basili, "Evolving a set of techniques for OO inspections," tech. rep., University of Maryland, MD, USA, 1999.

[49] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, "Designing documentation to compensate for delocalized plans," *Communications of the ACM*, vol. 31, no. 11, pp. 1259–1267, November 1988.

[50] G. H. Tracassos, F. Shull, J. Carver, and V. Basili, "Reading techniques for OO design inspections," tech. rep., University of Maryland, MD, USA, 2002.

[51] A. Wassyng and M. Lawford, "Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project," *FME 2003: International Symposium of Formal Methods Europe Proceedings*, vol. 2805, September 2003.

[52] M. Weiser, "Program slicing," *IEEE on Software Engineering*, vol. SE-10, no. 4, July 1984.

[53] N. Wilde, P. Matthews, and R. Huitt, "Maintaining object-oriented software," *IEEE Software*, vol. 10, no. 1, January 1993.

[54] N. Wirth, "Program development by stepwise refinement," *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, 1971.

Appendix A: Module Interface Specification (MIS) sample document

ScriptGenerationManager (SGM) Module Interface Specification

1.0 LEXICON

A. General Notation

(1) In section 1.2, the super script character (I) in the definition of Exported Types means the type is imported.

(2) In section 1.3, the semicolon " ; " in the effects table of an access program indicates the sequence of actions. For instance, "*a*; *b*; *c*; ... " means action *a* would execute first, then b would execute, and then c, and so forth.

(3) In section 1.3, IN means the parameter is an input; OUT means the parameter is an output; IN/OUT means the parameter is both an input and an output. i.e., an update.

(4) The dot notation follows Java convention, for instance, "script.addScriptElement(element)" means the left hand side of the dot, i.e. "script", is the name of an object, and the right hand side of the dot is the method name of that object.

(5) The "&" and "*OR*" in the condition statements of PFTs in section 1.3 are used to represent the logical relations of the conditions. We use them instead of "$\wedge$" and "$\vee$" because they are found more familiar and thus preferable and understandable in practice.

B. JAVA Language Terminology

public $\stackrel{df}{=}$ a feature that is accessible by methods of all classes

private $\stackrel{df}{=}$ a feature that is accessible only by methods of this class

protected $\stackrel{df}{=}$ a feature that is accessible only by methods of this class, its children, and other classes in the same package

1.1 INTRODUCTION

The ScriptGenerationManager module manages the transactions of the system in script generation mode. The user first commands the virtual equipment (manipulator) to define script elements, and then adds the defined script elements to the current script body until composition of a whole script.

1.2 INTERFACE OVERVIEW

Inheritance

Implements the interfaces of

- ca.mdrobotics.rtr.gs.vdm.SystemModeChangeListener
- ca.mdrobotics.rtr.gs.vdm.gui.CancellableProcess

Exports

Constants

None

Types

Name	Definition
ScriptGenerationManager	(genEmulator: GenerationEmulator$^{(I)}$, fileManager: ScriptFileManager$^{(I)}$, opConsole: OperatorConsole$^{(I)}$)

Imports

Constants

Name	Type	Value/Origin
SCRIPT_GENERATION	byte	ca.mdrobotics.rtr.gs.vdm.SystemModes

Types

Name	Definition/Origin
Autosampler	ca.mdrobotics.rtr.gs.vdm.autosampler.Autosampler
CancellableProcess	ca.mdrobotics.rtr.gs.vdm.gui.CancellableProcess
CommandFrameNode	ca.mdrobotics.rtr.gs.vdm.devices.CommandFrameNode
DeviceNameNotFoundException	ca.mdrobotics.rtr.gs.DeviceNameNotFoundException
ElementIdGenerator	ca.mdrobotics.rtr.gs.script.ElementIdGenerator
GenerationEmulator	ca.mdrobotics.rtr.gs.emulator.generation.GenerationEmulator
GraphicalScriptElement	ca.mdrobotics.rtr.gs.vdm.devices.GraphicalScriptElement
JointNumberOutOfRangeException	ca.mdrobotics.rtr.gs.JointNumberOutOfRangeException
Manipulator	ca.mdrobotics.rtr.gs.vdm.devices.Manipulator
MessagePopup	ca.mdrobotics.rtr.gs.vdm.gui.MessagePopup
ObjectNotFoundException	ca.mdrobotics.rtr.gs.vdm.devices.ObjectNotFoundException
OperatorConsole	ca.mdrobotics.rtr.gs.vdm.gui.OperatorConsole
ProjectFactory	ca.mdrobotics.rtr.gs.ProjectFactory
ReferenceFrame	ca.mdrobotics.rtr.gs.vdm.devices.ReferenceFrame
Script	ca.mdrobotics.rtr.gs.script.Script
ScriptElement	ca.mdrobotics.rtr.gs.script.ScriptElement
ScriptGenerationConsole	ca.mdrobotics.rtr.gs.vdm.gui.ScriptGenerationConsole
ScriptLanguage	ca.mdrobotics.rtr.gs.script.ScriptLanguage
ScriptSegment	ca.mdrobotics.rtr.gs.script.ScriptSegment
ScriptSyntaxException	ca.mdrobotics.rtr.gs.script.ScriptSyntaxException
SegmentCall	ca.mdrobotics.rtr.gs.script.SegmentCall
UnsupportedOperationException	ca.mdrobotics.rtr.gs.script.UnspupportedScriptElementException
ValueOutOfRangeException	ca.mdrobotics.rtr.gs.ValueOutOfRangeException
VirtualWorld	ca.mdrobotics.rtr.gs.vdm.devices.VirtualWorld
WaitOrCancelPopup	ca.mdrobotics.rtr.gs.vdm.gui.WaitOrCancelPopup

Class Data

Constants

Name	Type	Value	Description	Access Modifier
WAIT_DURATION	int	2000	Delay in milliseconds used in "addScriptElement"	private

Variables
None

Instance Data

Constants

Name	Type	Definition/Origin	Access Modifier
idGenerator	ElementIdGenerator	ca.mdrobotics.rtr.gs.script.ElementIdGenerator	protected

Variables

Name	Type	Definition/Origin	Access Modifier
autosampler	Autosampler	ca.mdrobotics.rtr.gs.vdm.autosampler.Autosampler	protected
cancelPopup	WaitOrCancelPopup	ca.mdrobotics.rtr.gs.vdm.gui.WaitOrCancelPopup	protected
console	ScriptGenerationConsole	ca.mdrobotics.rtr.gs.vdm.gui.ScriptGenerationConsole	public
currentSegment	ScriptSegment	ca.mdrobotics.rtr.gs.script.ScriptSegment	protected
emulator	GenerationEmulator	ca.mdrobotics.rtr.gs.emulator.generation.GenerationEmulator	protected
messenger	MessagePopup	ca.mdrobotics.rtr.gs.vdm.gui.MessagePopup	protected
selectedManipulator	Manipulator	ca.mdrobotics.rtr.gs.vdm.devices.Manipulator	protected
theOpConsole	OperatorConsole	ca.mdrobotics.rtr.gs.vdm.gui.OperatorConsole	protected
theScriptFileManager	ScriptFileManager	ca.mdrobotics.rtr.gs.vdm.ScriptFileManager	protected
verificationCancelled	boolean	False	private

1.3 ACCESS PROGRAMS

Constructor:

ScriptGenerationManager(GenerationEmulator genEmulator, ScriptFileManager fileManager, OperatorConsole opConsole)

Effects

scriptGenerationManager: ScriptGenerationManager

Comments:

{ Creates a new instance of ScriptGenerationManager, and also the instances of its state data, including theScriptFileManager, theOpConsole, emulator, idGenerator, aotosampler, messenger, cancelPopup. }

Access Program 0: initialise

Parameters/Return

none

Effects

Condition	Effects
ProjectFactory. isModeSupported(SCRIPT_GENERATION)	a new instance of state data "console" is created
NOT (ProjectFactory. isModeSupported(SCRIPT_GENERATION))	-

Access Program 1: addScriptElement

Parameters/Return

element: ScriptElement – IN

The new script element to be added

verify: boolean – IN

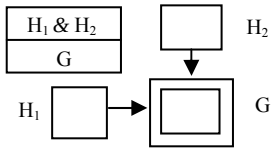
Indicates that the element needs to be executed by the emulator

Local Variables

Name	Type	Value
script	Script	theScriptFileManager.getScript()
graphicalElement	GraphicalScriptElement	element.createGraphicalScriptElement()

Reference: UC5.5, UC5.6, UC5.7.1.3

Effects



		NOT(verify)	Verify	
			ElemExecTime <= WAIT_DURATION *OR* NOT(verificationCancelled)	ElemExecTime > WAIT_DURATION & verificationCancelled
currentSegment = null	graphicalElement = null	script.addScriptElement(element); theScriptFileManager.refreshScriptPreview()	emulator.executeScriptElement(element); script.addScriptElement(element); theScriptFileManager.refreshScriptPreview()	theOpConsole.write("Verification cancelled")
	graphicalElement ≠ null	script.addScriptElement(element); theScriptFileManager.refreshScriptPreview(); VirtualWorld.addGraphicalScriptElement(graphicalElement)	emulator.executeScriptElement(element); script.addScriptElement(element); theScriptFileManager.refreshScriptPreview(); VirtualWorld.addGraphicalScriptElement(graphicalElement)	
currentSegment ≠ null	graphicalElement = null	currentSegment.addScriptElement(element); theScriptFileManager.refreshScriptPreview()	emulator.executeScriptElement(element); currentSegment.addScriptElement(element); theScriptFileManager.refreshScriptPreview()	
	graphicalElement ≠ null	currentSegment.addScriptElement(element); theScriptFileManager.refreshScriptPreview(); VirtualWorld.addGraphicalScriptElement(graphicalElement)	emulator.executeScriptElement(element); currentSegment.addScriptElement(element); theScriptFileManager.refreshScriptPreview(); VirtualWorld.addGraphicalScriptElement(graphicalElement)	

ElemExecTime = $t_{now} - t_0$, where t_{now} = current time,

t_0 = time when emulator.executeScriptElement(element) is invoked

{ Comments:
Adds the element to the current non-null script segment or script main body. Before adding, depending on the value of input verify, the element may be executed in a specified emulator to see if the element is supported by the emulator. If the verification is lengthy, a cancel button will appear so that the user may choose to skip out of the program directly. Once a script element is added, a graphical representation of this element is added to the virtual scene on the screen }

Events

@T(cancel) => verificationCancelled := True

{ @T(cancel) denotes the cancel button is pressed by the user during the execution of "element" }

Undesired Events

Condition	Undesired event
(script is null) *OR* (script is incorrect syntactically based on definition in ScriptLanguage class)	ScriptSyntaxException
the device emulator does not support the execution of element	UnsupportedScriptElementException

Access Program 2: cancel

** Implements CancellableProcess*

Parameters/Return

none

Effects

verificationCancelled := True;
ScriptGenerationManager.gotoStandby()

{ Comments:
Cancels the verification by setting the value of state data "verificationCancelled" to "true", and places the generation emulator in standby }

Access Program 3: getCurrentManipulator

Parameters/Return

Return: Manipulator - OUT

Effects

getCurrentManipulator := selectedManipulator

Undesired Events

none

Access Program 4: setCurrentManipulator

Parameters/Return

manipulator: Manipulator - IN

Effects

selectedManipulator := manipulator

{ Comments:

Sets the virtual manipulator under control by the user }

Undesired Events

none

Access Program 5: setCurrentManipulatorJoints

Parameters/Return

jointVector: double[] - IN

Effects

emulator.setManipulatorJoints(selectedManipulator, jointVector)

Reference: UC5.3.2

{ Comments:

Commands the virtual manipulator to move immediately to the "jointVector" }

Undesired Events

JointNumberOutOfRangeException

ValueOutOfRangeException

Access Program 6: moveCurrentManipulatorJointsTo

Parameters/Return

jointVector: double[] - IN

Effects

emulator.moveManipulatorJointsTo(selectedManipulator, jointVector)

Reference: UC5.3.2

{ Comments:

Moves the currentManipulator to the new "jointVector". The manipulator is moved from

its current position to its new position following the motion planning algorithm supplied by the Generation Emulator }

Undesired Events

JointNumberOutOfRangeException

ValueOutOfRangeException

Access Program 7: enableSJRM

Parameters/Return

joint: int - IN

enable: boolean - IN

Effects

emulator.enableSJRM(selectedManipulator, joint, enable)

{ Comments:
Enables single joint rate mode (SJRM) for the selected manipulator }

Undesired Events

none

Access Program 8: enableMAM

Parameters/Return

enable: boolean - IN

Effects

emulator.enableMAM(selectedManipulator, enable)

{ Comments:
Enables manual augmented mode (MAM) for the selected manipulator }

Undesired Events

none

Access Program 9: gotoStandby

Parameters/Return

none

Effects

emulator.gotoStandby()

{ Comments:
Cancels the movement of the generation emulator by putting it in standby }

Undesired Events

none

Access Program 10: newScript

Parameters/Return

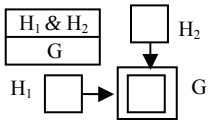
none

Inputs from screen user input

goAhead: boolean – IN

name: String – IN

Effects

<table>
<tr><td rowspan="2"> 
</td><td rowspan="2">theScriptFileManager.getScript() = null</td><td colspan="2">theScriptFileManager.getScript() ≠ null</td></tr>
<tr><td>goAhead</td><td>NOT(goAhead)</td></tr>
<tr><td>validName</td><td colspan="2">theScriptFileManager.newScript(name);
theScriptFileManager.refreshScriptPreview()</td><td rowspan="2">No action</td></tr>
<tr><td>NOT(validName)</td><td colspan="2">No action</td></tr>
</table>

Reference: UC5.1.2

{ Comments:

Creates a new script with a valid name, prompting before erasing any existing script. A valid name is determined by method "isScriptNameValid" in class ScriptLanguage }

Undesired Events

none

Access Program 11: newScriptSegment

Parameters/Return

Return: boolean – OUT

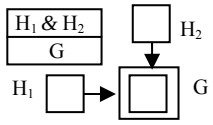
Local Variables

Name	Type	Value
script	Script	theScriptFileManager.getScript()

Inputs from screen user input

segmentName: String – IN

Effects

<table>
<tr><td rowspan="2"> 
</td><td>script ≠ null</td><td>script = null</td></tr>
<tr><td>segmentName ≠ null</td><td>script.addScriptSegment(segmentName);
theScriptFileManager.efreshScriptPreview();
return True</td><td rowspan="2">return False</td></tr>
<tr><td>segmentName = null</td><td>theScriptFileManager.refreshScriptPreview();
return False</td></tr>
</table>

{ Comments:
Creates a new script segment with name within the current script, and returns true if the segment was created and added to the script. Return false message in case there is no script open, i.e., the current script is null }

Undesired Events

none

Access Program 12: closeScriptSegment

Parameters/Return

none

Effects

currentSegment := null

Undesired Events

none

Access Program 13: setCurrentSegment

Parameters/Return

newCurrent: ScriptSegment - IN

Effects

currentSegment := newCurrent

{ Comments:
Sets the current script segment to the "newCurrent" if the newCurrent exists in the script and otherwise the current script segment is null }

Undesired Events

none

Access Program 14: getAllScriptSegments

Parameters/Return

Return: ScriptSegment[] - OUT

Effects

getAllScriptSegments := theScriptFileManager.getScript().getScriptSegment()

Undesired Events

none

Access Program 15: getCurrentScriptSegment

Parameters/Return

Return: ScriptSegment - OUT

Effects

getCurrentScriptSegment := currentSegment

Undesired Events

none

Access Program 16: getScriptName

Parameters/Return

Return: String - OUT

Effects

getScriptName := theScriptFileManager.getScript().getName()

{ Comments:
Returns the name of the current script or the default "null" if there is no current script being set }

Undesired Events

none

Access Program 17: setCurrentManipulatorPOR

Parameters/Return

newPOR: ReferenceFrame - IN

Effects

emulator.setManipulatorPOR(selectedManipulator, newPOR)

Reference: UC5.3.1

{ Comments:
Commands the virtual manipulator to move immediately to the "newPOR" }

Undesired Events

ValueOutOfRangeException

Access Program 18: moveCurrentManipulatorPORTo

Parameters/Return

newPOR: ReferenceFrame - IN

Effects

emulator.moveManipulatorPORTo(selectedManipulator, newPOR)

Reference: UC5.3.1

{ Comments:
Moves the currentManipulator to the "newPOR". The manipulator is moved from its current position to its new position following the motion planning algorithm supplied by the Generation Emulator }

Undesired Events

ValueOutOfRangeException

Access Program 19: setCurrentManipulatorCommandFrame

Parameters/Return

newCommandFrameNode: CommandFrameNode - IN

Effects

emulator.setManipulatorCommandFrame(selectedManipulator, newCommandFrameNode)

{ Comments:
Displays the graphical representation of the command frame at the position and orientation of "newCommandFrameNode" and instructs the emulator that the command frame has changed for the selected manipulator }

Undesired Events

none

Access Program 20: getCurrentManipulatorCommandFrame

Parameters/Return

Return: CommandFrameNode - OUT

Effects

Condition	Effects
selectedManipulator ≠ null	selectedManipulator.getCommandFrame()
selectedManipulator = null	VirtualWorld.getBaseCommandFrame()

Undesired Events

none

Access Program 21: modeChanged

** Implements SystemModeChangeListener*

Parameters/Return

newMode: byte - IN

Effects

Condition	Effects
newMode = SystemModes.(SCRIPT_GENERATION)	theOpConsole_write("Script Generation Mode"); console.setVisible(True)
NOT(newMode = SystemModes.(SCRIPT_GENERATION))	console.setVisible(False)

{ Comments:
Displays the script generation console if the input parameter is SCRIPT_GENERATION, hides the script generation console otherwise }

Undesired Events

none

Access Program 22: configueAutosampler

Parameters/Return

none

Effects

autosampler.configure(console)

Reference: UC5.7.1.8

{ Comments:
Launches the autosampler configuration dialog }

Undesired Events

none

Access Program 23: enableAutosampling

Parameters/Return

doEnable: boolean - IN

Return: boolean - OUT

Effects

autosampler.enable(doEnable)

enableAutosampling := doEnable

Reference: UC5.7.1.8

{ Comments:

Enables or disables autosampling, return true if enabled }

Undesired Events

ScriptSyntaxException

1.4 UNDESIRED EVENT DICTIONARY

Name	Definition
DeviceNameNotFoundException	ca.mdrobotics.rtr.gs.DeviceNameNotFoundException
IllegalStateException	(cannot find in the document)
Java.io.IOException	Java API document
JointNumberOutOfRangeException	ca.mdrobotics.rtr.gs.JointNumberOutOfRangeException
ObjectNotFoundException	ca.mdrobotics.rtr.gs.vdm.devices.ObjectNotFoundException
ScriptSyntaxException	ca.mdrobotics.rtr.gs.script.ScriptSyntaxException
UnsupportedScriptElementException	ca.mdrobotics.rtr.gs.script.UnspupportedScriptElementException
ValueOutOfRangeException	ca.mdrobotics.rtr.gs.ValueOutOfRangeException

Appendix B: Module Internal Design (MID) sample document

ScriptGenerationManager(SGM) Module Internal Design

General Notation

1. The Module Internal Design (MID) describes the internal design decision of each access program of this module. It is usually composed of four sections, namely, *Inputs*, *Updates*, *Outputs*, and *Behaviour*.
2. The Inputs, Updates, and Outputs are defined by the following fields: *name*, external value (denoted as *ext_value*), *type*, and *Source.*
3. The *name* field is the name of the data flow (parameter name, state data name, or a parameter of another access program). In this field, ":=" means the name on its left is assigned the external value on its right; "→" denotes that the value of the boolean variable on the left indicates whether or not the external program in the external field is invoked. "True" means invoked.
   For example, "mes → messenger.displayErrorMessage(msgTitle, msg)" means the program "messenger.displayErrorMessage(msgTitle, msg)" is invoked if "mes" is true.
4. The *ext_value* field contains an external program name (if any) and its parameter name(s) in parentheses in the form "program_name(param1_name, param2_name, ...)". The parameters are defined by their own value, type and Source if necessary.
5. The *source* field shows the source/origin of the data flow:
   - If the data flow represents a global variable, the module name in which the global variable is defined is entered.
   - If the data flow represents a program call from another access program, then "Call" is entered.
   - If the data flow represents a parameter from another access program, the module name in which the access program is defined is entered.
   - If the data flow represents a parameter (or return value) belonging to the access program itself, then "Param" (or "Func") is entered.
   - If the data flow represents a local value from (or to) another design block of the access program, "Local" is entered.

6. It is assumed that all "get" programs are invoked before any processing is performed. The sequence of invocations is not significant. Similarly, all "set" programs are invoked after all processing is performed. Unless otherwise indicated, the sequence of invocations is not significant. If the sequence is important, the order of invocations is indicated by integers, 1, 2, 3, etc. A program labeled *n* is invoked before one labeled *n+1*. An indication of "*FINAL*" indicates that the program shall be the last one invoked in the access program.
7. "-" means not applicable, or do not care. "NC" refers to state data that is not changed.
8. The constants (if any) local to an access program are defined in a section named *Constants*. Constants are defined by name, type and value (or Source if it is imported).
9. The variables (if any) local to an access program which have effects on the program's behaviour are defined in the section named *Locals*. Locals are defined by the same fields as Inputs, Updates and Outputs.
10. The effects of an access program are represented through a *Program Function Table* in the *Behaviour* section.
11. There are some alternative formats of program function tables. One that we use is the *Condition Table,* with the *condition* listing all the conditions, and *result* listing values of all the output variables for each condition.
12. We also use a normal function table with the *headers* representing the conditions, and the *grids* showing the effects of an access program.
13. In most cases, function tables describe the complete behavior within a program block. This includes loops within the block. In a few cases, it is important to describe the loop itself. For example, if an output is generated within the loop body and the output is different for each iteration of the loop.

    A *while* loop within a program block can be described by a single iteration in the function table. Identifiers within the scope of the *while* loop are tagged with <n>, indicating the value of the identifier at the n^{th} iteration through the loop. Results from the loop may be tagged with <n+1>. Loop termination must be explicitly described as a boolean value in the table.
14. Simple repetition of tables with indexed identifiers is described by indexing the identifiers and including the notation "{for i = 1, 2, …, n}" as a comment in the left top cell of the table.

Lexicon

1. @Acess program #1 addScriptElement: ElemExecTime = $t_{now} - t_0$, where t_{now} = current time, t_0 = time when emulator.executeScriptElement(element) is called.

Constructor: ScriptGenerationManager

Inputs

Name	Ext_Value	Type	Source
fileManager	-	ScriptFileManager	Param
genEmulator	-	GenerationEmulator	Param
opConsole	-	OperatorConsole	Param
GUIF :=	ProjectFactory.getInstance().getGUIFactory()	GUIFactory	ca.mdrobotics.rtr.gs.vdm.gui.GUIFactory

Outputs

Name	Ext_Value	Type	Source
autosampler	-	Autosampler	State
cancelPopup	-	WaitOrCancelPopup	State
emulator	-	GenerationEmulator	State
idGenerator	-	ElementIdGenerator	State
messenger	-	MessagePopup	State
theOpConsole	-	OperatorConsole	State
theScriptFileManager	-	ScriptFileManager	State

Behaviour

```
theScriptFileManager := fileManager;
theOpConsole := opConsole;
emulator := genEmulator;
idGenerator := new ElementIdGenerator();
autosampler := new Autosampler(this, idGenerator);
messenger := GUIF.createMessagePopup(opConsole);
cancelPopup := GUIF.createWaitOrCancelPopup(theOpConsole)
```

Access Program 0: initialise

Inputs

Name	Ext_Value	Type	Source
SGC := theOpConsole idGenerator	ProjectFactory.getInstance().getGUIFactory().createScriptGenerationConsole(theOpConsole, idGenerator, this)	ScriptGenerationConsole OperatorConsole ElementIdGenerator	ca.mdrobotics.rtr.gs.ProjectFactory State State
PF → SystemModes.SCRIPT_GENERATION	ProjectFactory.getInstance().isModeSupported(SystemModes.SCRIPT_GENERATION)	ProjectFactory byte	Call ca.mdrobotics.rtr.gs.vdm.SystemModes

Updates

none

Outputs

Name	Ext_Value	Type	Source
console	-	ScriptGenerationConsole	State
SMM →	SystemModeManager.getInstance().addModeChangeListener(this)	SystemModeManager	ca.mdrobotics.rtr.gs.vdm.SystemModeManager

Behaviour

	Result	
Condition	console	SMM
PF	SGC	True
NOT(PF)	*NC*	False

Access Program 1: addScriptElement

Inputs

Name	Ext_Value	Type	Source
currentSegment	-	ScriptSegment	State
element	-	ScriptElement	Param
graphicalElement :=	element.createGraphicalScriptElement()	GraphicalScriptElement	Call
script :=	theScriptFileManager.getScript()	Script	Call
verify	-	boolean	Param

Updates

Name	Ext_Value	Type	Source
verificationCancelled	-	boolean	State

Outputs

Name	Ext_Value	Type	Source
CP →	cancelPopup.start(WAIT_DURATION, "Verifying script element...", this)	boolean	State
CS →	currentSegment.addScriptElement(element)	boolean	State
EMU →	emulator.executeScriptElement(element)	boolean	State
MES → msg	messenger.displayErrorMessage(msg)	boolean String	State Local
SCR →	script.addScriptElement(element)	boolean	Call
SFM →	theScriptFileManager.setScriptModificationTime(System.currentTimeMillis())	boolean	State
SFM2 →	theScriptFileManager.refreshScriptPreview()	ScriptFileManager	State
TOC → msg2	theOpConsole.getSystemBulletinBoard.write(msg2)	boolean String	State Local
TOC2 →	theOpConsole.getSystemBulletinBoard.write ("Added script element")	boolean	State
VW →	VirtualWorld.addGraphicalScriptElement(graphicalElement)	boolean	Call

Behaviour

Table 1.1

Condition			*Result* CP	verification Cancelled	EMU	MES	msg	TOC	msg2	CS	SCR	SFM	SFM2	TOC2	VW
NOT(verify)			False	NC	False	False	-	False	-	see tbl 1.2			see tbl 1.3		
verify	NOT(Exception)	ElemExecTime <= WAIT_DURATION OR NOT(verificationCancelled)	True	False	True	False	-	True	"Verified Ok"	see tbl 1.2			see tbl 1.3		
		ElemExecTime > WAIT_DURATION & verificationCancelled	True	True	True	False	-	True	"Verification cancelled"	False	False	False	False	False	False
	Exception	I/O Exception	False	NC	True	True	"Element verification failed"	False	-	False	False	False	False	False	False
		script = null	False	NC	True	True	"No script open"	False	-	False	False	False	False	False	False

Table 1.2

	Result		
Condition	SCR	CS	SFM
currentSegment = null	True	False	True
currentSegment ≠ null	False	True	False

Table 1.3

	Result		
Condition	VW	SFM2	TOC2
graphicalElement = null	False	True	True
graphicalElement ≠ null	True	True	True

Access Program 2: cancel

*** Implements ca.mdrobotics.rtr.gs.vdm.gui.CancellableProcess**

Inputs

none

Updates

none

Outputs

Name	Ext_Value	Type	Source
CP → 1	cancelPopup.stop()	boolean	State
GS → 2	gotoStandby()	boolean	this module
verificationCancelled	-	boolean	State

Behaviour

CP := true;

GS := true;

verificationCancelled := true

Access Program 3: getCurrentManipulator

Inputs

Name	Ext_Value	Type	Source
selectedManipulator	-	Manipulator	State

Updates

none

Outputs

Name	Ext_Value	Type	Source
getCurrentManipulator	-	Manipulator	Func

Behaviour

getCurrentManipulator := selectedManipulator

Access Program 4: setCurrentManipulator

Inputs

Name	Ext_Value	Type	Source
manipulator	-	Manipulator	Param

Updates

none

Outputs

Name	Ext_Value	Type	Source
selectedManipulator	-	Manipulator	State
SMF →	setCurrentManipulatorCommandFrame(manipulator.getCommandFrame())	boolean	this module

Behaviour

	Result	
Condition	selectedManipulator	SMF
manipulator ≠ null	manipulator	True
manipulator = null	null	False

Access Program 5: moveCurrentManipulatorJointsTo

Inputs

Name	Ext_Value	Type	Source
emulator	-	GenerationEmulator	State
jointVector	-	double[]	Param
manipulatorName :=	selectedManipulator.getIdentifier()	String	Call

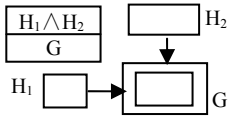
Updates

none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.moveManipulatorJointsTo(manipulatorName, jointVector)	boolean	State

Behaviour



	emulator.isConnected() = True	NOT (emulator.isConnected() = True)
emulator ≠ null	EMU = True	EMU = False
emulator = null	EMU = False	EMU = False

Access Program 6: setCurrentManipulatorJoints

Inputs

Name	Ext_Value	Type	Source
emulator	-	GenerationEmulator	State
jointVector	-	double[]	Param
manipulatorName :=	selectedManipulator.getIdentifier()	String	Call

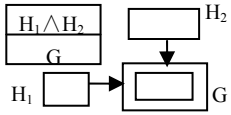
Updates

none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.setManipulatorJoints(manipulatorName, jointVector)	boolean	State

Behaviour



	emulator.isConnected() = True	NOT (emulator.isConnected() = True)
emulator ≠ null	EMU = True	EMU = False
emulator = null	EMU = False	EMU = False

Access Program 7: enableMAM

Inputs

Name	Ext_Value	Type	Source
emulator	-	GenerationEmulator	State
enable	-	boolean	Param
manipulatorName :=	selectedManipulator.getIdentifier()	String	Call

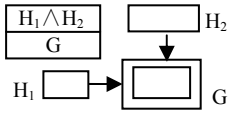
Updates

none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.enableMAM(manipulatorName, enable)	boolean	State

Behaviour



	emulator.isConnected() = True	NOT (emulator.isConnected() = True)
emulator ≠ null	EMU = True	EMU = False
emulator = null	EMU = False	EMU = False

Access Program 8: enableSJRM

Inputs

Name	Ext_Value	Type	Source
emulator	-	GenerationEmulator	State
enable	-	boolean	Param
joint	-	int	Param
manipulatorName :=	selectedManipulator.getIdentifier()	String	Call

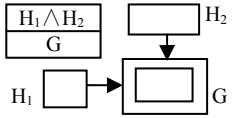
Updates

none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.enableSJRM(manipulatorName, joint, enable)	boolean	State

Behaviour



	emulator.isConnected() = True	NOT (emulator.isConnected() = True)
emulator ≠ null	EMU = True	EMU = False
emulator = null	EMU = False	EMU = False

Access Program 9: gotoStandby

Inputs

none

Updates

none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.gotoStandby()	boolean	State

Behaviour

EMU = True

Access Program 10: newScript

Block 1:

Inputs

Name	Ext_Value	Type	Source
console	-	ScriptGenerationConsole	State
script :=	theScriptFileManager.getScript()	Script	Call
sname :=	theScriptFileManager.getScript().getName()	String	Call

Updates

none

Outputs

Name	Ext_Value	Type	Source
goAhead	-	boolean	Local

Behaviour

	Result
Condition	goAhead
script ≠ null	console.promptForConfirmation(“Replace” + sname + “?”)
script = null	True

Block 2:

Inputs

Name	Ext_Value	Type	Source
goAhead	-	boolean	output of Block 1
language :=	ProjectFactory.getInstance().getScriptLanguage()	ScriptLanguage	call
messenger	-	MessagePopup	State

Updates

Name	Ext_Value	Type	Source
cancelled<>	-	boolean	Local
name<> :=	console.promptForScriptName()	String	call
validName<>	-	boolean	Local

Outputs

Name	Ext_Value	Type	Source
ERRMes →	messenger.displayErrorMessage(msgTitle, msg)	boolean	State

Locals

isValid<> :=	language.isScriptNameValid(name<>)	boolean	Call

Initialisation

cancelled := false; validName := false

Behaviour

Condition	**n = 0, 1, 2, ...**		*Result* cancelled<n+1>	validName<n+1>	ERRMes	msgTitle	msg	**Loop Termination**
goAhead	NOT(cancelled<n>) & NOT(validName<n>)	name<n> = null	True	False	False	-	-	False
		name<n> ≠ null	False	isValid<n>	NOT (isValid<n>)	"Invalid Script Name"	language.getScriptNameHint()	False
	NOT(cancelled<n>) & validName<n>		False	True	False	-	-	True
	cancelled<n> & NOT(validName<n>)		True	False	False	-	-	True
	cancelled<n> & validName<n>	***Impossible***	-	-	-	-	-	-
NOT(goAhead)			False	False	False	-	-	-

Block 3:

Inputs

Name	Ext_Value	Type	Source
goAhead	-	boolean	output of Block 1
validName *	-	boolean	output of Block 2
name *	-	String	from Block 2

* The values of "validName" and "name" should be their final values at the end of the while loop

Updates

none

Outputs

Name	Ext_Value	Type	Source
SFM → 1	theScriptFileManager.newScript(name)	boolean	State
SFM2 → 2	theScriptFileManager.refreshScriptPreview()	boolean	State
FINAL	console.refresh()	ScriptGenerationConsole	State

Behaviour

goAhead & validName → SFM & SFM2

Access Program 11: newScriptSegment

Inputs

Name	Ext_Value	Type	Source
script :=	theScriptFileManager.getScript()	Script	Call
segmentName :=	console.PromptForScriptSegmentName()	String	Call

Locals

Name	Ext_Value	Type	Source
newSegment :=	new ScriptSegment(segmentName)	ScriptSegment	Call

Updates

Name	Ext_Value	Type	Source
currentSegment	-	ScriptSegment	State

Outputs

Name	Ext_Value	Type	Source
MES → msgTitle msg	messenger.displayErrorMessage(msgTitle, msg)	boolean String String	State Local Local
newScriptSegment	-	boolean	Func
SCR →	script.addScriptSegment(newSegment)	boolean	ca.mdrobotics.rtr.gs.script.Script
SFM →	theScriptFileManager.refresheScriptPreview()	boolean	State
FINAL	console.refresh()	ScriptGenerationConsole	State

Behaviour

Condition		*Result* MES	msgTitle	msg	currentSegment	SCR	SFM	newScriptSegment
script ≠ null	segmentName ≠ null	False	-	-	newSegment	True	True	True
	segmentName = null	False	-	-	*NC*	False	True	False
script = null		True	"No script"	"A script must first be created"	*NC*	False	False	False

Access Program 12: closeScriptSegment

Inputs

none

Updates

none

Outputs

Name	Ext_Value	Type	Source
currentSegment	-	ScriptSegment	State
FINAL	console.refresh()	ScriptGenerationConsole	State

Behaviour

currentSegment = null

Access Program 13: getAllScriptSegments

Inputs

Name	Ext_Value	Type	Source
i	-	int	Local
numSegments :=	script.getNumberOfScriptSegments()	int	Call
script :=	theScriptFileManager.getScript()	Script	Call

Locals

Name	Ext_Value	Type	Source
segmentList :=	script.getScriptSegment(i) for all i in [0, numSegments-1]	ScriptSegment[]	Call

Updates
none

Outputs

Name	Ext_Value	Type	Source
getAllScriptSegments	-	ScriptSegment[]	Func

Behaviour

for i = 0, 1, ... , numSegments -1

	Result
Condition	getAllScriptSegments
script ≠ null	segmentList[i]
script = null	segmentList[0]

Access Program 14: getCurrentScriptSegment

Inputs

Name	Ext_Value	Type	Source
currentSegment	-	ScriptSegment	State

Updates

none

Outputs

Name	Ext_Value	Type	Source
getCurrentScriptSegment	-	ScriptSegment	Func

Behaviour

getCurrentScriptSegment := currentSegment

Access Program 15: getScriptName

Inputs

Name	Ext_Value	Type	Source
script :=	theScriptFileManager.getScript()	Script	Call
sname :=	theScriptFileManager.getScript().getName()	String	Call

Updates

none

Outputs

Name	Ext_Value	Type	Source
getScriptName	-	String	Func

Behaviour

	Result
Condition	getScriptName
script ≠ null	sname
script = null	“”

Access Program 16: setCurrentSegment

Inputs

Name	Ext_Value	Type	Source
i	-	int	Local
newCurrent	-	ScriptSegment	Param
numSegments :=	script.getNumberOfScriptSegments()	int	Call
script :=	theScriptFileManager.getScript()	Script	Call

Locals

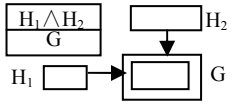
Name	Ext_Value	Type	Source
foundSegment :=	script.getScriptSegment(i) for i = 0, 1, ..., numSegments -1	ScriptSegment	Call

Updates

Name	Ext_Value	Type	Source
currentSegment	-	ScriptSegment	State

Outputs
none

Behaviour



	script ≠ null		script = null
	foundSegment = newCurrent	NOT (foundSegment = newCurrent)	
newCurrent ≠ null	currentSegment = newCurrent	No action	No action
newCurrent = null	currentSegment = newCurrent		

Access Program17: moveCurrentManipulatorPORTo

Constants

Name	Value	Type	Source
ReferenceFrame.PYR	0	byte	ca.mdrobotics.math.EulerFrame

Inputs

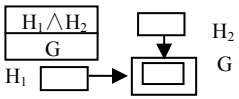
Name	Ext_Value	Type	Source
emulator	-	GenerationEmulator	State
manipulatorName :=	selectedManipulator.getIdentifier()	String	Call
newPOR	-	ReferenceFrame	Param
PX :=	newPOR.getX()	double	Call
PY :=	newPOR.getY()	double	Call
PZ :=	newPOR.getZ()	double	Call
PP :=	newPOR.getPitch(ReferenceFrame.PYR)	double	Call
Pya :=	newPOR.getYaw(ReferenceFrame.PYR)	double	Call
PR :=	newPOR.getRoll(ReferenceFrame.PYR)	double	Call

Updates none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.moveManipulatorPORTo(manipulatorName, PX, PY, PZ, PP, Pya, PR)	boolean	State

Behaviour



	emulator.isConnected() = True	NOT (emulator.isConnected() = True)
emulator ≠ null	EMU = True	EMU = False
emulator = null	EMU = False	EMU = False

Access Program 18: setCurrentManipulatorPOR

Constants

Name	Value	Type	Source
ReferenceFrame.PYR	0	byte	ca.mdrobotics.math.EulerFrame

Inputs

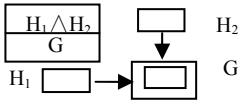
Name	Ext_Value	Type	Source
emulator	-	GenerationEmulator	State
manipulatorName :=	selectedManipulator.getIdentifier()	String	Call
newPOR	-	ReferenceFrame	Param
PX :=	newPOR.getX()	double	Call
PY :=	newPOR.getY()	double	Call
PZ :=	newPOR.getZ()	double	Call
PP :=	newPOR.getPitch(ReferenceFrame.PYR)	double	Call
Pya :=	newPOR.getYaw(ReferenceFrame.PYR)	double	Call
PR :=	newPOR.getRoll(ReferenceFrame.PYR)	double	Call

Updates none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.setManipulatorPOR(manipulatorName, PX, PY, PZ, PP, Pya, PR)	boolean	State

Behaviour



	emulator.isConnected() = True	NOT (emulator.isConnected() = True)
emulator ≠ null	EMU = True	EMU = False
emulator = null	EMU = False	EMU = False

Access Program 19: getCurrentManipulatorCommandFrame

Inputs

Name	Ext_Value	Type	Source
SM :=	selectedManipulator.getCommandFrame()	CommandFrameNode	State
VW :=	VirtualWorld.getInstance().getBaseCommandFrame()	CommandFrameNode	Call

Updates

none

Outputs

Name	Ext_Value	Type	Source
getCurrentManipulatorCommandFrame	-	CommandFrameNode	ca.mdrobotics.rtr.gs.vdm.devices.CommandFrameNode

Behaviour

	Result
Condition	getCurrentManipulatorCommandFrame
selectedManipulator ≠ null	SM
selectedManipulator = null	VW

Access Program 20: setCurrentManipulatorCommandFrame

Inputs

Name	Ext_Value	Type	Source
manipulatorName :=	selectedManipulator.getIdentifier()	String	Call
newCommandFrameNode	-	CommandFrameNode	Param
selectedManipulator	-	Manipulator	State

Updates

none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.setManipulatorCommandFrame(manipulatorName, newCommandFrameNode)	boolean	State
SM →	selectedManipulator.setCommandFrame(newCommandFrameNode)	boolean	State
VW →	VirtualWorld.setCommandFrame(newCommandFrameNode)	boolean	Call

Behaviour

	Result		
Condition	EMU	SM	VW
selectedManipulator ≠ null	True	True	True
selectedManipulator = null	False	False	False

Access Program 21: modeChanged

*** Implements ca.mdrobotics.rtr.gs.vdm.SystemModeChangeListener**

Inputs

Name	Ext_Value	Type	Source
newMode	-	byte	Param
SystemModes.SCRIPT_GENERATION	-	byte	ca.mdrobotics.rtr.gs.vdm.SystemModes

Updates

none

Outputs

Name	Ext_Value	Type	Source
EMU →	emulator.setWorldState(VirtualWorld.recallWorldState(SystemModes.SCRIPT_GENERATION))	boolean	State
TOC →	theOpConsole.getSystemBulletinBoard.write("* Script Generation Mode *")	boolean	State
FINAL	console.setVisible(activeMode)	boolean	State

Behaviour

	Result	
Condition	TOC	EMU
newMode = SystemModes.SCRIPT_GENERATION	True	True
NOT(newMode = SystemModes.SCRIPT_GENERATION)	False	False

Access Program 22: configureAutosampler

Inputs

Name	Ext_Value	Type	Source
console	-	ScriptGenerationConsole	State

Updates

none

Outputs

Name	Ext_Value	Type	Source
AUT →	autosampler.configure(console)	boolean	State
MES →	messenger.displayErrorMessage(“Cannot configure autosampler”)	boolean	State

Behaviour

	Result	
Condition	AUT	MES
NO Exception	True	False
Exception	False	True

Access Program 23: enableAutosampling

Inputs

Name	Ext_Value	Type	Source
doEnable	-	boolean	Param

Updates

none

Outputs

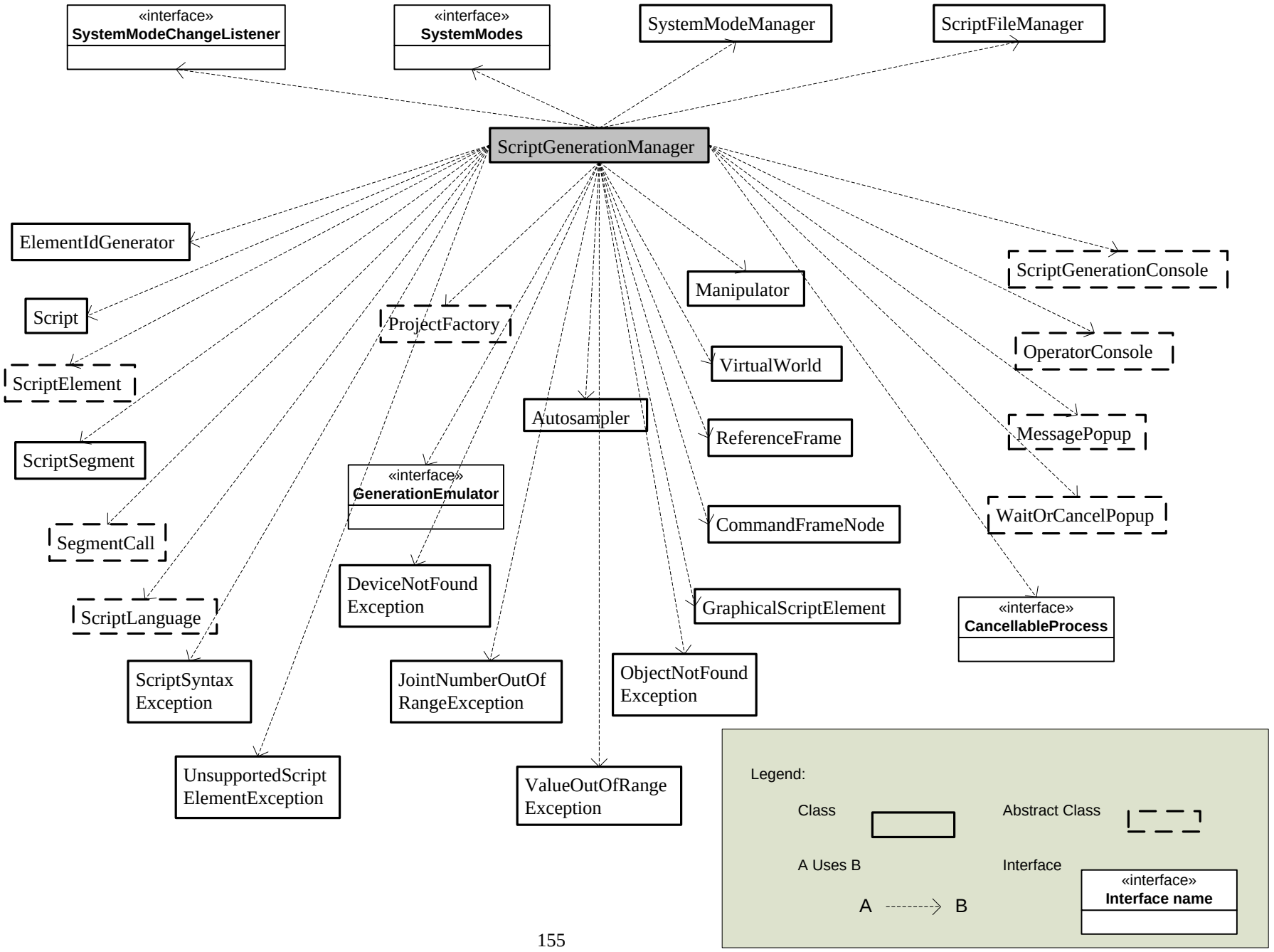
Name	Ext_Value	Type	Source
AUT →	autosampler.enable(doEnable)	boolean	State
enableAutosampling	-	boolean	Func
MES →	messenger.displayErrorMessage(“Cannot enable/disable autpsampler”)	MessagePopup	State

Behaviour

	Result		
Condition	AUT	MES	enableAutosampling
NO Exception	True	False	doEnable
Exception	-	True	False

Appendix C: The Uses Relation of ScriptGenerationManager Class

ScriptGenerationManager Class Uses Relation Graph



ScriptGenerationManager Class Uses Relation Graph – all levels